Co-funded by the
Erasmus+ Programme
of the European Union

We're on the Web!

Visit us at:

https://makers-project.eu

USING PYTHON TO ENABLE CREATIVE EXPLORATIONS OF 3D MODELS

Creative Commons licence - Attribution-NonCommercial-ShareAlike CC BY-NC-SA

Year of publication: 2022

Editors: Nektarios Moumoutzis and Aristarchos Tzatzos

Project "MAKER SCHOOLS: Enhancing Student Creativity and STEM Engagement by Integrating 3D Design and Programming into Secondary School Learning" (Agreement no. 2020-1-BG01-KA201-079274)

Contents

1. Introduction 5
1.1 Abbreviations 7
1.2 Getting to know the Micro:bit 7
1.2.1 What is micro:bit 7
1.2.2 Front side overview 8
1.2.3 Back side overview 9
1.2.4 Edge Connector and Pinout 10
1.2.5 Expansion Breakout Board 11
1.3 The EduBlocks programming environment 11
2. Measuring Time - First Steps in Python using Micro:bit LEDs and Buttons 14
2.1 Aim 14
2.2 Synopsis 14
2.3 Theory 14
2.3.1 The Python programming language 14
2.3.2 Functions and modules in Python / EduBlocks 14
2.3.3 Variables and loops 15
2.3.4 Data types 17
2.3.5 Micro:bit LED screen 17
2.3.6 Micro:bit buttons 18
2.4 Practice 19
2.5 Time for fun 22
2.6 Self check 23
3. Fill the screen - Wireless communication among micro:bit boards 25
3.1 Aim 25
3.2 Synopsis 25
3.3 Theory 25
3.3.1 Making choices in Python 25
3.3.2 Logical expressions in Python 28
3.3.3 Functions in Python 29
3.3.4 Images 30
3.3.5 Radio 31
3.3.6 Accelerometer 31
3.4 Practice 33
3.5 Time for fun 41

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

3.6 Self check 41

4. Battleship - Deepen Your Knowledge about Python and Micro:Bit 43

4.1 Aim....................43

4.2 Synopsis....................43

4.3 Theory 43

4.3.1 Global vs Local variables....................43

4.3.2 String manipulation 45

4.3.3 Objects 47

4.3.4 Functions as objects 47

4.4 Practice 48

4.5 Time for fun 56

4.6 Self check 56

5. A Quiz Game - Advanced Topics in Python with Micro:Bit 59

5.1 Aim 59

5.2 Synopsis....................59

5.3 Theory 59

5.3.1 Lists and tuples 59

5.3.2 Dictionaries 60

5.3.3 For - enumerate 61

5.3.4 Random numbers 62

5.3.5 *Try-except*....................62

5.3.5 Connecting an external speaker with micro:bit....................63

5.4 Practice 63

5.4.1 True-False quiz game 63

5.4.2 Multiple choice quiz game....................66

5.5 Time for fun 69

5.6 Self check 69

6. Introduction to Raspberry Pi Pico 71

6.1 Aim....................71

6.2 Synopsis....................71

6.3 Theory 71

6.3.1 What is the Raspberry Pi Pico 71

6.3.2 Pinout 71

6.3.3 Micro:bit vs Raspberry Pi Pico 72

6.3.4 Classes 73

6.3.5 Electronic components....................74

Co-funded by the
Erasmus+ Programme
of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

6.3.6 Interrupts 75
6.3.7 Thonny Python IDE 76
6.4 Practice 78
6.5 Time for fun 83
6.6 Self check 84
References 86

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

1. Introduction

This is one of the two modules of the Extracurricular Training Program "Python for 3D printing and creative explorations of 3D models" developed in the framework of the Erasmus+ MAKERS project (https://makers-project.eu/). In the spirit of the project's focus on cross-curricular approaches, and taking into account the target groups' expressed interests, Intellectual Output 2 showcases how STEM education can combine 3D design and printing with programming/coding, including coding of selected microcontrollers that can be used in 3D designs to develop mobile robots, and various automation and physical computing creations.

Programming and coding are among the key skills that are currently targeted in both curricular and extracurricular STEM education. Students start learning basic Computer Science and programming already at the start of high school. Extracurricular activities such as Coding Clubs are among the most popular after-school STEM activities. More recently, the popularity of combining programming and the Arts has increased (not least due to the availability of accessible software like Scratch and App Inventor), and digital creativity is becoming a major theme of science outreach events. All in all, there is little doubt that learning various programming languages benefits both career prospects and generic skills like problem solving, persistence, and collaboration. The benefits are even more pronounced when Programming can be combined with design and creativity.

Offering opportunities to combine coding with 3D design and printing can help strengthen the programming skills of students, while also building skills for design. For such opportunities to be realized in schools, however, both teachers and students need teaching/learning resources and guidance.

Within this overall framework, the Extracurricular Training Program "Python for 3D printing and creative explorations of 3D models" aims to provide tools, resources, and support for combining Programming/Coding and 3D technology in extracurricular STEM education to achieve engaging and effective learning in both fields. The resources can be used by teachers to design their own training, as well as by students who want to learn on their own.

The expected impact can be summarized as follows:

- Enable and motivate Computer Science teachers to design and deliver extracurricular STEM or STEAM training which exploits the Python programming language for both producing and creatively exploring 3D models
- Motivate and enable students to learn how Python can be used in emerging technologies such as 3D
- Raise awareness about the benefits of combining Programming and 3D technology for more effective STEM learning.

The methodology applied to the design and development of both modules is based on:

- *Combination of theoretical training and sample exercises, for which sample code has been developed*: The training program provides theoretical training by introducing relevant Python syntax and data structures (module A), as well as the interface and functions of OpenSCAD (module B), including the use of Python libraries with the software. Each major element in the theoretical presentation is followed by tasks for the learners. Learners are assisted with the sample code. They are able to follow the procedures step-by-step, learning in the process. Additional exercises are suggested for practice.

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

- *Learner assessment guidance*: Each module defines the knowledge and skills that are developed and how they can be measured. Sample tests are provided to help teachers and to allow for self-assessment on the part of individual learners.

The rest of this document contains a number of sections that describe activities designed to be offered in sessions of 3-6 hours each, depending on the level of knowledge and skills targeted. To offer this flexibility in terms of duration, each section points to certain extensions that can be done beyond the minimum duration of 3 hours. The first section is introductory and contains an overview of the micro:bit microcontroller (components of the micro:bit board, edge connectors, pinout, expansion breakout board) and the EduBlocks visual programming environment that enables the creation of MicroPython using click-and-snap graphical blocks like Lego bricks. Following this introductory section, the material presented is organized as follows:

- Measuring Time - First Steps in Python using micro:bit LEDs and Buttons: Overview of Python Language, functions and modules, variables and loops, data types. Micro:bit LED screen, buttons.
- Fill the Screen - Wireless Communication among micro:bit boards: Making choices in Python, logical expressions, functions. Micro:bit images, radio, accelerometer, wireless communication.
- Battleship - Deepen your Knowledge about Python and micro:bit: Global vs local variables, string manipulation, objects, functions as objects. More elaborate wireless communication among micro:bits.
- A Quiz Game - Advanced topics in Python with micro:bit: Lists and tuples, dictionaries, for / enumerate, random numbers, *try-except*. Connecting external speakers to the micro:bit.
- Introduction to Raspberry Pi Pico: Raspberry Pi Pico overview, pinout, comparison with micro:bit. Classes, electronic components, interrupts, Thonny IDE.

All sections follow the same organization

- **Aim:** The aim of the activity presented in each section linking to specific topics in Python programming language on the one side and microcontroller/electronic components.
- **Synopsis:** An overview of the specific project that will be developed in terms of its functionality and use.
- **Theory:** Details about the corresponding programming structures presented as well as physics and/or electronic components used and their operation.
- **Practice:** A step-by-step development of the specific project with all the necessary explanations and triggers for observations to enhance the understanding of the topics covered.
- **Time for fun:** Triggers for further work on the project to develop interesting extensions that help the students deepen their knowledge and further develop their skills and competencies.
- **Self-check:** A multiple choice quiz that helps the students assess the skills and competencies they have developed.

This module can be effectively combined with the Module “Using Python for Procedural 3D Content Generation for 3D Printing”.

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

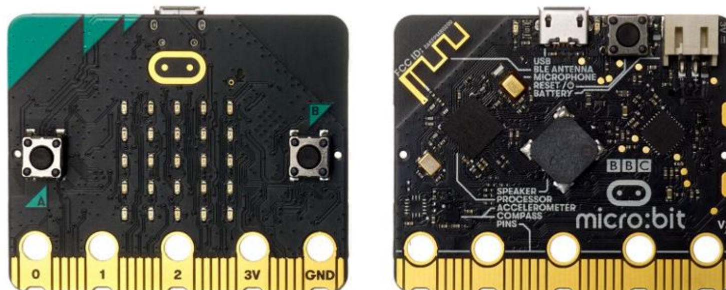
The module is transferable to non-formal learning contexts and youth work. It would not require a high level of prior knowledge. It is especially transferrable to VET training.

1.1 Abbreviations

ADC	Analog-to-digital
BBC	British Broadcasting Corporation
GND	Ground or Earth
GPIO	General Purpose Input/Output
HEI	Higher Education Institutions
IDE	Integrated Development Environment
LED	Light Emitting Diode
PC	Personal Computer
PWM	Pulse-width Modulation
SE VET	Secondary Education VET
SPI	Serial Peripheral Interface
STEAM	Science, Technology, Engineering, Arts and Mathematics
STEM	Science, Technology, Engineering and Mathematics
USB	Universal Serial Bus
VET	Vocational Education and Training

1.2 Getting to know the Micro:bit

1.2.1 What is micro:bit



Micro:bit is an open-source embedded system designed by the British Broadcasting Corporation (BBC) for use in computer education. It is a single development board with a microcontroller and various sensors, buttons, LEDs, and GPIOs included. It is really flexible when it comes to programming since it supports modern programming languages such as Python and JavaScript. There are also some block-based editors that offer an intuitive graphical user interface for beginners, much like the popular Scratch platform, one of which we are going to see in the next section.

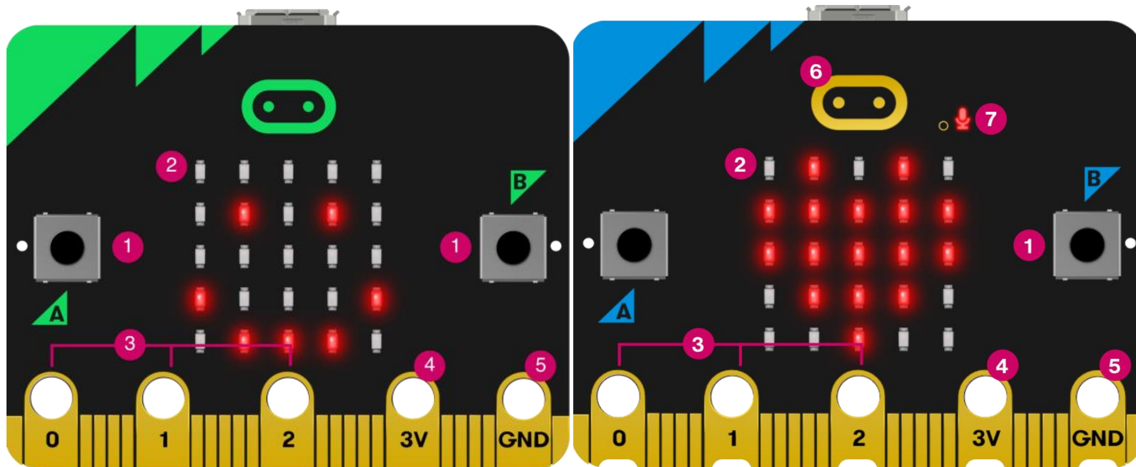
The next two subsections provide an overview of the micro:bit board we are going to use, showing all of its components and their function. As of 2021, there are two versions of the

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

board, with their key difference being the addition of a speaker, a microphone, and a touch-sensitive button to the newest version. The overview that follows presents both versions.

1.2.2 Front side overview

The figure below depicts the front side of micro:bit V1 on the left and micro:bit V2 on the right. All numbered elements are explained below the figure.



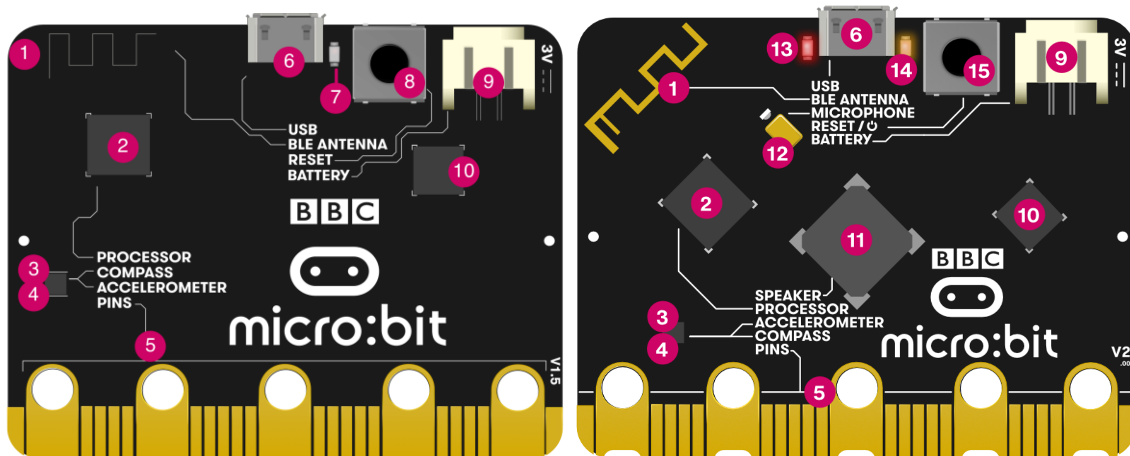
1	Two programmable buttons A and B that can be assigned in code.	5	GND is the ground/Earth pin. It is used to complete electric circuits when you use external components.
2	25 LEDs arranged in a 5x5 matrix make up a display that can be used to show words, numbers, or images. The display can also be used as a sensor that measures how much light is falling on it.	6	(V2 only) A touch-sensitive button that can be used in the same way as buttons A and B.
3	Pins - GPIO that can be used to attach external components such as headphones, motor drivers, more buttons, etc. They can also sense touch if they are set up accordingly.	7	(V2 only) Microphone LED to indicate the sound levels that the microphone picks. Next to it is a small hole leading to the actual microphone.
4	3V power pin that can be used to power external electronics.		

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

1.2.3 Back side overview

The figure below depicts the back side of micro:bit V1 on the left and micro:bit V2 to the right. All numbered elements are explained below the figure.



1	Radio & Bluetooth antenna used to communicate with other micro:bits by radio and with other Bluetooth devices.
2	Processor & temperature sensor. The processor is the brain of the micro:bit and the sensor can be used to sense the environmental temperature.
3	Compass that can measure the strength of magnetic fields or find magnetic North.
4	Accelerometer that measures forces in 3 dimensions including gravity so we can determine the orientation of the board. It can also be used as a shake sensor.
5	Pins - GPIO that can be used to attach external components such as headphones, motor drivers, more buttons, etc. They can also sense touch if they are set up accordingly.
6	Micro USB port used to connect the micro:bit to a computer so that it can be programmed.
7	Single LED that flashes when a program is being downloaded to the micro:bit or stays on to indicate that the board is powered.
9	Battery socket that can be used to power the micro:bit with two AAA batteries or 3V, instead of the USB port.
10	USB interface chip that is used to flash the new code to the micro:bit and communicate with a computer.
11	(V2 only) Speaker that can be programmed in code.
12	(V2 only) Microphone that can be used to interact with the board by voice or sound.
13	(V2 only) Red power LED indicating that the board is powered.
14	(V2 only) Yellow USB LED indicating communication with a computer by flashing.
15	(V2 only) Reset & power button. Press once to restart the micro:bit. Press and hold to put it in power-saving mode when used with batteries and press once

Co-funded by the Erasmus+ Programme of the European Union

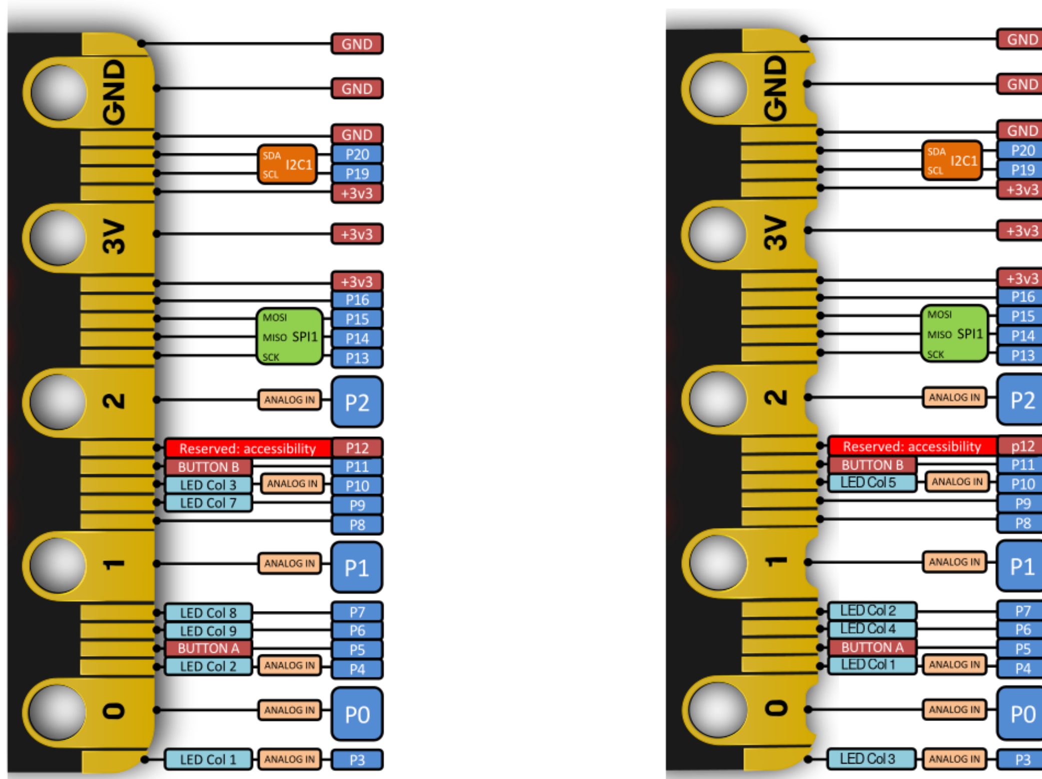
The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

again to wake it.

8 Reset button to restart micro:bit.

1.2.4 Edge Connector and Pinout

The following figure presents in detail the edge Pins-GPIO section of the micro:bit (V1 on the left and V2 on the right).



LED Col	These pins are reserved for the display. In order to use them, we must first disable the display's driver by calling display.enable(false). To turn the display back on we call display.enable(true)	ANALOG IN	These pins can be used to read analog values of voltages applied to them. For example, to read pin 1 we use pin1.read_analog()
BUTTON A & B	These pins are connected to the onboard buttons of the micro:bit.	Pins 13-15	These pins can be configured for communication with other devices using the Serial Peripheral Interface (SPI) bus.
P12	This pin is reserved for accessibility and must never be used.	Pins 19 & 20	These pins can be configured for communication with other devices using the I^2C bus protocol.

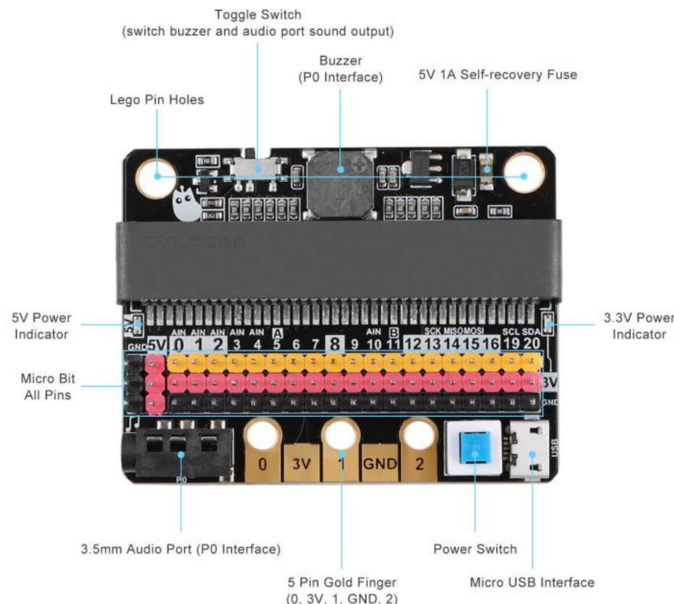
Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

GND	These pins are the ground point connection.	+3V3	These pins are the supply voltage of 3.3 Volts.

1.2.5 Expansion Breakout Board

The pins of the edge connector of the micro:bit are not so useful due to their very small size. To deal with this problem, an expansion breakout board can be used, which can be purchased separately and is shown in the image below.



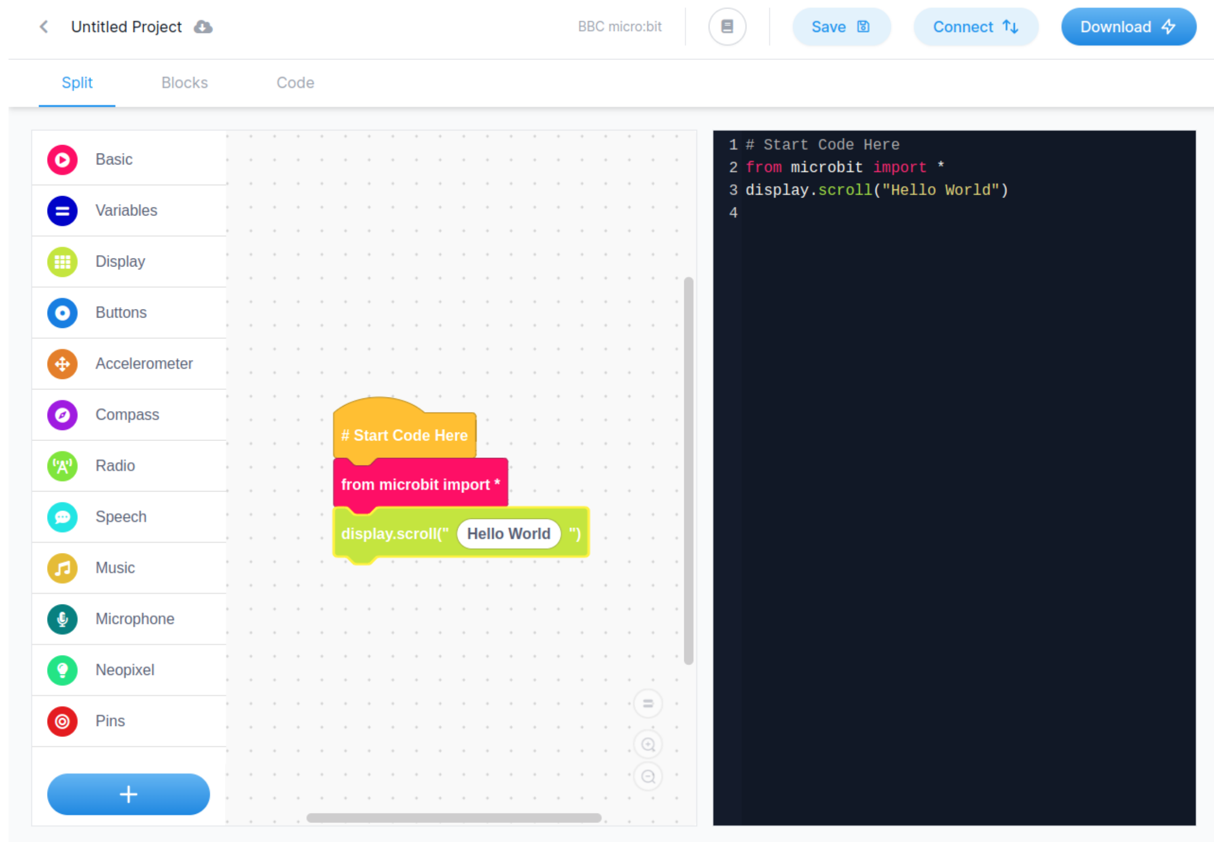
This expansion board is designed to hold the micro:bit module in the big black connector and break out all the pins below it. The pins can be accessed using the yellow headers which have their corresponding pin number written above them. Also, the pins in the header are accompanied by pairs of 3.3 V and GND pins which can be used to power the external devices we want to use with micro:bit. There are many different models of the expansion breakout board. The one depicted above has some more functionality built into it. For example, we have a 3.5 mm audio jack, a built-in speaker (Buzzer), an on-off switch, a micro USB port, and a toggle button for the speaker.

1.3 The EduBlocks programming environment

EduBlocks is an online editor for programming the micro:bit and other development boards using the Python programming language. To begin, navigate to https://edublocks.org/ and click the "Get Started" button. Then, you should select the "BBC micro:bit" under the "Create New Project" section. Now you are ready to use the EduBlocks editor to program the micro:bit. It can be used in Split, Blocks or Code view by selecting the appropriate tab under the name of the project (initially, the name is "Untitled Project"). The Blocks view presents the code in graphical blocks, while the Code view presents it in text form. The Split view combines both and it is the default option when you start the EdubBlocks editor. It is advisable to use this option since it shows in real-time the generated Python code from the graphical blocks.

Co-funded by the
Erasmus+ Programme
of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



We can see that in the Blocks view as well as in the corresponding area of the Split view, all commands are categorized and color-coded in order to better indicate their function category. Blocks can be easily selected from the categories on the left and dragged in the white area, where they can be connected with other blocks to form scripts (chunks of code). One or more scripts compose our program which can be downloaded as a file and saved to the micro:bit's memory. Once a program file is saved successfully, the micro:bit can run the code and we can see the outcome and/or interact with it.

To flash a program to the micro:bit we first have to connect it to our computer through a micro USB cable. Let us see how: Recreate the following simple code in EduBlocks. The corresponding blocks can be found in the categories Basic and Display. Connect your micro:bit with a micro USB cable to your computer, click the "Connect" button on Edublocks and make the appropriate selection in the pop-up window that will appear. Once your code is ready and your micro:bit is connected properly, click the "Flash" button. Once the code is uploaded to the micro:bit, a smiley face will appear on the built-in LED screen.

In the text-based form of the code we can start writing our code from scratch or edit the code generated by transforming whatever blocks have been previously entered in block-based form. The text-based form is very useful because it helps us move from block-based programming to usual text-based programming in the Python programming language. Block-based programming

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

is much easier for beginners because they do not have to remember all the syntactic details of the programming language. Experts prefer text-based programming because they can develop their code faster. Consequently, the most effective learning path for novices is to start with block-based programming and gradually move to text-based programming. EduBlocks is an ideal environment for such a learning path.

Once we learn a general-purpose programming language, such as Python, we have the basic knowledge to program anything that we can imagine, from applications or games that run on our computer to projects that use micro:bit or other microcontrollers. Even though text-based programming can be a bit more difficult to learn and memorize, we can really benefit from it.

The learning activities 2, 3, 4, and 5 of this module take advantage of the EduBlocks support for a gradual transition from block-based to text-based programming. Both the block-based and the text-based code are presented so that they can be easily compared and used in the Edublocks editor. Consequently, this module is to be used as an introduction to the micro:bit board as well as to the Python programming language. The module ends with a learning activity that introduces the Raspberry Pi Pico board, which is more powerful than the micro:bit but has fewer built-in components.

Happy Python coding!

Co-funded by the
Erasmus+ Programme
of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

2. Measuring Time - First Steps in Python using Micro:bit LEDs and Buttons

2.1 Aim

The aim of this activity is to get familiar with the EduBlocks Programming Environment and start using some basic operations, functions, and commands of the Python programming language as well as micro:bit LED matrix and buttons. Our coding project will be a simple game to test the player's reflexes by measuring the time needed to press a micro:bit button as soon as a countdown finishes.

2.2 Synopsis

In this activity we will develop a simple game that measures the player's response time to visual stimulus. The game will show a countdown from three (3) to zero (0) on the micro:bit's display at the end of which the player has to quickly press buttons A or B. After that, the game will calculate the player's reaction time and finally display it.

2.3 Theory

2.3.1 The Python programming language

A computer program can be considered as a sequence of instructions that are executed one after the other. These instructions could present information on the screen, receive user input from the keyboard, process data and store the results in computer memory or even pause the execution of the program for a certain time duration.

There are many programming languages that can be used to develop a program. Python is one of the most interesting ones to learn due to its simplicity and popularity. It was created by Guido van Rossum in the '90s. It is a general-purpose programming language, meaning that it can be used to develop any kind of computer program. It is an interpreted language, meaning that most of its implementations execute instructions directly one by one, without the need to compile a Python program as a whole into **machine language** instructions. The Python interpreter transforms the Python program into an **intermediate language** which is again translated into machine language that is executed.

One of the merits of Python when it comes to physical computing is its object orientation. Object-oriented Programming is a programming paradigm that addresses the issue of structuring programs by providing the means to specify properties and behaviors bundled into individual **objects**. It is an approach for modeling concrete, real-world things as software objects, which have some data associated with them and can perform certain functions. This way, electronic devices such as LEDs, sensors, etc., can be very effectively modeled as objects and easily manipulated through software.

2.3.2 Functions and modules in Python / EduBlocks

To be able to use certain functions in a Python program, we should import the corresponding libraries that provide the specifications of the desirable functions. One such library is the microbit library which contains everything needed to program the micro:bit board. By importing the micro:bit library the Python interpreter essentially pre-loads every function or any other object that is available for programming the micro:bit. The syntax of such an import can be seen below, both in text-based and block-based code.

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
from microbit import *
```

The statement begins with the keyword *from*, followed by the name of the library and the keyword *import*. Lastly, we specify the functions or other objects we want to import from this library separated with commas. In case we want to import everything, we use the asterisk character * as above.

In this activity, in order to create the countdown feature of the game, we have to pause the execution of the code several times. To do so, we will use the sleep function. Its syntax is shown below.

```
sleep(1000)
```

To use this function, we write its name sleep followed by opening and closing parentheses. Inside the parentheses, we type a number that indicates the time to pause the program in milliseconds. 1000 is 1 second, so the statement above, when executed by a program, will cause a delay of 1 second before the execution of the next statement below it.

To measure the time that elapsed between two time points, we need to record the time t1 of the first time point and the time t2 of the second time point and then compute the difference t2 - t1. To do so in Python we have to import the utime library as shown below.

```
import utime
```

This library offers a function to read the value of an onboard counter which is increasing in microseconds and wraps around after some value. Note that in order to transform a certain number of microseconds into seconds we have to divide it by 1,000,000 (one second corresponds to 1000 milliseconds and 1 millisecond corresponds to 1000 microseconds). Here is the function that will read the value of the time counter:

```
utime.ticks_us()
```

We observe that this block has an oval shape, which is different from the shape of the previous blocks we used. The round shape indicates that this block corresponds to a certain value that is returned by the corresponding function. This value can then be used in an expression. The shape of blocks is always related to their use and is complementary to placeholders of similar shapes that can accept such blocks.

2.3.3 Variables and loops

In programming, whenever we have to keep a value that we will later display or do some kind of mathematical operation with it, we have to place it in a reserved position in the computer's memory. All programming languages facilitate memory storage of values via variables. Variables are essentially human-readable names that correspond to certain memory positions. We can retrieve and store a value in a variable using its name. The term variable stems from the word *vary* which means that something changes, which in the case of a variable is its contents. We can

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

declare a variable in Python just by assigning a value to it and afterward in the code we can increase, decrease or reassign a new value to this variable. Below we declare a variable named my_first_variable and we assign the value 6 to it using the = operator.

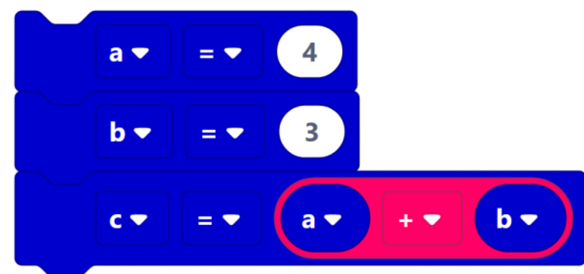
```
my_first_variable = 6
```

Now we can, for example, increase its value by 2. This is done by using the increase operator which is noted as +=. By doing so, we increase the previously saved value of the variable named my_first_variable which was 6, by adding 2 and we save the result again in the same variable. From now on when we use the variable my_fist_variable in our code, its value will be 8.

```
my_first_variable += 2
```

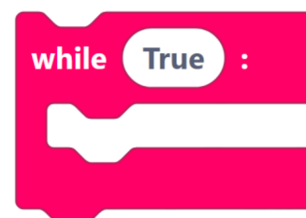
To access a variable's contents, we have to use the oval block with the variable's name in it, as shown below. As we saw earlier, this oval shape corresponds to blocks that produce a certain value and can be used to build complex expressions using operators such as the arithmetical operators for addition, subtraction, multiplication, and division. Let us see how: In the code below, we declare two variables named a and b and assign to them the values 4 and 3 respectively. Then we add a + b and assign the sum to a new variable named c. We observe the difference between the blocks that assign a value to a variable and those that read the value of a variable.

```
a = 4
b = 3
c = a + b
```



Sometimes in our code we want to repeat a set of instructions. We can do so using a *while loop* as shown below:

```
while True:
```



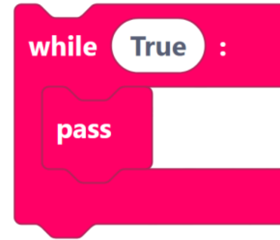
Whatever statements we put inside such a block will be repeated forever because we have used the keyword True as the condition of the loop. If we replace True with another condition, we could repeat the loop as long as the condition is True. For example, we will see how we can use a condition to execute a *while* loop as long as a certain button is not pressed.

To indicate that a *while* loop does not do anything while it repeats itself, we put inside it the statement pass as shown below. Such a *while* loops is useful when we want to stall our code execution until an event happens, for example, a button is pressed.

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
while True:
  pass
```



Instead of the pass statement, in a *while* loop that does not do anything while it repeats, the continue statement can be used.

Note that in text-based code, all statements within a loop should be indented with one tab character.

2.3.4 Data types

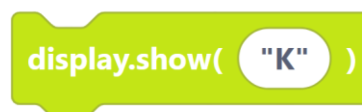
Any value produced by an oval block has a data type. The most common data types that we will use are:

- **Strings** are sequences of characters. Double quotes or single quotes can be used for defining them: "a12", 'nick'. "" (or '') is the empty string, a string with zero characters. Any letter, number, or symbol character can be used in a string.
- **Integers** are numeric values, positive or negative, such as 12, -1, 0, and -34. You can combine integer values to create complex arithmetic expressions using the familiar operators of addition (+), subtraction (-), multiplication (*), division (/) as well as the remainder (%) operator to compute the remainder of the division between two integers.
- **Floating point numbers**, positive or negative, such as 3.14, -19.0, 0.0, and -34.4533. You can combine floating-point numbers the same way as integers using arithmetic operations. You can also compute the closest integer part of a floating-point number x using int(x). For example, int(3.2) and int(3.9) will both give the value 3.
- **Booleans** are two special values - True and False - that represent if something is valid (true) or invalid (false). More about this type of value will be presented in subsequent activities.

2.3.5 Micro:bit LED screen

The micro:bit has an LED screen that consists of an array of 5 x 5 LEDs. It can be used to show lots of interesting stuff. If we want to display a single character (letter, number, or symbol) we use display.show(). Between the parentheses, we give as a string the character we want to display.

```
display.show("K")
```



If we execute again display.show() for a different character, the display first clears itself and then shows the new character. In the case that we want to clear the screen manually, we use the display.clear() without passing any parameter.

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
display.clear()
```

If we want to display more than one character, for example, the word Hello!, display.show() will display one character at a time which is not always what we want. That is why there is display.scroll() which scrolls the text we give between the parentheses.

```
display.scroll("Hello!")
```

In EduBlocks there are two options, one using quotes "..." which is used for strings, and one without which is used for numbers.

Lastly, we can manipulate each pixel of the screen individually. That means we can pick an intensity level from 0 to 9 for any pixel we want, with the 0 being off and 9 as the brightest setting. To do so we use the display.set_pixel(). Below is an example for setting the center-most pixel to full intensity. The first parameter is the *x* position, the second one is the *y* position and the third one is the intensity.

```
display.set_pixel(2, 2, 9)
```

The index for the *x*, *y* coordinates of the center pixel is 2, 2 instead of 3, 3 because the counting of the elements in the array along each row and column starts at index 0. We start counting from the upper left corner of the screen that corresponds to coordinates 0, 0.

2.3.6 Micro:bit buttons

The micro:bit board is equipped with 2 buttons that are labeled A and B. We can use them to enable us to interact with a running program so that it is possible to control its behavior during its execution. There are two methods we can use for the buttons: The first one is the is_pressed() method which returns True when the button is currently being pressed. To use it, we first select the name of the button we want to check, as shown below:

```
button_a.is_pressed()

button_b.is_pressed()
```

The other method is was_pressed() which returns True if the button was pressed from the time we powered on the micro:bit, or from the last time, we called was_pressed(), whichever is most recent.

```
button_a.was_pressed()

button_b.was_pressed()
```

Co-funded by the
Erasmus+ Programme
of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

2.4 Practice

As it is described in the Synopsis, the game of this section starts with a countdown from 3 to 0. This is the first thing that we are going to implement in code. The first statement in our code should import the microbit library so that we could use its functions and objects:

```
from microbit import *
```

The above statement, when executed, will pre-load every function or object that is included in the microbit library. This is an absolutely necessary statement in our code. Without it most of the functionalities of micro:bit could not be used, and therefore our game will not work.

Now that we imported everything we need, we can start coding the countdown. The numbers will be shown on the micro:bit's display. To do so we will use display.show():

```
display.show()
```

This allows us to show on the display, whatever value we put between the parentheses. In our code, we will put the numbers of the countdown. Below is the code needed to display the numbers 3 and 2. Extend this code and make the appropriate additions in order to display the numbers from 3 to 0. Upload the final code to the micro:bit.

```
from microbit import *

display.show(3)
display.show(2)
```

We observe that after some flashing, the display shows only the last number of our countdown. This happens because all the numbers are displayed one after the other so fast that our eyes cannot perceive them. To address this problem, we can put a small delay before presenting every number. This is done with the sleep(). In the parentheses, we put the amount of time we want our code to be delayed in milliseconds, which in our case will be 1 second or 1000 milliseconds. Below is the final code with delays, which you can further elaborate on by adding the necessary statements to display numbers 1 and 0 after 3 and 2.

Co-funded by the
Erasmus+ Programme
of the European Union

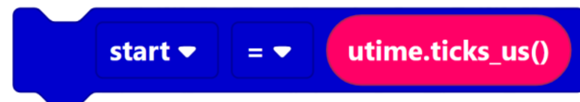
The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
from microbit import *

display.show(3)
sleep(1000)
display.show(2)
```

Now that our countdown is working, we can implement the time measuring part. In order to calculate how many milliseconds elapsed between the number 0 is shown and the player pressing a button, we need two timestamps. We need to know at what time the number 0 was shown and at what time the player pressed any button. After we acquire these two time measurements, we subtract the first from the second: the result will be the time elapsed between the two events. For example, let us say that the number 0 was shown after 3000 milliseconds (from the micro:bit's power-on) and the player responded by pressing a button at 3700 milliseconds. If we subtract 3000 from 3700 we are left with 700 milliseconds, which is the time elapsed.

Now that we know what information we need, we can add to our code the following line. This line is placed right after the command that shows the number 0 on the display. It saves the time (in milliseconds) that elapsed from the last micro:bit's power-on, to a variable named start which, as the name implies, is the start of our time measurement.

```
start = utime.ticks_us()
```



In order to test that our time measurement concept works as we intend, we can try to measure a predefined period. To do so, we can put the micro:bit to sleep for 1 second and then take the second timestamp and save it to a variable named stop. After this is done, we can calculate the time that elapsed from start to stop, by subtracting the start from the stop and saving the result in a variable named time. Then we can show the result on the micro:bit's display, using the command display.scroll(time) instead of display.show(time) because the display can only fit one digit at a time. The code needed to perform this test is shown below and you can compare it to the code you wrote so far, in order to confirm that it is right.

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
from microbit import *

display.show(3)
sleep(1000)
display.show(2)
sleep(1000)
display.show(1)
sleep(1000)
display.show(0)
start = utime.ticks_us()
sleep(1000)
stop = utime.ticks_us()
time = stop - start
display.scroll(time)
```

We observe that after the countdown is finished, a number scrolls on the display which is really close to 1000 and that indicates that our code measures the time from start to stop rather accurately.

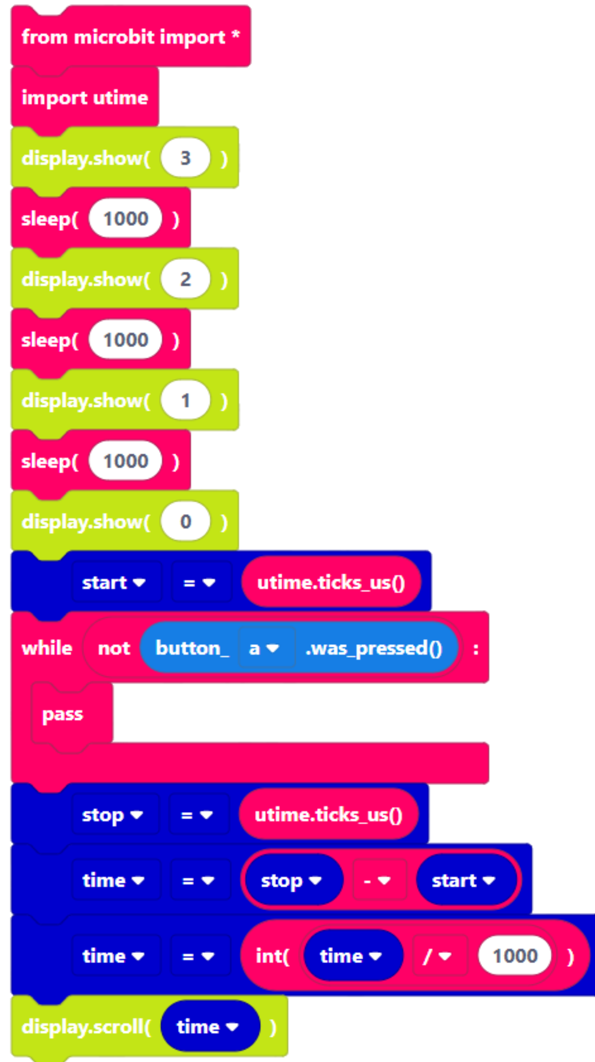
To complete our game, we have to replace the sleep(1000) statement with another one that stalls the execution of code in that position until the player pushes a button. This can be done using a *while* loop, that loops without doing anything while there are no button presses registered. To indicate that a *while* loop does not do anything we put a pass statement inside it. After a button is pressed, we exit the *while* loop and the execution continues with the rest of the code.

Let us select button A for the user to press. The method is_pressed() returns True if the Button is pressed or False if it is not. We would like the *while* loop to continue iterating as long as the method returns False and stop iterating when it returns True. This corresponds to the logical negation of the return value of the method. To compute the negation of a logical value we use the not operator. We will see more logical operators in the next learning activity. For now, it is sufficient to know how the not operator works and how to use it in our code. The finished code is given below and you can download it to your micro:bit in order to play the game

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
from microbit import *
import utime
display.show(3)
sleep(1000)
display.show(2)
sleep(1000)
display.show(1)
sleep(1000)
display.show(0)
start = running_time()
while not button_a.is_pressed():
    pass
stop = running_time()
time = stop - start
display.scroll(time)
```



2.5 Time for fun

Here are some ideas for extending the game:

1. The game in its current state is executed only once. You can surround the whole code with a while True loop to execute the game indefinitely.

2. Revise the previous extension so that the game ends when the player presses button B.

3. Revise the game so that after the countdown the player has to press button A 5 times.

4. You can easily see that the countdown part of the code essentially repeats 3 times the process of presenting a number and sleeping for 1 second before it finishes to 0. Could you replace this part of the code with an appropriate *while* loop that counts down using a new variable to gradually reduce this variable and display its value on the LED matrix? This way your code is much more elegant and modular. You can then very easily change the initial value of the countdown from 3 to any number you desire.

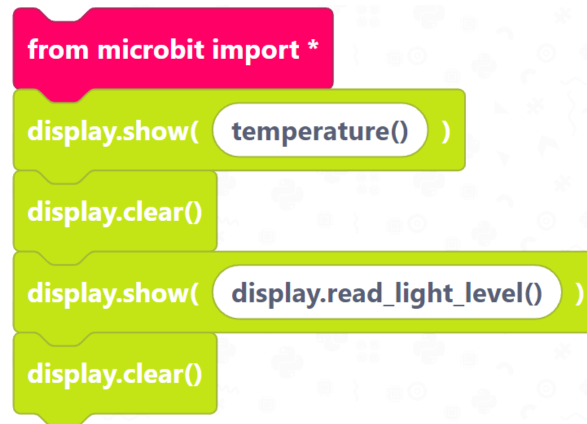
5. You can use your micro:bit to measure the temperature or the light intensity of your environment. Try to download and run the following code on your micro:bit. It will display on the LED matrix the current temperature and light intensity:

```
from microbit import *
```

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
display.show(temperature())
display.clear()
display.show(display.read_light_level())
display.clear()
```



You can use temperature() and display.read_light_level() in many different creative ways to make your game more interesting. For example, you may use the current temperature as the initial value of the countdown variable in the previous extension. Or you could use the light intensity to compute the delay (sleep time) between the steps in the countdown in combination with arithmetic operators. For example, a sleep time of 100000/display.read_light_level() will cause a slower countdown in a darker environment.

2.6 Self check

1. *While* loops are executed while their condition is:
    a. True
    b. False
    c. True or False
    d. They are executed only once
2. Between the parentheses of sleep(), we put the amount of time in:
    a. Seconds
    b. Microseconds
    c. Milliseconds
    d. Hours
3. The correct way to measure the time passed from start to stop is to:
    a. Subtract stop from start
    b. Add start and stop
    c. Divide stop with start
    d. Subtract start from stop
4. To display a number or text that is bigger than the screen can show we use:
    a. display.show()

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

b. display.show() for each character

c. display.scroll()

d. display.showAll()

5. To include everything from a library into our code we use:

a. **from** libraryName **import** everything

b. **from** libraryName **import** *

c. **from** libraryName **import** all

d. **import** all **from** libraryName

6. To get how many milliseconds the micro:bit is running since its last power-on we call:

a. running_time()

b. up_time()

c. executing_time()

d. on_time()

7. display.show() is used to show on the display of micro:bit:

a. Letters

b. Numbers

c. Images

d. All of the above

8. To invert a condition we use:

a. The keyword invert in front of it

b. The keyword invert after it

c. The keyword no in front of it

d. The keyword not in front of it

9. The instructions that are meant to be part of a loop should be:

a. Put inside brackets { }

b. Indented with a tab

c. Put inside parentheses ( )

d. None of the above

10. To indicate that a loop does not perform any task, we put in its body:

a. The control statement continue

b. The control statement pass

c. The control statement break

d. The control statement continue or pass

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

3. Fill the screen - Wireless communication among micro:bit boards

3.1 Aim

The aim of this activity is to get more in-depth knowledge of Python. We will learn some new useful functions, data types, and control statements. We will also learn about wireless communication in the micro:bit and we will create our own function. Our coding project will be a multiplayer board game that we can upload to the micro:bit to play with our friends.

3.2 Synopsis

The aim of the game is to compete with other players, each one using his/her micro:bit, to light up the whole display by turning on one pixel at a time. The first player to light up his/her whole display wins. Each player has to tilt the micro:bit in order to move a bright red indicator and select the pixel he/she wants to turn on by pressing the button A. The pixels that have been already turned on are indicated by a low-intensity red light. The player that is the first to fill the whole display wins and his micro:bit transmits the message "I WON" to the other boards. The winner's display shows the letter W and every other player's display shows L, indicating that they lost this round.

3.3 Theory

3.3.1 Making choices in Python

There are cases when our programs need to take certain actions depending on certain conditions related to the value of a variable, the state of a sensor, and many more. In such a case our code will check if the corresponding condition is True or False and will decide what to do next depending on the result. There are two Python statements that can be used to make choices: The *if* statement and the *while* statement. We have already seen the *while* statement. Let us see now the *if* statement.

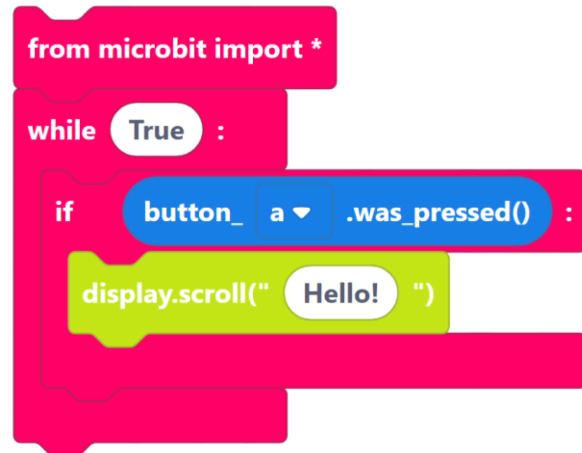
Let us assume we have a game that asks for the user to give an answer to a question and the player has the possibility to give 3 wrong answers at most before he loses. We can store the number of wrong answers in a variable and using the *if* statement we can check to see if the number of wrong answers is three, in which case we stop the game. The code that is inside an *if* statement is executed only when a certain condition is True.

Let us see now another example of using the *if* statement to control the behavior of micro:bit depending on the input from a button. We want the micro:bit to scroll on the display the word "Hello!" each time we press the button A. In a *while True* loop, we put an *if* statement which checks if the button A was pressed and in that case scrolls the word "Hello!". Try the code below:

Co-funded by the
Erasmus+ Programme
of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

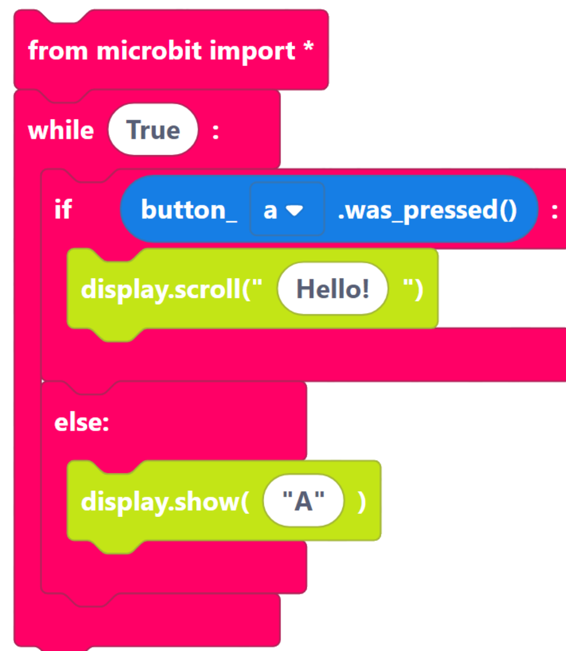
```
from microbit import *
while True:
  if button_a.was_pressed():
    display.scroll("Hello!")
```



How does this code work? It constantly checks if button A is pressed. If it is not, the *while True* loop will continue. If button A is indeed pressed, the display.scroll("Hello!") statement will be executed and after that the *while True* loop will continue again.

An *if* statement can be extended using an *else* statement which is executed when the condition of the *if* statement is false. Let's add an *else* statement to the previous code so that when button A is not pressed, the letter A is shown on the display to remind the player what button to press.

```
from microbit import *
while True:
  if button_a.was_pressed():
    display.scroll("Hello!")
  else:
    display.show("A")
```



The *if* statement can be further extended with one or more *elif* statements. Let us assume that we want to create a program specifying the age group of a person using his/her age. The age to age group correspondence we want is the following:

- age below 3: **baby**
- age 3 or above and below 12: **kid**
- age 12 or above and below 18: **teenager**
- age 18 or above: **grown up**

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

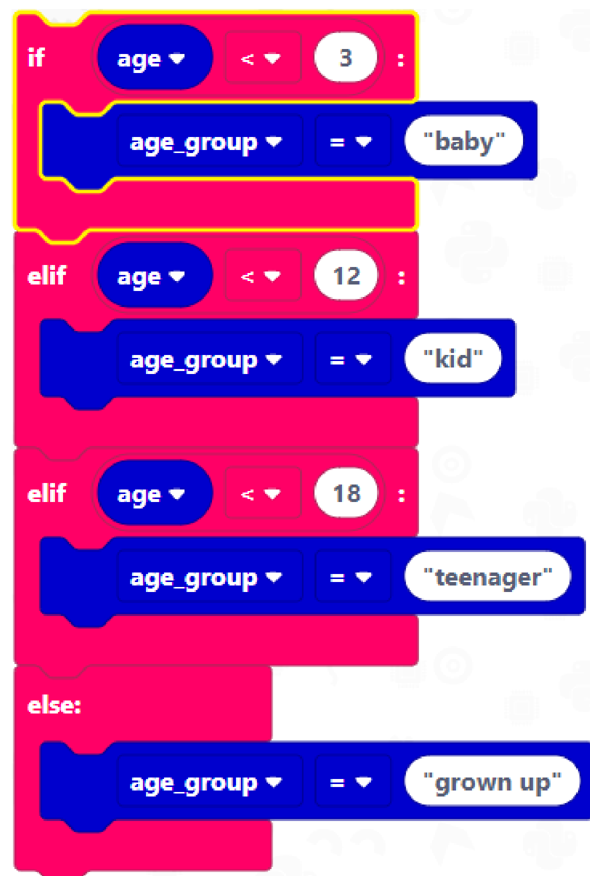
The code below shows how this processing is done using the *if*, *elif* and *else* statements. The person's age is stored in the age variable and the age group in the age_group variable. It is interesting to study how this code actually works. The first statement checks if the value of the age variable is less than 3 and, if it is, it assigns the value "baby" to the age_group variable and the code terminates.

If the value of the age variable is 3 or more, the condition of the initial *if* statement is False, and the first *elif* statement is executed. In other words, when the first *elif* statement is executed, the value of the age variable is guaranteed to be 3 or more. Then the condition age<12 is checked. If this is True, it means that the value of the age variable is 3 or above (due to the condition of the initial *if* statement) and below 12. In that case, the value "kid" is assigned to the age_group variable and the code terminates.

If the value of the age variable is 12 or more, the condition of the first *elif* statement is False, and the second *elif* statement is executed. In other words, when the second *elif* statement is executed, the value of the age variable is guaranteed to be 12 or more. Then, the condition age<18 is checked. If this is True, it means that the value of the age variable is 12 or above (due to the conditions checked in initial *if* and first *elif* statements) and below 18. In that case, the value "teenager" is assigned to the age_group variable and the code terminates.

If the value of the age variable is 18 or more, the condition of the second *elif* statement is also False, and the final *else* statement is executed. In other words, when the *else* statement is executed, the value of the age variable is guaranteed to be 18 or more. In that case, the value "grown up" is assigned to the age_group variable and the code terminates.

```
if age < 3:
  age_group = "baby"
elif age < 12:
  age_group = "kid"
elif age < 18:
  age_group = "teenager"
else:
  age_group = "grown up"
```



The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

3.3.2 Logical expressions in Python

Both *if* statements and *while* statements rely on logical expressions. Let us see how these expressions are structured. Simple logical expressions can be variables that have a Boolean type value, i.e., True or False. Furthermore, they could be comparisons of arithmetic values using the comparison operators <, <=, ==, >=, >, or != for testing the relationship between two numbers. The following table presents the details.

Operator	Description	Example
==	It evaluates to True if the values of two operands are equal.	(1 == 2) is False (3 == 3) is True (3 == 2) is False
!=	It evaluates to True if the values of two operands are not equal.	(1 != 2) is True (3 != 3) is False (3 != 2) is True
<	It evaluates to True if the value of the left operand is less than the value of the right operand.	(1 < 2) is True (3 < 3) is False (3 < 2) is False
<=	It evaluates to True if the value of the left operand is less than or equal to the value of the right operand.	(1 <= 2) is True (3 <= 3) is True (3 <= 2) is False
>	It evaluates to True if the value of the left operand is greater than the value of the right operand.	(1 > 2) is False (3 > 3) is False (3 > 2) is True
>=	It evaluates to True if the value of the left operand is greater than or equal to the value of the right operand.	(1 >= 2) is False (3 >= 3) is True (3 >= 2) is True

The examples shown in the table above use arithmetic values. However, comparisons can be used between values of other data types as well. For example, strings can be compared also using their lexicographic ordering. Here are some examples:

"George" > "Geography" is True

"George" < "Gorge" is True

"George" != "Geography" is True

Logical expressions can be combined into more complex ones using Boolean operators. The following table presents them in detail:

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Operator	Description	Example
not p	It evaluates to True if p is False and False if p is True.	(not 1 < 2) is False (not 2 >= 3) is True
p and q	It evaluates to True if both p and q are True. False otherwise.	(1 != 2 and 3 == 3) is True (3 != 3 and 1 < 2) is False
p or q	It evaluates to True if p or q is True. False otherwise.	(1 != 2 or 3 < 2) is True (3 != 3 or 3 < 2) is False

The meaning of the Boolean operators *not*, *and*, *or* can be further explained using the so-called truth tables. A truth table gives the result value when applying a Boolean operator depending on the values of its operands. All possible combinations of values are included in such tables:

A	**B**	**A** and **B**	**A** or **B**	not **A**
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

It is evident from the above table that the *and* operator is True only when both its two operands are True, and it is False in all other cases. The *or* operator is True when any of its two operands is True, and it is False when both of them are False. The *not* operator takes only one operand and inverses its value: If the operand value is True, it becomes False and vice versa.

An example of the application of the *and* operator in real life would be the existence of a rainbow. In order to have a rainbow we must have both rain and sun, in the absence of any of them or both no rainbow can be seen.

An example of the use for the *or* operator in real life would be the decision to take an umbrella with us. We take an umbrella with us when either if the forecast shows rain in our area or when it is actually raining right now. In the case that there is no rain right now and the forecast shows a sunny day we don't take an umbrella with us.

Finally, an example of the use of the *not* operator in real life is a photo sensor that turns on or off the lights of our house depending on the existence of light outside the house. When there is light outside, the inside lights are off and when there is no light outside, the inside lights are on.

To summarize, a logical expression in Python consists of one or more logical or comparison operators and true or false statements. We use these expressions in *if* statements or in *while* loops.

3.3.3 Functions in Python

Functions are an important construct that helps us better organize code in Python. Functions also help in reusing code fragments, thus making the programmer's work easier and more

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

efficient. We have already used several functions until now. Let us see, then, how to create our own functions.

Functions are essentially fragments of code that can be called and that usually return some results after they finish their execution. They can also affect the status of a program or change the state of electronic elements that are connected to the computer. This is the way that we will use them in this lab activity.

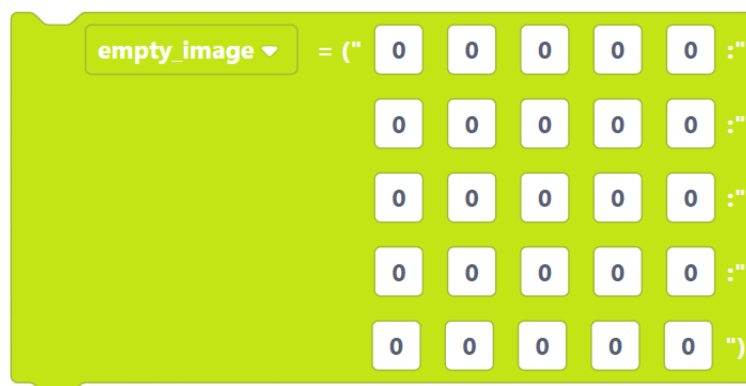
To define a function in Python, we used the *def* keyword. We provide a name for the function and a list of parameters inside parentheses. The list of parameters may be empty if we want to. In such a case the function will not receive any input when it is called. The code of the function follows indented with one tab character. To return a value from a function and terminate its execution we use the keyword *return* along with the value or an expression that computes a value to be returned.

As you already know, to call a function, just type its name and put in parentheses a list with the values for the function parameters. If the function does not accept any parameter, put nothing inside the parentheses.

3.3.4 Images

To control the LED display of the micro:bit we need to use a special *class* named Image. We have not yet seen classes in Python but for the moment we can consider them as special data types, just like strings, integers, and Booleans. What the Image class does is to provide a mechanism that we can use to hold a 5x5 array of numbers from 0 to 9 which correspond to each pixel of the display and indicate its respective intensity. We can use Images to prepare something that we want to display or to hold the state of a game that is played on the screen. To create an Image, we should create an instance of the Image class using a string of 25 numbers grouped in five rows separated by colons ':' (more about classes and instances will be explained in following learning activities). Below we create an empty Image and assign it to a variable named empty_image.

```
empty_image = Image("00000:00000:00000:00000:00000")
```



We can edit the Image using set_pixel(). It takes the same arguments as display.set_pixel(), which are *x* position, *y* position, and intensity value. Also, we can get the intensity value of a pixel of an Image using get_pixel() with the *x* and *y* position of the pixel as arguments. Lastly, we can fill an entire Image with the same intensity using fill() and the intensity value as an argument.

3.3.5 Radio

The micro:bit has an onboard radio antenna that can be used to communicate with other micro:bit boards. We can enable this feature by including the radio library and executing radio.on() in our code. radio.off() turns it off.

```
radio.on()
```

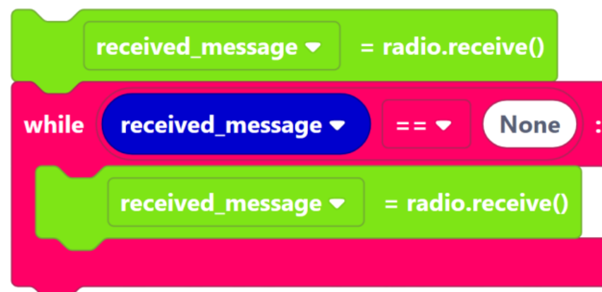
To send data we use radio.send(message) where the message is a string like "Hello!". To receive data, we use radio.receive() and assign the value to a variable. In the case that there was no message transmitted, the radio.receive() will return the special value None.

```
radio.send("Hello!")
```

```
received_message = radio.receive()
```

Usually, when we wait for a response and we want our program to pause until the response is received, we use a technique called *polling*: Checking the value of the received_message repeatedly until the desired message appears. The following code demonstrates this process:

```
received_message = radio.receive()
while received_message == None:
  received_message = radio.receive()
```

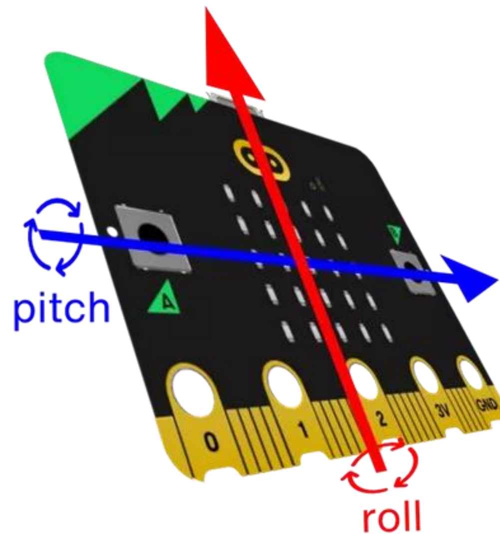


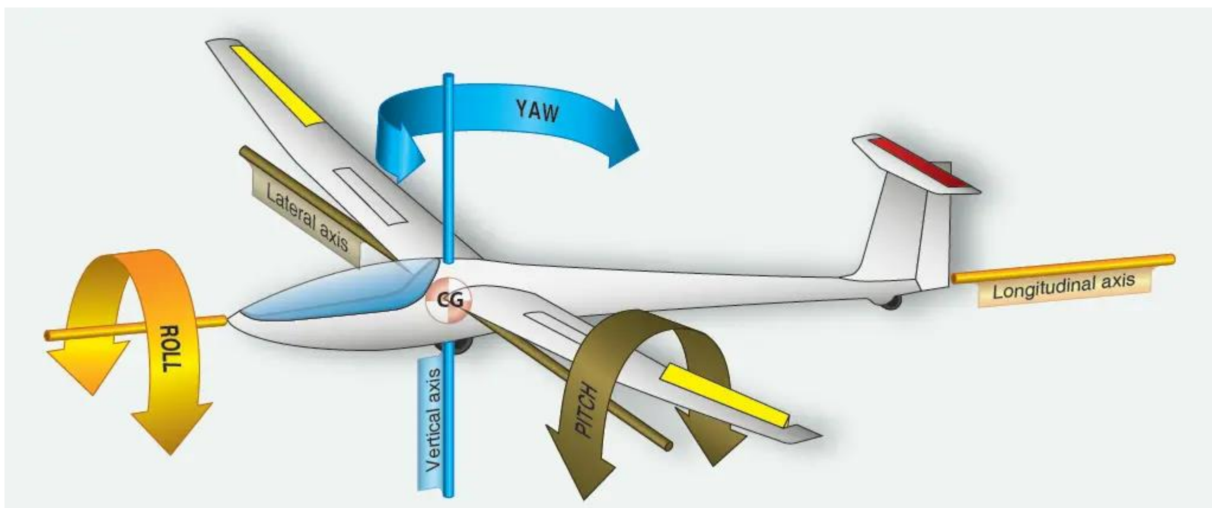
3.3.6 Accelerometer

Another very useful component of micro:bit is the built-in accelerometer. This component can detect the orientation of the board in the hyperplane (3D plane), meaning that it can fairly accurately assert the orientation on the axes of Pitch, Roll, and Yaw. The following two pictures demonstrate the position of the axes on the micro:bit, together with an example of an airplane so that the significance of each axis is clear.

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.





In the instruction set of EduBlocks, the Roll axis is indicated with the letter *x*, the Pitch axis with the letter *y*, and the Yaw axis with the letter *z*. When we roll the board to the right the value of *x* is positive and if we roll it to the left it becomes negative. In the same way, when we pitch the board backward the value of *y* is positive and if we pitch it forwards it becomes negative. To acquire these values, we use accelerometer.get_x(), accelerometer.get_y() and accelerometer.get_z() that can be found in the Accelerometer category.

```
accelerometer. get_x()
accelerometer. get_y()
accelerometer. get_z()
```

The accelerometer can also detect some gestures by using accelerometer.was_gesture() and accelerometer.is_gesture().

```
accelerometer.was_gesture( 'shake' )
accelerometer.is_gesture( 'shake' )
```

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

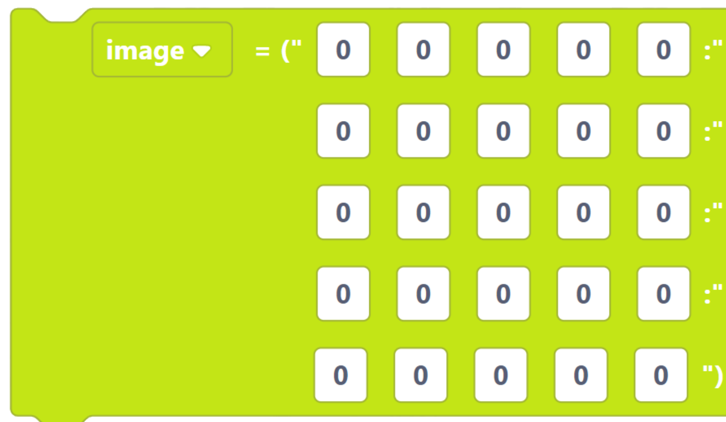
3.4 Practice

Let us begin our code by first importing the libraries that we will use, which are the microbit and radio libraries, as shown below.

```
from microbit import *
import radio
```

Then we have to create an Image variable that will hold the current state of the game and will be used to display the players' progress on the micro:bit LED matrix. We initialize the Image with zeros so that when we display it every pixel will be turned off.

```
image = Image("00000:00000:00000:00000:00000")
```



Now it is about time we created our first function! It will be a function that will be responsible for the selection of a single-pixel on the display by tilting the micro:bit. We declare it as shown below and we name it fillPixel.

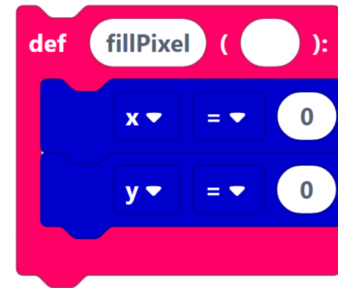
```
def fill_pixel():
```

First, in the function's body, we declare two variables *x* and *y*, which will hold the coordinates of the pixel to be selected. Both of these variables will be initialized to 0. This will set the position of the indicator of the pixel to be selected at the upper-left corner of the display.

Co-funded by the
Erasmus+ Programme
of the European Union

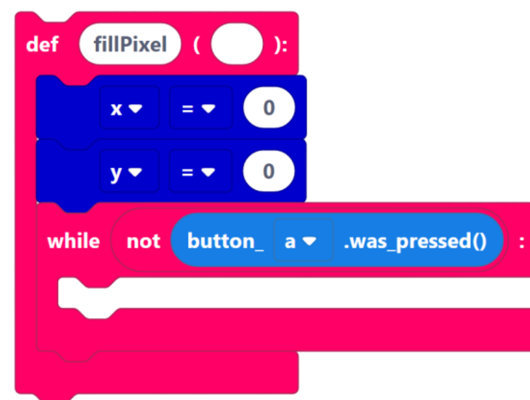
The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
def fill_pixel():
        x = 0
        y = 0
```



The next thing we have to do is to repeat the pixel selection code until the player presses the button A. To do so, we will use a *while* loop. This loop will just check when button A will be pressed. To create such a condition, we put the button_a.was_pressed() method call inside a *not* operator. These will force the *while* loop to continue execution until the button A is actually pressed.

```
def fill_pixel():
        x = 0
        y = 0
        while not button_a.was_pressed():
```



Inside the *while* loop's body, we check the orientation of the micro:bit board using its onboard accelerometer. Using the *x* and *y* orientation we can determine if the board was tilted in the Roll or Pitch axes respectively. When we roll the board to the right the value of *x* is positive and if we roll it to the left it becomes negative. In the same way, when we pitch the board backward the value of *y* is positive and if we pitch it forwards it becomes negative.

We also have to account for the position of the indicator, using the *x* and *y* variables we declared above. For example, if the indicator is at the topmost of the display and we continue to pitch the board forwards we don't want to change the value of *y* anymore.

Considering all of the above we create one *if-elif* statement to control the movement along the Pitch axis and one *if-elif* statement to control the movement along the Roll axis. In the condition of the *if* statement we check if the board is rolled more than 200 units to the right and if the indicator hasn't reached the rightmost of the display (meaning $x<4$). If the condition is True, we increase the value of *x* by 1. If the previous condition is False, the *elif* statement is executed. The new condition checks if the board is rolled less than -200 units to the left and if the indicator hasn't reached the left most of the display (meaning $x>0$). If this condition is True, we decrease the value of *x* by 1.

```
if accelerometer.get_x() > 200 and x < 4:
  x += 1
elif accelerometer.get_x() < -200 and x > 0:
  x -= 1
```

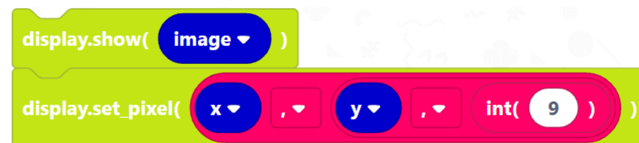
Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

We follow the same steps for the second set of *if* and *elif* statements which will be responsible for the pitch axis and the *y* position of the indicator.

After we assign the new position of the indicator to the variables *x* and *y*, we have to show it on the micro:bit LED matrix. Also, we have to show the player's progress that is kept in the image variable. Then, we set the pixel of the indicator to the maximum intensity which is the value 9.

```
display.show(image)
display.set_pixel(x,y,int(9))
```



The last thing we have to do in the body of the *while* loop is to introduce a delay. This is essential because the repetition occurs so fast that it is practically impossible to control the position of the indicator. A delay of 200 milliseconds is adequate.

```
sleep(200)
```

To complete our function we have to update our progress that is saved in the image variable every time the player selects a new pixel by pressing the button A. When the player pressed this button the execution of the *while* loop stops and the execution continues to the next blocks of code. To update the image, we have to use a function block to call image.set_pixel(). For the arguments, we give the values *x* and *y* of the indicator and the intensity of the light as 6. We set the intensity as 6 because otherwise, the indicator will not differentiate from the other pixels.

```
def fill_pixel():
        x = 0
        y = 0
        while not button_a.was_pressed():
                if accelerometer.get_x() > 200 and x < 4:
                        x += 1
                elif accelerometer.get_x() < -200 and x > 0:
                        x -= 1
                if accelerometer.get_y() > 200 and y < 4:
                        y += 1
                elif accelerometer.get_y() < -200 and y > 0:
                        y -= 1
                display.show(image)
                display.set_pixel(x,y,int(9))
                sleep(200)
```

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

image.set_pixel(x,y,int(6))

```
def fill_pixel ( ):
    x = 0
    y = 0
    while not button_a.was_pressed() :
        if accelerometer.get_x() > 200 and x < 4 :
            x += 1
        elif accelerometer.get_x() < -200 and x > 0 :
            x -= 1
        if accelerometer.get_y() > 200 and y < 4 :
            y += 1
        elif accelerometer.get_y() < -200 and y > 0 :
            y -= 1
        display.show( image )
        display.set_pixel( x , y , int( 9 ) )
        sleep( 200 )
    image.set_pixel ( x , y , int( 6 ) )
```

Now that we created our new function, we can test its functionality by calling it using the function call block with our function's name in it.

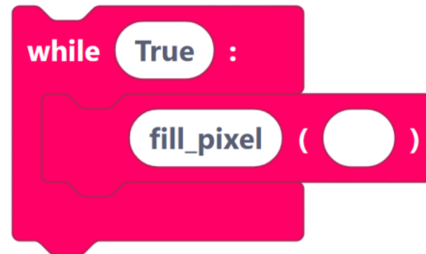
fillPixel()

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Load the code to the micro:bit and try to fill some pixels. What do you observe?

You should be able to observe that we can only fill one pixel and after that, the game is not responding. This happens because we only call our function once and after it finishes the program ends. To resolve this problem, we can put the function call in a *while True* loop as shown next:

```
while True:
  fill_pixel()
```



Try loading the game now and fill the whole display. What do you observe?

You should be able to observe two things, one is that the indicator always returns to the upper left corner after each round and the other is that the game never stops, even after the whole display is filled. To solve this problem, we have to modify the condition of the *while* loop in order to stop either when the whole display is filled or when another player has done so (remember that this game is to be played by a number of players with their micro:bits communicating with each other).

To be able to stop the game when another player has filled the display in his/her micro:bit, it is necessary to extend our code so that the micro:bit boards used by the players can communicate with each other. To do so, we first turn the radio communication on and save to a variable the result the radio.receive(). If the result is empty, the value None is saved to our variable. This is the code we can use:

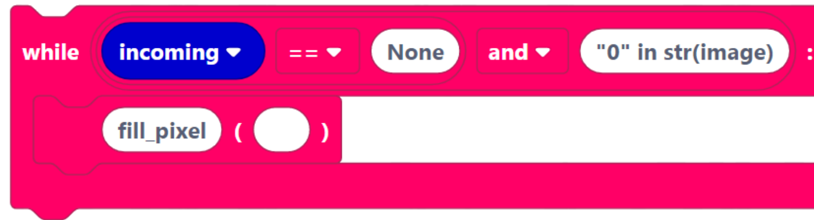
```
radio.on()
incoming = radio.receive()
```

The *while* loop should continue executing while no other player has filled the whole display, i.e., while the value of the variable incoming is None. We also check using the Boolean and operator that we have not filled our own display, which can be determined by checking for the presence of zeros in the image. This is checked with the condition "0" in str(image). The str(image) transforms the image contents into a string and then we use the in operator to check if the character representing zero is contained in the string. The code now is as follows:

```
while incoming == None and "0" in str(image):
        fill_pixel()
```

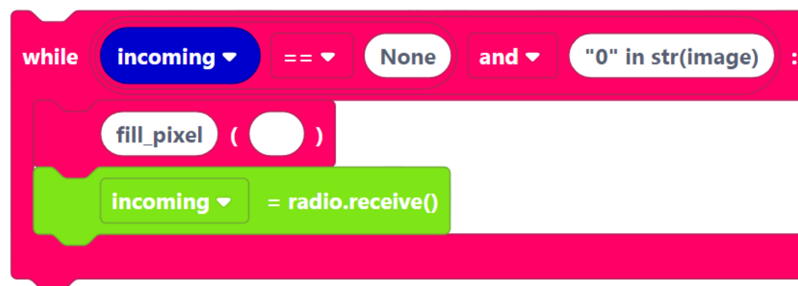
Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



In the body of the *while* loop we already call the function, we created but we have to add one last thing. That is, to update the contents of the incoming variable by calling again radio.receive(). This is important in order to know if any other player will finish before us during the next game round. This is the final code:

```
while incoming == None and "0" in str(image):
        fill_pixel()
        incoming = radio.receive()
```



In the case that either we filled our display or another player has done so, the *while* loop terminates and the rest of the code is executed. By checking to find what caused the *while* loop to terminate, our code can determine if the player has won or lost via an *if else* statement to check the value of the incoming variable. If the variable has the value None, that means that the player has won, the code transmits an "I WON" message to the other micro:bit boards and displays the character *W* on the LED matrix. If the value of the incoming variable is not None, another player has won and his/her micro:bit has transmitted an "I WON" message, so the player loses and the code displays the character L on the LED matrix.

The full code of the final version of the game is presented next both in the text-based and blocks-based form:

Co-funded by the
Erasmus+ Programme
of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
from microbit import *
import radio
image = Image("00000:00000:00000:00000:00000")
def fill_pixel():
	x = 0
	y = 0
	while not button_a.was_pressed():
		if accelerometer.get_x() > 200 and x < 4:
			x += 1
		elif accelerometer.get_x() < -200 and x > 0:
			x -= 1
		if accelerometer.get_y() > 200 and y < 4:
			y += 1
		elif accelerometer.get_y() < -200 and y > 0:
			y -= 1
		display.show(image)
		display.set_pixel(x,y,int(9))
		sleep(200)
	image.set_pixel(x,y,int(6))
radio.on()
incoming = radio.receive()
while incoming == None and "0" in str(image):
	fill_pixel()
	incoming = radio.receive()
if incoming == None:
	radio.send("I WON")
	display.show('W')
else:
	display.show('L')
```

```
from microbit import *
import radio
image = ("0 0 0 0 0 :"
         "0 0 0 0 0 :"
         "0 0 0 0 0 :"
         "0 0 0 0 0 :"
         "0 0 0 0 0 ")
def fill_pixel ( ):
    x = 0
    y = 0
    while not button_a .was_pressed() :
        if accelerometer. get_x() > 200 and x < 4 :
            x += 1
        elif accelerometer. get_x() < -200 and x > 0 :
            x -= 1
        if accelerometer. get_y() > 200 and y < 4 :
            y += 1
        elif accelerometer. get_y() < -200 and y > 0 :
            y -= 1
        display.show( image )
        display.set_pixel( x , y , int( 9 ) )
        sleep( 200 )
    image.set_pixel ( x , y , int( 6 ) )
radio.on()
incoming = radio.receive()
while incoming == None and "0" in str(image) :
    fill_pixel ( )
    incoming = radio.receive()
if incoming == None :
    radio.send( "I WON" )
    display.show( 'W' )
else:
    display.show( 'L' )
```

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

3.5 Time for fun

Here are some ideas for extending the game developed so far:

1. To make the game more interesting, revise the code so that if the player selects a pixel that is already selected, it is deselected.
2. Another approach is to start with a certain image in the LED matrix that the player has to delete (make all pixels zero) in order to win. Start from the previous extension and develop this one.
3. As a final extension, you can start with an empty image, present a desired image to the player and then check when the player has created the desired image. In this case, the player has to remember the desired image in order to create it.

3.6 Self check

1. A logical expression may contain:
   a. Logical operators
   b. Comparison operators
   c. True or False statements
   d. All of the above
2. The logical expression A or B is True when:
   a. Both A and B are True
   b. A or B is True
   c. Both A and B are False
   d. It is always True
3. Select the correct operator to make the statement "George" ___ "Geography" True:
   a. >
   b. >=
   c. !=
   d. All of the above
4. When we call radio.receive() and nothing is sent, the return value:
   a. None
   b. Nothing
   c. ""
   d. Empty
5. The logical expression A and B is True when:
   a. Both A and B are True
   b. Either A or B are True
   c. Both A and B are False
   d. It is always True

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

6. Select the correct operator to make the statement 1 ___ 4 True:
    a. >
    b. ==
    c. <
    d. All of the above
7. An *if* statement can consist of:
    a. Only one *if* statement
    b. Multiple *elif* statements
    c. Only one *else* statement
    d. Exactly one *if* statement, none or many *elif* statements, and none or exactly one *else* statement
8. The *x* and *y* values of the accelerometer are for the:
    a. Pitch and Roll axis respectively
    b. Roll and Pitch axis respectively
    c. Yaw and pitch axis respectively
    d. Roll and Yaw axis respectively
9. The accelerometer can detect:
    a. The North Pole
    b. The South Pole
    c. The rotation of the micro:bit
    d. None of the above
10. Through the radio capability of the micro:bit we can only transmit:
    a. Strings
    b. Numbers
    c. Images
    d. Anything, if we have first converted it to a string

Co-funded by the
Erasmus+ Programme
of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

4. Battleship - Deepen Your Knowledge about Python and Micro:Bit

4.1 Aim

The aim of this learning activity is to go more in-depth in Python and learn some new useful coding techniques to make our code more elegant and effective. Our coding project will be a 2-player game, a simplified version of the famous Battleship game.

4.2 Synopsis

Battleship is a strategy-type guessing game for two players. It is played on ruled grids (paper or board) on which each player's ships are marked. The locations of each player’s ships are concealed from the other player. Players alternate turns calling "shots" at the other player's ships, and the objective of the game is to destroy the opposing player's fleet. An example of a game state is shown below where the position of ships is shown with grey rectangles and the shots of the other player are shown with X symbols. A grey rectangle with all its cells marked with Xs is essentially a sunk ship. A grey with some, but not all, rectangles with Xs is hit but not sunk yet. The game ends when all ships of a player are sunk.

	A	B	C	D	E	F	G	H	I	J
1										
2										
3										
4			X							
5						X	X			
6		X						X		X
7				X						X
8	X	X						X		
9										
10										

In our version of the game, the players will have 3 ships, 2 that are 2 cells long and 1 that is 3 cells long. The position of each ship is specified in the code. The player selects the position to shoot by tilting the micro:bit, which moves a red dot indicator, and then pressing the button A. To differentiate the multiple states of ships and shoots the corresponding pixels in the LED matrix appear in different color intensities. On the ships’ board, the ships are marked as bright red and the parts that are shot are marked as dim red. In the shot’s board, the hits are marked as bright red and the misses are marked as dim red.

4.3 Theory

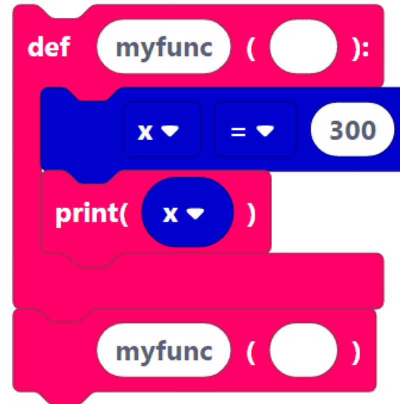
4.3.1 Global vs Local variables

In programming the variables are categorized as global or local depending on their scope. The scope refers to the visibility of a variable, meaning which parts of our program are aware of the variable and can use it. Usually, the variables we declare in our main program have a global scope. In the case that we declare a variable inside a function’s body, we cannot have access to that variable outside the function. Let’s take for example the following code snippet. You can try it in Python 3 mode of EduBlocks:

Co-funded by the Erasmus+ Programme of the European Union

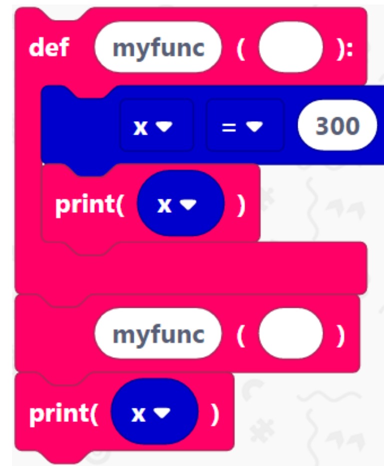
The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
def myfunc():
  x = 300
  print(x)

myfunc()
```



In this example we declare the *x* variable inside the function and we use it again inside the same function to print its value. Then we call the function and we see that it runs without any problems. Now let's try to print the value of *x* again after calling myfunc().

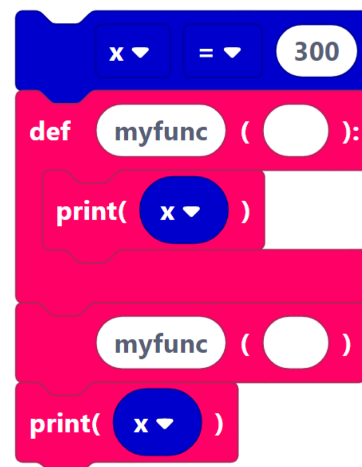
```
def myfunc():
  x = 300
  print(x)

myfunc()
print(x) #error
```



We see that if we try to run this code snippet, we get an error in the last line saying that *x* is not defined. This is due to the fact that *x* is declared inside the function's body and therefore its scope is local and cannot be accessed outside myfunc.

Let's see how we can use global variables. At the top of the program below, we declare the variable *x* and assign the value 300 to it. Now the scope of the variable is global and every part of our code has access to it.

```
x = 300

def myfunc():
  print(x)

myfunc()
print(x)
```

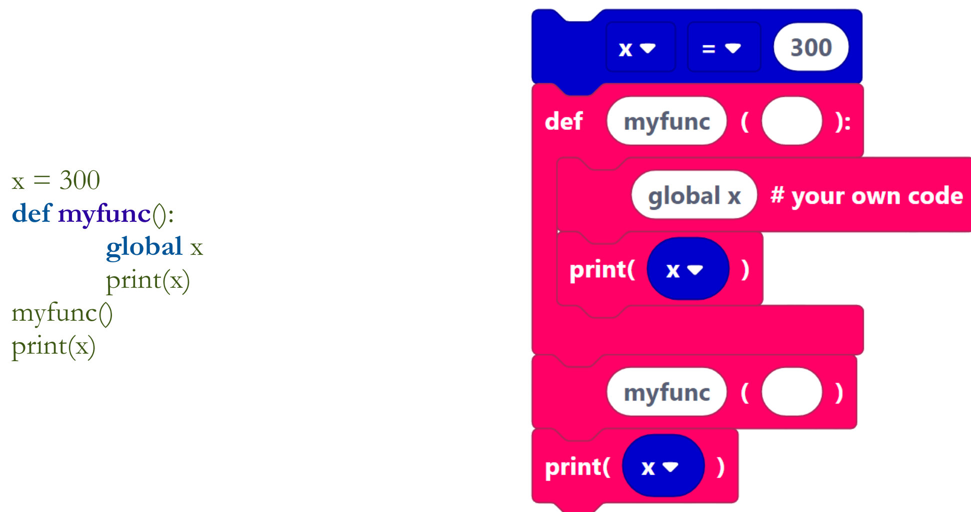


Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

If we run the above program (Python 3 mode of Edublocks) we see the value 300 printed two times in our console without any errors. We have to be very careful with global variables, especially in large programs since their value can easily be changed from anywhere on the code and we may find it difficult to troubleshoot our code in the case of unwanted behavior.

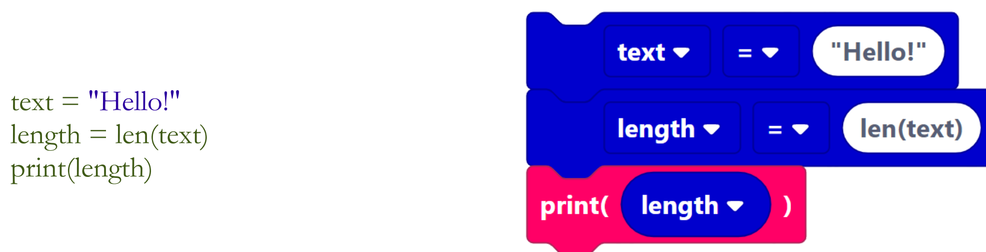
In MicroPython, which we use for our micro:bit projects, in order to access a global variable, we have to specify that it is global before we use it inside a function. In order for the above example to work in MicroPython, it has to be modified as shown below.



4.3.2 String manipulation

In programming there are some cases where we will have to know a string's length or even use only a part of it. To do that we do what's called *string manipulation*.

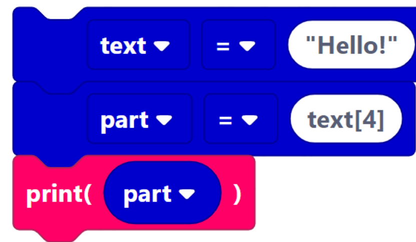
One important string manipulation operation is to find the length of a string *s*, meaning how many characters there are inside it. This is done using len(s). If we take for example the following code snippet (using Python 3 mode in EduBlocks), we get as output the value 6.



Another useful operation is to take a part of a string. To do so we perform the so-called *string slicing*. String slicing uses the indexes of the characters in a string. The first character of a string is in index 0, the second is in index 1, and so on. To take a character of a string *s* at index a we use s[a]. The snippet below shows how to do exactly that and the result is the letter "o" (using Python 3 mode in Edublocks).

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
text = "Hello!"
part = text[4]
print(part)
```



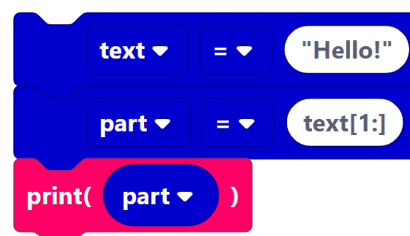
To take the part of a string *s* starting at index start and ending just before index stop, we use s[start:stop]. Run the following code using Python 3 mode in EduBlocks. What output do you see in the console?

```
text = "Hello!"
part = text[1:4]
print(part)
```

You should be able to see "ell" as output. The first thing to note here is that we start counting from 0, therefore in position 1 is the letter "e" and not "H". The number 4 at the end of the slicing statement represents the fifth character of "Hello", i.e. "o". However, this character is not part of the result, because in slicing the stop position of the slice is one less than the corresponding number. So, the result will consist of characters at indices 1, 2 and 3 (exclusive of 4), which is "ell".

In the case that we want to remove only the beginning of the string, we can omit the indices after the start index and replace them with the colon ":". This way the slice will start at the start index and continue until the end of the string. The following example will have as output the string "ello!" (using Python 3 mode in EduBlocks).

```
text = "Hello!"
part = text[1:]
print(part)
```

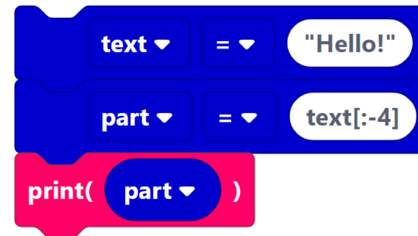


Another way to specify a slice is by omitting the start index and specifying the stop index. This way we can take a part of a string that starts from the first character up the stop index. We can also use negative indices to specify an index relative to the end of the string: -1 is the index of the last character of a string, -2 is one character before the last one, etc. The following example will have as output the string "He" (using Python 3 mode in Edublocks).

Co-funded by the
Erasmus+ Programme
of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
text = "Hello!"
part = text[:-4]
print(part)
```



All of the above string manipulation operations can be combined whenever necessary in order to get the desired output.

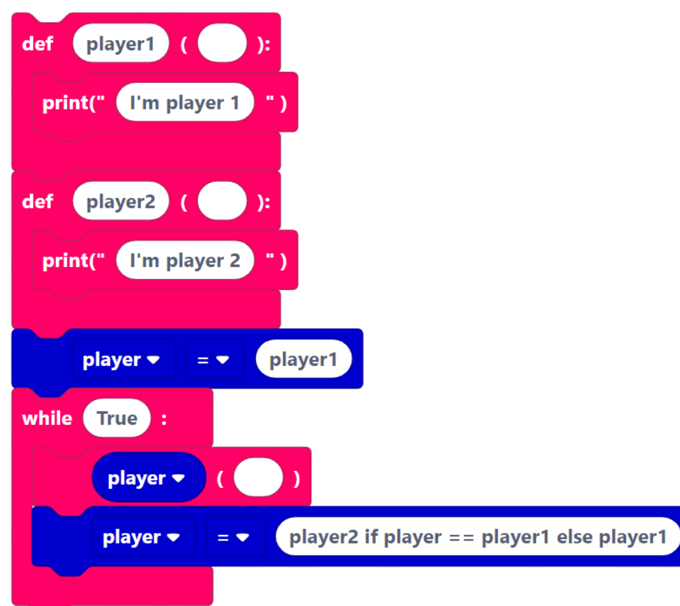
4.3.3 Objects

In Python, all functions, variables, numbers, and names are objects with their respective data attributes and methods. Specifically, objects are instances of classes. We can think of classes as prototypes for objects. They define a set of attributes that characterize any object of a certain class. The attributes can be data attributes or methods accessed via dot notation. In the following learning activities, we will see how we can create our own classes.

4.3.4 Functions as objects

Python functions can be used as objects. For example, if we have two functions player1() and player2() we can call them alternately as shown below. This code will print forever "I'm player 1" and "I'm player 2" alternately.

```
def player1():
        print("I'm player 1")

def player2():
        print("I'm player 2")

player = player1
while True:
  player()
  player = player2 if player == player1 else player1
```



The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

The last line of the above code uses the so-called ternary operator also known as conditional expression. This is an operator that evaluates something based on a condition being True or False. It simply allows testing a condition in a single line replacing the multiline *if else* statement making the code compact. Compare with the following code that is equivalent to this single line conditional expression:

```
if player == player1:
  player = player2
else:
  player = player1
```

4.4 Practice

To start coding our game we have to import the microbit and radio that provide all the functionality that we will need. We are now using the BBC micro:bit mode of EduBlocks as usual:

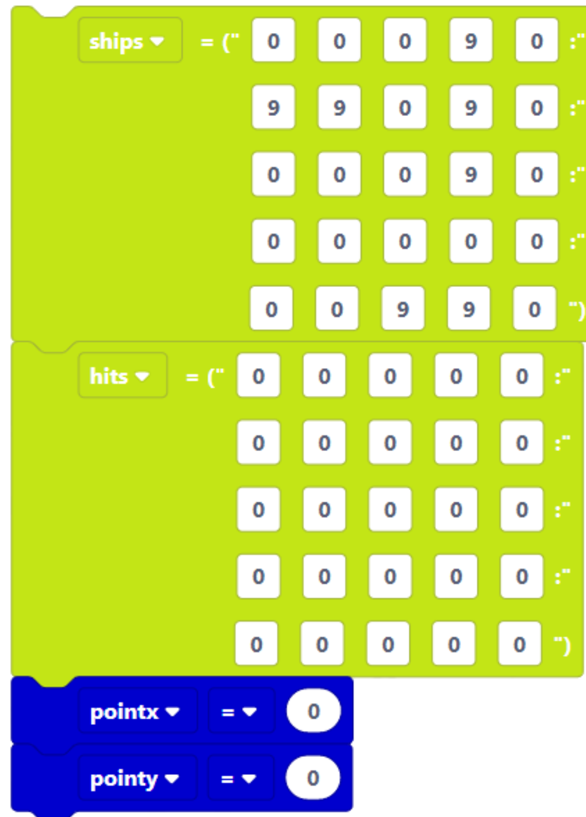
```
from microbit import *
import radio
```

Next, we must declare two Images, one that stores the placement of our ships and one that stores the shots we have done, with an indication for each shot if it was a hit or a miss. Here we also declare two variables that will store the *x* and y coordinates of the selected hit point in each game round. Note that the variable ships, which stores the placement of our ships, is initialized with a specific position of the three ships. To change this position, the corresponding statement should be changed.

```
ships = Image("00090:99090:00090:00000:00990")
hits = Image("00000:00000:00000:00000:00000")
pointx = 0
pointy = 0
```

Co-funded by the
Erasmus+ Programme
of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



The hit point selection function is identical to the one we used in the previous learning activity. In both cases, the micro:bit accelerometer is used to enable the player to select the hit point by tilting the micro:bit. An important difference is that the *x* and y coordinates are now global. Their names are pointx and pointy respectively. To be able to use them as global variables inside the hit point selection function, we have to use a **global** pointx, pointy statement inside this function. Yet another difference in our new hit point selection function is that when we press button A the function does not save our selection in the hits Image. It will save it after it is checked to find if it was a hit or a miss. The full code of the hit point selection function is as follows:

```
def select_hit_point():
        global pointx, pointy
        while not button_a.was_pressed():
                if accelerometer.get_x() > 300 and pointx < 4:
                        pointx += 1
                elif accelerometer.get_x() < -300 and pointx > 0:
                        pointx -= 1
                if accelerometer.get_y() > 300 and pointy < 4:
                        pointy += 1
                elif accelerometer.get_y() < -300 and pointy > 0:
                        pointy -= 1
                display.show(hits)
                display.set_pixel(pointx,pointy,int(9))
                sleep(200)
```

Co-funded by the
Erasmus+ Programme
of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
def select_hit_point ( ):
    global pointx, pointy # your own code
    while not button_ a .was_pressed() :
        if accelerometer. get_x() > 300 and pointx < 4 :
            pointx = pointx + 1
        elif accelerometer. get_x() < -300 and pointx > 0 :
            pointx = pointx - 1
        if accelerometer. get_y() > 300 and pointy < 4 :
            pointy = pointy + 1
        elif accelerometer. get_y() < -300 and pointy > 0 :
            pointy = pointy - 1
        display.show( hits )
        display.set_pixel( pointx , pointy , int( 9 ) )
        sleep( 200 )
```

This game has two modes that alternate between the players: the defender and the attacker. When one player is the attacker, the other has to be the defender, and vice versa. In order to better organize our code, we create two functions, one for each mode which we name defender() and attacker().

Let's begin with the attacker function. First, we specify that the variables *pointx* and *pointy* are global, the same way we did in the select_hit_point() function. After this statement, the code calls the select_hit_point() function in order for the attacker to select the next hit point.

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
def attacker():
    global pointx, pointy
    select_hit_point()
```

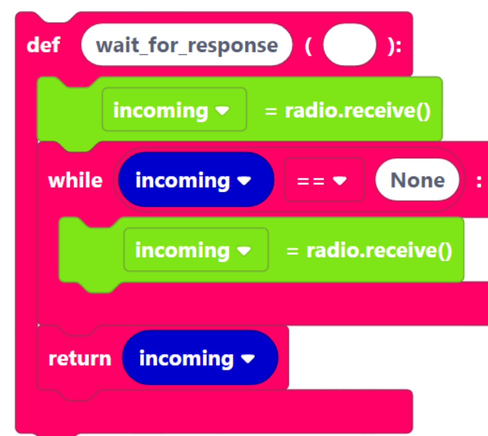
After the selection is complete, the *x* and y coordinates of the hit point are stored in the variables *pointx* and *pointy* respectively. Then, these coordinates are transmitted to the other player's micro:bit. To do so it is necessary to transform the arithmetic values of the coordinates into a string because radio.send() works only with string argument. To do this conversion we use the str() function for each one of the two coordinates and then combine the two using the + operator, which is the string concatenation operator. For example, if we had the coordinates 5 and 3 for pointx and pointy respectively we would compute str(5)+str(3) and the result would be the string "53". So, the statement that needs to be added in the attacker() function to send the message with the hit point to the defender is the following:

```
radio.send(str(pointx)+str(pointy))
```

The defender has to process the received hit point and send an answer to the attacker whether it was a hit or a miss. In the meantime, the attacker should wait for the response. To do so we will create a function that waits for a response via radio.receive() and returns the data received. This is the function wait_for_response() that we describe next.

As shown in the code below, in the wait_for_response() function we first save the data received from radio.receive() to a variable incoming and then we repeatedly check the contents of this variable using a *while* loop. If the variable has the value None it means we did not receive anything, so we read again the contents of radio.receive(), save them in the incoming variable and continue with the same loop. When the incoming variable's content is different from None, we know that we have received something and we return it.

```
def wait_for_response():
    incoming = radio.receive()
    while incoming == None:
        incoming = radio.receive()
    return incoming
```



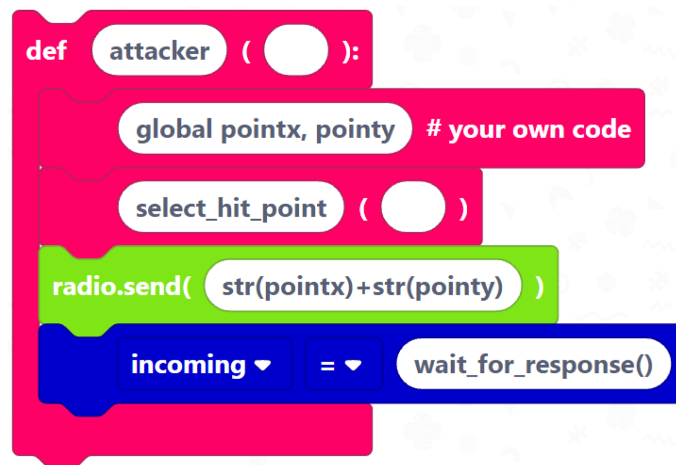
Using this new function, we can now continue with the attacker() function. The code we have developed so far enables the player to select the hit point, and send it to the other player's micro:bit. Then we can use the wait_for_response() function so that the attacker can receive the

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

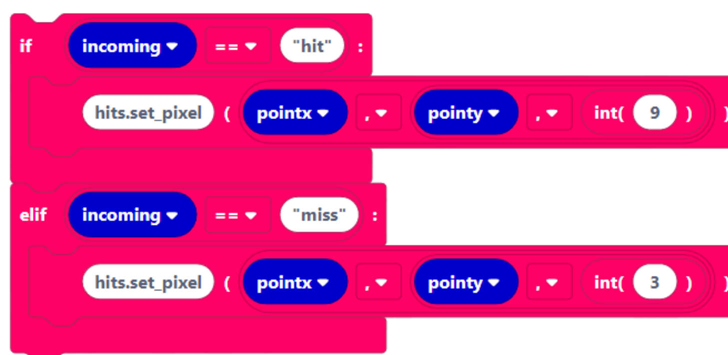
defender's response and learn if the hit was successful or not. This is the code of the attacker() function so far:

```
def attacker():
        global pointx, pointy
        selectHitPoint()
        radio.send(str(pointx)+str(pointy))
        incoming = wait_for_response()
```



Next, we need to check the contents of the incoming variable if it is "hit" or "miss". If it is "hit" we mark this position in the Image hits with bright red (intensity 9); otherwise, we mark a miss with low intensity (3). The code that accomplishes this check and further processing is the following:

```
if incoming == "hit":
        hits.set_pixel(pointx,pointy,int(9))
elif incoming == "miss":
        hits.set_pixel(pointx,pointy,int(3))
```



A last step is necessary in the attacker() function: to show the updated Image hits on the micro:bit's LED matrix. The complete code including this last step is given below:

```
def attacker():
        global pointx, pointy
        selectHitPoint()
        radio.send(str(pointx)+str(pointy))
        incoming = wait_for_response()
```

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
if incoming == "hit":
        hits.set_pixel(pointx,pointy,int(9))
elif incoming == "miss":
        hits.set_pixel(pointx,pointy,int(3))

display.show(hits)
```

The defender's function is complementary to the previous function to enable an appropriate interaction between the two players' micro:bits: When the defender's function is running on the micro:bit of one player, on the micro:bit of the other player the attacker's function is running and vice versa, until the game ends.

The defender() function begins by waiting for the selected hit point of the attacker, using the wait_for_response() function we created above. After we receive the hit point, we have to check if it points to a ship in the ships Image. Note here that the hit point is received as a string so we need to split the two coordinates and convert them to integers in order to use them for further processing. Further note that both coordinates are one digit long, so the message string will always be two characters long with the first character (at index 0) being the *x* coordinate and the second one (at index 1) being the *y* coordinate. So, the coordinates of the hit point will be int(incoming[0]) for *x* and int(incoming[1]) for *y*. Using these values, we get the appropriate pixel of the ships Image and check its intensity: if it is 9, we have a hit, otherwise we have a miss. For each outcome, we send the appropriate message to the attacker. Furthermore, in the case of a hit we have to change the intensity of the respective pixel in the ships Image in order to inform the player that a ship was hit. The last step in the defender's function is to display the new value of the ships Image on the LED matrix of the micro:bit. The final code that implements all this processing is as follows:

```
def defender():
```

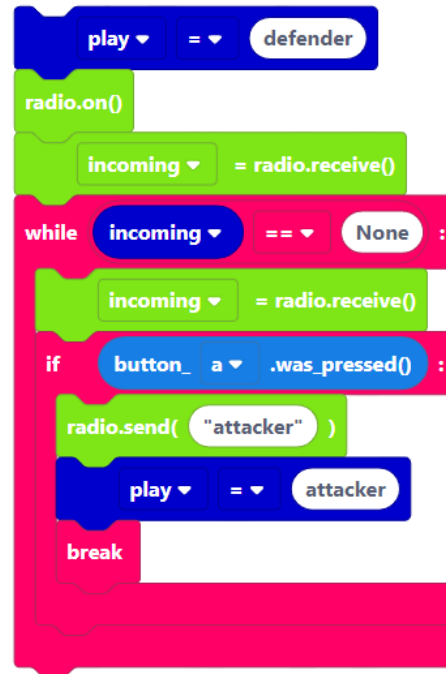
```
incoming = wait_for_response()
if ships.get_pixel(int(incoming[0]), int(incoming[1])) == 9:
        radio.send("hit")
        ships.set_pixel(int(incoming[0]), int(incoming[1]),3)
else:
        radio.send("miss")
display.show(ships)
```

At this point we have completed all the functions that we are going to use in our game. So, we can now go on with the main program. The first thing we have to do when the game starts is an initialization to determine who will be the attacker in the first round. To do so we create a variable named play in which we will save the function name which corresponds to the player's current role. We begin with both players being defenders and wait for one of them to press button A to become the attacker. As soon as one of the players becomes the attacker, an "attacker" message is sent to the other player's micro:bit. The reception of this message makes it possible to exit the *while* loop of the other player's micro:bit code and start operating in the defender's role. The complete code for this initialization process is shown below:

Co-funded by the
Erasmus+ Programme
of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
radio.on()
incoming = radio.receive()
while incoming == None:
    incoming = radio.receive()
    if button_a.was_pressed():
        radio.send("attacker")
        play = attacker
        break
```

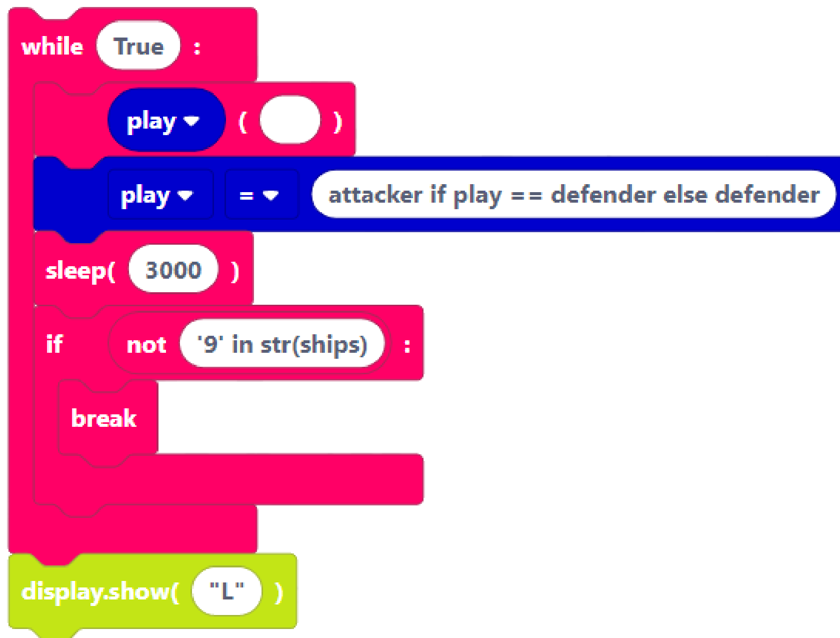


After the players have decided who the attacker is and who the defender is, the main game loop can start. This is a *while True* loop that alternates execution between the attacker() and the defender() functions for one player and between the defender() and the attacker() functions for the other player. In both cases, we call the play() function and then change the play variable to "attacker" if the previous value was "defender" and to "defender" if the previous value was "attacker". This is done using the techniques we have already seen in the theory of this learning activity.

Next, the code sleeps for 3 seconds in order for the players to read the Images that are shown on their screens, which are ships for the defender and hits for the attacker. The last thing is to check if there are any remaining ships in the ships Images. The ships are indicated by the intensity 9 so if there are no 9s in the Image ships then it means that all the ships of this player are destroyed and the game is over. In this case, we break the *while True* loop to stop the execution of our game and display the letter L on the screen to indicate that we lost. The final code is shown below and it is to be used just after the initialization code described above.

```
while True:
    play()
    play = attacker if play==defender else defender
    sleep(3000)
    if not '9' in str(ships):
        break

display.show("L")
```


The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



4.5 Time for fun

Here are some ideas for possible extensions of the game we have developed so far:

1. The game, as it is implemented, displays a message to the player that loses but does not display anything to the winner. Note that the micro:bit of the losing player will stop executing the *while True* loop. Note also that the losing player will always have the defender role, the attacker cannot lose because none of his/her ships can be hit before the defender takes the attacker role during the next game round. So, in the next round, if the defender of the previous round has lost, the defender of the new round will wait forever for the other player to send a hit selection point. Taking into account these facts, can you make a final extension to the game so that the winner can see an appropriate message in his/her micro:bit? You are free to change any portion of the code to achieve this.

2. Use random numbers to assign ship positions in the beginning. To do so, you have to use the statement import random in your code and then call random.randit(a,b) whenever you want to produce a random integer between a and b. Using this function, you can select a ship's position as follows: first, select randomly one of the five rows or one of the five columns of the 5X5 board. Then position the ship randomly in the selected row or column: if you want to place a 3-point ship, select its start position to be between 1 to 3, for a 2-point ship select its start position between 1 and 4. Check if the ship overlaps with an already placed ship and, if yes, repeat the process until there is no conflict.

4.6 Self check

1. In Python, if we declare a variable in the global scope, we can access it:
    a. From anywhere
    b. From the main program
    c. From functions only
    d. From functions that are defined after the declaration of the variable
2. In MicroPython in order to access a global variable from a function, we have to:
    a. Do nothing

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

b. Declare it again in our function

c. Use the keyword *global* followed by the name of the variable before we access it

d. Use the keywords *global variable* followed by the name of the variable before we access it

3. In Python, if we declare a variable inside a function, we can access it:

a. From anywhere

b. From the main program

c. From inside the function

d. From inside the function, and all functions defined after it

4. In order to find the number of characters (length) of a string saved in a variable named s we use:

a. size(s)

b. len(s)

c. characters(s)

d. print(s)

5. In order to take only the last word of the string "Hello World" which is saved in a variable named s we use:

a. s[6:]

b. s[6:-1]

c. s[7:11]

d. s[2]

6. Objects are:

a. Strings

b. Variables

c. Numbers

d. All of the above

7. Classes are:

a. Prototypes that define objects of the same type

b. Groups of objects

c. Both a. and b.

d. None of the above

8. A distinct entity of the real world can be represented by:

a. A class

b. An object

c. A method

d. All of the above

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

9. Function names:
    a. Can start with a "_" character (underscore)
    b. Can be used as objects
    c. Can be assigned to variables
    d. All of the above
10. What is the slicing operation that will return a copy of a string s without its last character?
    a. s[0:-1]
    b. s[:len(s)-1]
    c. s[:-1]
    d. all of the above

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

5. A Quiz Game - Advanced Topics in Python with Micro:Bit

5.1 Aim

The aim of this activity is to get more in-depth knowledge of Python and learn about lists, dictionaries and the speech library of micro:bit. We are going to use only the text-based mode of EduBlocks. We do so because the code in this learning activity is rather difficult to develop using block-based representation. Furthermore, as we noted in the introduction, the ultimate goal of using the dual code representation of EduBlocks is to gradually get used to text-based coding which is the format used by experienced programmers. Our coding project will be a simple quiz game that can be very easily extended with new questions.

5.2 Synopsis

In the first form of the quiz game a random question is selected from a predefined list of questions and an appropriate oral message is played by micro:bit using the speaker. After that, the player has to decide if the statement was True or False by pressing the button A or B respectively. If the player's answer is correct a happy face is shown on the display, otherwise, a sad one is displayed. The player earns one point for each correct answer and loses one for each wrong answer. After all the questions have been asked, the score of the player is shown on the display.

In the second form of the quiz game, instead of using true-false questions, we use multiple-choice questions. The player can go through the possible choices by pressing the button A repeatedly, and by pressing the button B he/she can select the current choice as his/her answer. The player can listen to the choices as many times as it is needed. For every correct answer a happy face is shown on the display and the player earns one point. For every wrong answer a sad face is shown on the display and the player loses one point. After all the questions have been announced, the score of the player is shown on the display.

5.3 Theory

5.3.1 Lists and tuples

In Python lists can be used to manage collections of values that are stored as a sequence in a single variable. We create a list using the brackets [ ], and between them we place as many items as we want separated with commas. The items can be of any type, even other lists. For example, we can create a list of strings containing fruit names as shown below.

```
fruits = ["apple", "banana", "cherry"]
```

We can print the entire list using print(fruits) and we can access each item individually by its index, beginning from 0, just like the characters of a string. For example, to print the second item in the above list we use print(fruits[1]) and the output will be "banana".

In the case that we want to add yet another fruit name ("orange") at the end of the list, we use the statement fruits.append("orange"). To change an item that is already present in the list, e.g., the third item from "cherry" to "strawberry", we use the statement fruits[2] = "strawberry". If we run both of these commands the new list will be: ["apple", "banana", "strawberry", "orange"]. Finally, to remove list items we use the statement fruits.remove("banana"). This last statement will change the list to: ["apple", "strawberry", "orange"].

Elements in a list need not be necessarily unique. Repetitions are allowed, e.g., in the list of the number of days per month in a typical year starting from January: days = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31].

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

The function len() can be used to find the length of the list, that is, how many elements it contains. For example, len([2, 1, 9]) will return 3.

Negative indices can be used to access elements in the tail of a list. The last element can be accessed with index -1, the element before that with index -2, etc. For example, the following code prints the number of days in November: print("November has ", days[-2], " days.")

Finally, it is possible to specify slices of a list using a starting index, an ending index, and an optional step. For example, the days of months in summer are days[5:8] while the days of months that start each season (March, June, September, and December) are days[2::3]. It is interesting to note that slicing indices can be omitted as it is the case for one of the ending indices in this last example. If the starting index is omitted, the slice starts from the beginning of the list. If the ending index is omitted, the slice ends at the end of the list. If the step is omitted, +1 is assumed. Consequently, the notation days[::] will produce a full copy of the original list because all three slicing indices will use the default values.

Lists can be added and multiplied as well! Adding two lists with the '+' operator produces a new list that is the concatenation of two lists. For example, the expression: [0,1]+[2,3] will evaluate to [0,1,2,3]. Multiplication between a list l and an integer k produces a new list with k copies of l. For example [0]*3 will produce [0,0,0] while [0,1]*2 will produce [0,1,0,1].

To delete or remove an element from a list, there are two alternatives. For example, either del days[2] or days.remove(28) will remove the value 28 at the second position of the days list. Note that the first alternative deletes an element at a certain position while the second alternative will remove a certain value from a list. If the value appears more than once, only the first occurrence will be removed.

Apart from lists, Python also supports tuples. Tuples are very similar to lists. They are sequences of elements. They can be handled using the same operators and methods. The only difference between tuples and lists is that they cannot be changed. To differentiate their representation tuples, use parentheses, whereas lists use square brackets. For example, if we specify numbers = (2, 5, 9) it will be a tuple that could not be changed by operations such as the ones described in the previous paragraph. Whenever you want your sequence of elements to be fixed through the rest of your code, use tuples instead of lists.

5.3.2 Dictionaries

Python dictionaries are used to store data values in *key:value* pairs. For example, if we want to store information about a car, we could use a dictionary like this:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

As another example, we present a dictionary to associate months with the number of days they have in a typical year:

```
days = { "January" : 31, "February": 28, "March" : 31, "April" : 30, "May" : 31, "June" : 30, "July"
: 31, "August" : 31, "September" : 30, "October" : 31, "November" : 30, "December" : 31}
```

To access the elements in a dictionary, the brackets notation is used as in lists and tuples. However, inside the brackets, keys are used and not numbered indices as in the case of lists and tuples. For example, the following statements can be used to retrieve or change the number of days in certain months:

Co-funded by the
Erasmus+ Programme
of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
print("January has ",days["January"]," days.")
days["February"]=29
```

Consequently, there are several alternative ways to define a dictionary:

- By providing all *key:value* pairs inside brackets:

  ```
  d = {'foo': 100, 'bar': 200, 'baz': 300}
  ```

- By defining one by one its *key:value* pairs:

  ```
  d = {}
  d['foo'] = 100
  d['bar'] = 200
  d['baz'] = 300
  ```

- By creating a *dict* object with appropriate initialization as a comma separated list of key=value constructs:

  ```
  d = dict(foo=100, bar=200, baz=300)
  ```

- By creating a *dict* object with appropriate initialization as a list of *(key,value)* tuples

  ```
  d = dict([
      ('foo', 100),
      ('bar', 200),
      ('baz', 300)
  ])
  ```

Removing an item from a dictionary can be done using pop. For example, to remove the item that corresponds to the "year" key of the car dictionary, we use the statement car.pop("year").

5.3.3 For - enumerate

In order to iterate over the elements of a list or the characters of a string we can use a *for* loop. At each iteration, the *for* loop will take a single item of the list or a single character of the string and do something with it before moving to the next one. The iterations will continue until it reaches the end of the list or string. For example, if we want to iterate over a list of fruits and print each item, we would do it like this:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
        print(fruit)
```

The result of executing the above code would be to print each item on the list on a separate line.

In the same way, we would print each character of a string, but what if we would like to show the number of the letter next to it? Then we should use the function enumerate() with our for loop as shown below.

```
text = "Hello!"                                  #Result
for index, character in enumerate(text):         0 H
        print(index, character)                  1 e
                                                 2 l
                                                 3 l
                                                 4 o
                                                 5 !
```

What this does is that in every iteration it assigns a pair of the current character and its index to the variables character and index respectively. In general, the enumerate function produces a

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

number each time we iterate in the *for* loop, starting from zero and incrementing it by one in each iteration. It can be used with lists as well.

5.3.4 Random numbers

Sometimes in our code it is necessary to be able to produce random numbers. This is very useful in games whenever we want our game to operate in a less predictable way so that it is more interesting for the player. To be able to produce random numbers, we should import the random library first. To produce a random integer between numbers start and stop, we can use random.randint(start, stop). For example, the following code will print a random number from 1 through 10:

```
import random

number = random.randint(1, 10)
print(number)
```

If we want to get a random floating-point number between 1 and 10, we should use random.uniform(1, 10).

Another very interesting operation is to choose a random item from a list. To do so, we use the random.choice() and we give the list's name as an argument. The following code randomly selects a fruit name from a list and prints it:

```
import random

fruits = ["apple", "banana", "cherry"]
random_fruit = random.choice(fruits)
print(random_fruit)
```

5.3.5 *Try-except*

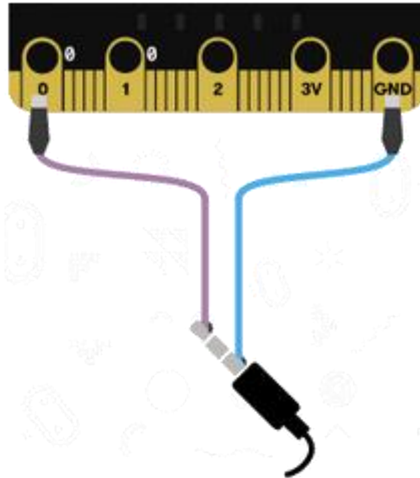
Sometimes in our code there are parts where we expect an error to occur under certain conditions. To deal with such errors and prevent our code from crashing (terminating unexpectedly) we use a *try-except* block. We first specify what we want to do under the *try* statement and under the *except* statement we specify what we want to do if the *try* block fails, i.e., if an error happens (or, to be more precise, an exception). Such an exception could happen if we try to access a non-existing list item. For example, the following code will produce an exception as soon the fruit list remains empty (elements are removed one by one). At that point, the *except:* block will be executed and the while True block will be terminated (the break statement will be executed).

```
import random

fruits = ["apple", "banana", "cherry"]
while True:
  try:
    random_fruit = random.choice(fruits)
  except:

    print("No item to print")
    break
  fruits.remove(random_fruit)
  print(random_fruit)
```


The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

5.3.5 Connecting an external speaker with micro:bit

In the learning activity we are going to use the sound feature of micro:bit. In the case of micro:bit V2, there is a built-in speaker that will output the sounds we want. In the case of version 1 micro:bit we need to connect an external speaker or headphones. This is done very easily, as shown in the picture below, by connecting Pin0 to the tip and GND to the sleeve of the jack of our headphones. If we want to connect an external speaker, we connect Pin0 to the red lead and GND to the black lead of the speaker.



5.4 Practice

5.4.1 True-False quiz game

We begin our code by importing the libraries microbit, speech, and random that we will use in our code.

```
from microbit import *
import speech
import random
```

Next, we have to create a list which will hold the questions of the quiz along with their correct answers. The questions will be in the form of dictionaries that have two *key:value* pairs, the first pair storing the question text and the second the correct answer (either True, or False). You can create more questions by adding more dictionaries to the questions list. Here is a possible initialization:

```
questions = [
    {"question": "-The capital of Greece is, Athens.", "correct": True},
    {"question": "-The capital of Germany is, London.", "correct": False},
    {"question": "-The capital of Italy is, Rome.", "correct": True},
    {"question": "-The capital of Poland is, Moscow.", "correct": False},
    {"question": "-The capital of Germany is, Berlin.", "correct": True},
    {"question": "-The capital of Russia is, Moscow.", "correct": True},
]
```

Now that we know the form of our questions, we can proceed to create some useful functions. First, we create a function named announce_question() which announces the question given to it as an argument. To do so we use the speech.say() function from the speech library which takes a string as an argument. To access the string of the question that was given as an argument to the function we use question["question"].

```
def announce_question(question):
```


The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
    speech.say(question["question"])
```

After that we can create a function named process() which will be in charge of checking if the answer of the player was true, showing a happy or sad face on the screen depending on the outcome, and returning the points to add or remove from the score. The arguments of this function are the current question and the answer of the player (True or False). To access the correct answer for the question we use question["correct"]. In the case of a correct answer, we show a happy face and return 1, otherwise, we show a sad face and return -1.

```
def process(question, option):
    if question["correct"] == option:
        display.show(Image.HAPPY)
        return 1
    display.show(Image.SAD)
    return -1
```

We can now create the main loop of the game. We first initialize the score to 0 and create a while True loop. Inside we first choose a question from the questions list using the random.choice(questions), which returns one random question. Then we remove the chosen question from the list so it does not play again, and call the announce_question() function we created giving the chosen question as an argument to be announced.

```
score = 0
while True:
    question = random.choice(questions)
    questions.remove(question)
    announce_question(question)
```

We should now wait for the player's response using a while True loop. Inside the loop, we repeatedly check whether button A or B was pressed. We should remember that button A is assigned to the True answer and B to the False answer. In the case that the player presses the button A we call the function process we created with the current question and the value True as arguments. The result of the process() function (+1 or -1) is added to the score. In the case that the player presses button B we do exactly the same but this time we give False instead of True as an argument. In both cases, we break the while True loop that we are in, in order for the game to continue to the next question.

```
    while True:
        if button_a.was_pressed():
            score += process(question, True)
            break
        elif button_b.was_pressed():
            score += process(question, False)
            break
```

The last thing to do is to check if there are any questions left in the list. To do that, we use len(questions) which returns the number of questions in the list. If the list has 0 questions that means that the game is finished and so we break the main loop of the game. After that we show the score of the player to the display.

```
score = 0
while True:
    question = random.choice(questions)
```


The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
    questions.remove(question)
    announce_question(question)
    while True:
        if button_a.was_pressed():
            score += process(question, True)
            break
        elif button_b.was_pressed():
            score += process(question, False)
            break
    if len(questions) == 0:
        break
display.show(score)
```

The complete code of the true-false quiz game is given below:

```
from microbit import *
import speech
import random


questions = [
    {"question": "-The capital of Greece is, Athens.",

        "correct": True},
    {"question": "-The capital of Germany is, London.",

        "correct": False},
    {"question": "-The capital of Italy is, Rome.",

        "correct": True},
    {"question": "-The capital of Poland is, Moscow.",

        "correct": False},
    {"question": "-The capital of Germany is, Berlin.",

        "correct": True},
    {"question": "-The capital of Russia is, Moscow.",

        "correct": True},
]


def announce_question(question):
    speech.say(question["question"])


def process(question, option):
    if question["correct"] == option:
        display.show(Image.HAPPY)
        return 1
    display.show(Image.SAD)
    return -1


score = 0


while True:
```


The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
        question = random.choice(questions)
        questions.remove(question)
        announce_question(question)

        while True:
            if button_a.was_pressed():
                score += process(question, True)
                break
            elif button_b.was_pressed():
                score += process(question, False)
                break
        if len(questions) == 0:
            break


    display.show(score)
```

5.4.2 Multiple choice quiz game

As an extension of the quiz game, we can create one that will use multiple-choice questions instead of True-False questions. The same libraries as before are imported and we create a list named questions. The form of the questions in this new quiz game will be dictionaries with three *key:value* pairs. The first one is the text of the question as it was in the True-False quiz game. The second *key:value* pair provides a list of possible choices for the answer to the question. The last *key:value* pair provides the index of the correct answer in the previous list of choices. Remember that the indexes of a list begin from 0, not 1, which means that for example, the third choice has the index 2. The code below implements all these ideas and can be extended with more multiple-choice questions.

```
from microbit import *
import speech
import random

questions = [
    {
        "question": "-The capital of Greece is,",
        "options": ["-Rome", "-Athens", "-Berlin", "-Paris"],
        "correct": 1,
    },
    {
        "question": "-The capital of Germany is,",
        "options": ["-Rome", "-Athens", "-Berlin", "-Paris"],
        "correct": 2,
    }
]
```

Next, we will develop a function that will use the speech.say() function to give a question to the player. The function will also announce the choices that the player has. This is done with a while True loop which repeatedly cycles through the options when the player presses button A and stops when button B is pressed. To do that, we first need a *for* loop to iterate over the list of options, which not only gives us the options but also its index in the list. We need the index in order to give it as an argument to the process function which checks if the option we selected is the correct one. To get the indexes of the options we use the enumerate(question["options"]) in the *for* loop.

```
def make_question(question):
    speech.say(question["question"])
    sleep(1000)

    while True:
        for i, opt in enumerate(question["options"]):

            speech.say(opt)
            while True:
                if button_a.was_pressed():
                    break
                elif button_b.was_pressed():
                    return process(question, i)
```

Note that inside the *for* loop in the code we announce the current option and enter a while True loop which waits for an input from the player. In the case that button A is pressed we break the loop and let the *for* loop iterate to the next option. If the for reaches the end of the options list, it exits, but because we are also in a while True loop it starts again from the beginning. If button B is pressed, we call the process() function with the current question and the current option's index as arguments to check if the player's selection was correct. The result of the process function is then returned, which exits all the loops we are in.

The process function that we have created in the first version of the game remains unchanged:

```
def process(question, option):
    if question["correct"] == option:
        display.show(Image.HAPPY)
        return 1
    display.show(Image.SAD)
    return -1
```

We can now set the score to 0 and enter our main game *while True* loop. Here, in order to stop the execution of the game when the questions list is empty, instead of checking the length of the list we use a *try-except* statement. First, we try to get a random choice from the list and if we succeed, the game continues. In the case that the list is empty the random choice function will fail and throw an *IndexError* exception, which we then handle by breaking the main loop of the game and thus ending the game.

After we make a random selection from the list of questions, we remove that question from the list so it would not be used again. Then we add to the score the outcome of the make_question() function to which we give the chosen question as an argument. Finally, we display the score of the player.

```
score = 0
while True:
    try:
        question = random.choice(questions)
    except IndexError:
        break
    questions.remove(question)
    score += make_question(question)

display.show(score)
```

The completed code of the multiple-choice quiz game is given below:

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
from microbit import *
import speech
import random

questions = [
    {
        "question": "-The capital of Greece is,",
        "options": ["-Rome", "-Athens", "-Berlin", "-Paris"],
        "correct": 1,
    },
    {
        "question": "-The capital of Germany is,",
        "options": ["-Rome", "-Athens", "-Berlin", "-Paris"],
        "correct": 2,
    }
]


def make_question(question):
    speech.say(question["question"])
    sleep(1000)

    while True:
        for i, opt in enumerate(question["options"]):

            speech.say(opt)
            while True:
                if button_a.was_pressed():
                    break
                elif button_b.was_pressed():
                    return process(question, i)


def process(question, option):
    if question["correct"] == option:
        display.show(Image.HAPPY)
        return 1
    display.show(Image.SAD)
    return -1


score = 0


while True:
    try:
        question = random.choice(questions)
    except IndexError:
        break
    questions.remove(question)
    score += make_question(question)


display.show(score)
```


The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

5.5 Time for fun

Here are some ideas for extending the quiz games:

1. Modify the true-false game to give a certain amount of time to the player to answer each question. If no answer is given within the time limit, the player loses the question. Another option would be to offer to the player a total time limit for the whole game. The player continues answering questions till the time limit is reached.

2. Modify the multiple-choice game so that it does not stop after the presentation of the questions but continues with all the questions that were not answered correctly by the player. This is repeated until the player answers all the questions correctly. The score is computed the same way: +1 point for each correct answer and -1 point for each wrong answer.

3. Modify the multiple-choice game so that the questions are classified in three categories: easy, medium, and difficult. Each category corresponds to different points: +1/-1 for easy questions, +2/-2 for medium questions, and +3/-3 for difficult questions. Revise as required the code to handle the player's score following these new rules. Present the letters E, M, and D in the LED matrix when an easy, medium, or difficult question is announced respectively.

5.6 Self check

1. Lists can be used to:
   a. Store multiple items in one variable
   b. Store any sequence of values of a certain data type
   c. Store any sequence of values including values of different data types
   d. All of the above
2. To create a list, we put all the items between:
   a. Parentheses ( )
   b. Brackets [ ]
   c. Curly brackets { }
   d. Quotes " "
3. We separate the items of a list by:
   a. Commas ,
   b. Dots .
   c. Dashes -
   d. Spaces
4. To create a dictionary, we put all items (*key:value* pairs) inside:
   a. Parentheses ( )
   b. Brackets [ ]
   c. Curly brackets { }
   d. Quotes ""
5. We separate a dictionary's pairs by:
   a. Commas ,

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

b. Dots .

c. Dashes -

d. Spaces

6. We separate the key and value of a pair in a dictionary by a:

a. Semicolon ;

b. Dot .

c. Dash -

d. Colon :

7. If we want to print every item of a list or a dictionary by iteration we use:

a. The print() function

b. A *for* loop

c. An appropriate print() function inside a *for* loop

d. None of the above

8. If we want to print every item of a list enumerated we use:

a. The print() function

b. A *for* loop

c. A *print-enumerate* loop

d. A *for* loop with enumerate and a print() function inside it

9. To choose a random item from a list named L we use:

a. random(L)

b. L.random_choice()

c. L.random.choice()

d. random.choice(L)

10. When we expect a part of our code to fail, in order to protect our program from crashing, we place it inside a:

a. *try* block

b. *except* block

c. *if* statement

d. *try-except* block

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

6. Introduction to Raspberry Pi Pico

6.1 Aim

The aim of this activity is to learn about another microcontroller that can be programmed with MicroPython, namely Raspberry Pi Pico. It is more powerful compared to the micro:bit and has a lot more memory in order to support bigger and more complex programs.

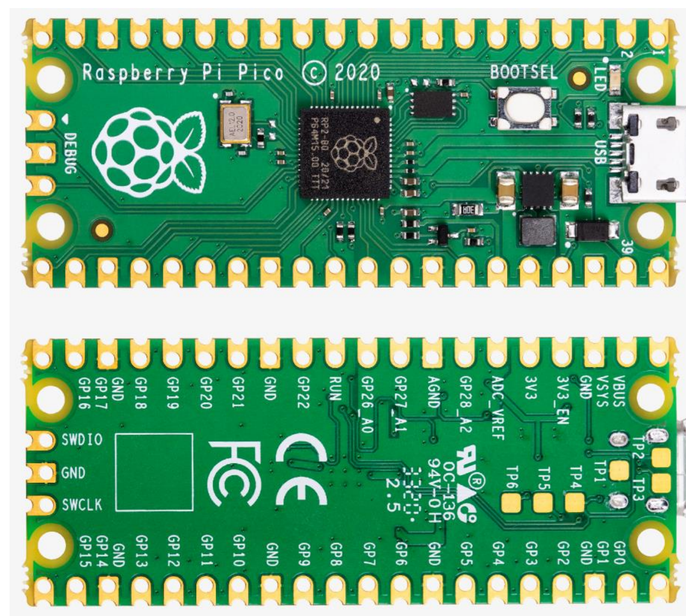
6.2 Synopsis

In this activity we will create a single-player game. The game will consist of an array of five LEDs (1 red & 4 blue) which light up one after the other. The goal is to pause the blinking of the LEDs when the red LED is on, using a button as user input.

6.3 Theory

6.3.1 What is the Raspberry Pi Pico

The Raspberry Pi Pico is a microcontroller board, similar to the micro:bit, but more powerful. It has a lot more GPIO pins while also supporting more communication protocols. It can be programmed with various languages including MicroPython which we have also used for programming the micro:bit. The following picture presents the Raspberry Pi Pico board.



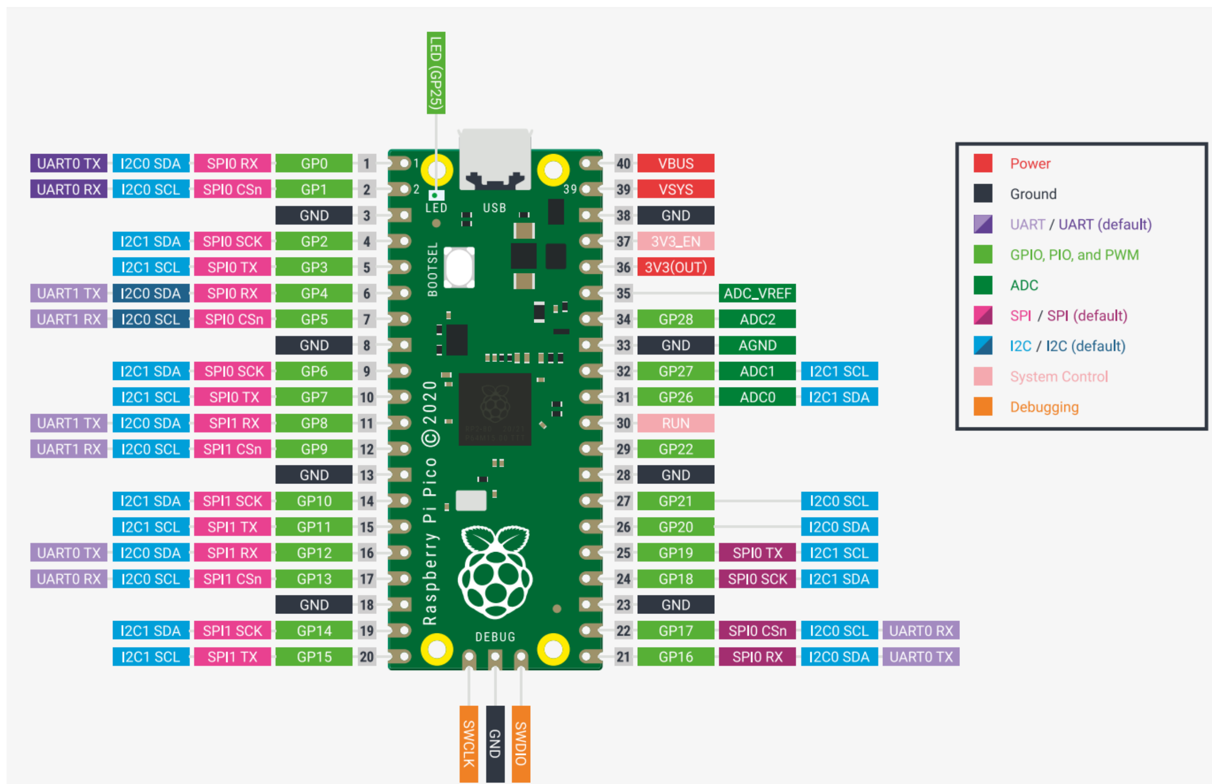
To set up the Raspberry Pi Pico with MicroPython you have to follow the instructions that are given through the official site of Raspberry Pi. They can be found here: [Getting started with Raspberry Pi Pico - Add the MicroPython firmware | Raspberry Pi Projects]

6.3.2 Pinout

The back side of the Raspberry Pi Pico shows us the position and the number of each GPIO, power, ground and ADC pin. The secondary functions of the GPIO pins are shown in the following pinout description.

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



All of the 26 GPIO pins are PWM capable but only up to 16 can be used simultaneously.

6.3.3 Micro:bit vs Raspberry Pi Pico

The following list compares the BBC micro:bit V2 to the Raspberry Pi Pico. We observe that the micro:bit has more ADC capable pins, more built-in sensors, and a screen with LEDs on-board but the Pico has more GPIO overall, more communication ports, and a lot more memory.

S.no	BBC micro:bit	Raspberry Pi Pico
1	Nordic nRF52833 32-bit ARM® Cortex™ M0 CPU	RP2040 microcontroller with 2MByte Flash
2	5V DC via USB connector	5V DC via USB connector
3	20 GPIO in which 5 can be used for ADC	26 GPIO (3.3v rated), 23 are digital-only and 3 are ADC capable
4	1x UART, 1x I2C, 1x SPI	2x UART, 2x I2C, 2x SPI, up to 16 PWM channels
5	NA	On-board USB1.1 support
6	512KB Flash	2MByte QSPI Flash memory
7	128KB RAM	264K multi-bank high-performance SRAM
8	32-bit ARM Cortex M0 CPU at up to 64 MHZ	Dual-core Cortex M0+ at up to 133MHzş
9	25 Programmable led's in a 5x5 grid	LED_BUILTIN - GP25
10	Inbuilt temperature sensor	Inbuilt temperature Sensor
11	NA	3-pin ARM Serial Wire Debug (SWD) port
12	Accelerometer	NA
13	Light Sensor	NA
14	Compass	NA
15	Touch Logo	NA
16	52mm x 42 mm	51mm x 21mm x 1mm

6.3.4 Classes

A Python class is a prototype for an object that defines a set of attributes that characterize any object of the class. The attributes can be either data attributes or methods, accessed via dot notation.

For example, let's say that we want to create a program that manages people. We can create a class named Person with an initializer that assigns values to the instance variables of the person. The initializer has the name __init__ and takes as arguments the desired values for the instance variables of the object we want to create. When we declare an initializer, we must always include the self object in its arguments and use it with dot notation to access the attributes of the newly created object. An example class is shown below:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
```

The way that we assign the given values to the properties of self is as shown above. Now, in order to create objects of this class, we must use the so-called class instantiation. Class instantiation uses function notation. Just pretend that the class object is a function that returns a new instance of the class. The arguments that you should use are the ones that have been defined in the class initializer just after self. The following code creates two instances of the Person class:

```
p1 = Person("John", 22)
```

```
p2 = Person("George", 34)
```

An example of a method for the Person class could be to introduce oneself, which is a very common action for a person. Below is shown how we can create the method, create an object and then call its method to introduce itself.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def introduce(self):
    print("Hello, my name is " + self.name)

p1 = Person("John", 22)
print("Hello, my name is " + p1.name)
p1.introduce()
```

The output of the above code is "Hello, my name is John" twice. You can see that in order to access the data that we saved to the object we use self.name inside the class and p1.name outside the class.

Note that not all classes require an initializer, especially those with no member (instance) variables.

6.3.5 Electronic components

In this activity we are going to use some electronic components such as LEDs, buttons, potentiometers, and servo motors. So, we need to get familiar with these components first.

LEDs

Light Emitting Diodes or LEDs are semiconductors that emit light when they are connected properly to a power source. An LED consists of two lead connectors, one of which is smaller than the other one. The longer lead is the anode (+) of the LED and the short one is the cathode (-). The LED will light up only when we connect it the right way: The anode to positive voltage and the cathode to the ground. LEDs usually work fine with 3.3 Volts but, when we work with more Volts than that, we must always use resistors to protect them from overcurrent. For example, if we use 5 Volts and a common 5mm LED, we can use a 330 Ohm resistor to be safe.

Buttons

Buttons are like switches and can be used to control the flow of electricity. When we intersect a button on a line, there is no flow of electricity until we press it. In our activity, a button will be used as user input.

Buttons suffer from an effect called *switch bouncing*, which for fast microcontrollers means that when we press or release them, they oscillate from close to open a few times. To solve this, we connect a capacitor in parallel with the button.

When we do not press the button, the pin that it is connected to is left hanging, meaning that the Pico cannot decide if it is in a HIGH state or LOW state. To solve this, we enable a pull-down resistor on that pin in order to keep the pin's state to LOW until we press the button.

Potentiometers

Potentiometers are essentially variable resistors. There is a knob that can be rotated in order to select the desired resistance. They are labeled with their maximum resistance. In our activity, we are going to use them for selecting the speed of the LEDs flashing. We connect GND and 3.3VOUT of the Raspberry Pi Pico to the edge pins and we measure the resistance using the middle pin attached to an ADC capable pin of the Raspberry Pi Pico.

Servo Motors

Servos are PWM-controlled motors that move relatively slowly and can rotate their axle to the desired position extremely accurately. Their range of motion is usually 180 degrees.

Pulse Width Modulation (PWM) is a way to control the pulse width (on time) of a signal. For example, for a PWM signal that is on for half the time and off for the rest, we say that it has a duty cycle of 50%. If the signal's frequency is 50Hz (20ms period), this results in a 10ms pulse width. The most common servos (SG90) operate between 1ms and 2ms, with 1ms being the 0-degree position and 2ms being the 180-degree position. To achieve this range of pulse widths, we have to generate duty cycles from 5% to 10%. Because the SG90 servos are usually cheaply made, in order to compensate for small variances between them we adjust the range to 2% for 0 degrees and 12.5% for 180 degrees.

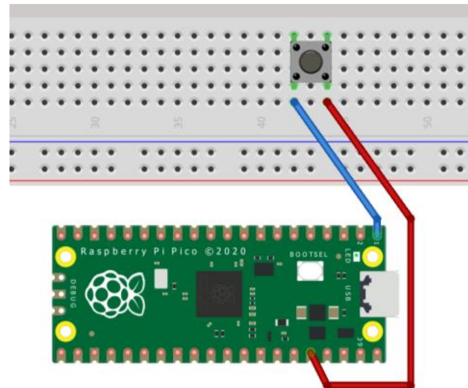
6.3.6 Interrupts

In computers and microcontrollers, when we want to wait for an event to happen, e.g., a button press, we can set up an interrupt. The first thing we could do in order to detect a button press is to continuously check the pin of the button to see if it is currently being pressed. This has two disadvantages, one being that it takes too much computing power for such a simple task, and also

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

depending on other tasks that our code executes it is possible to miss a short (quick) button press. To solve this problem, we can use interrupts, which, when set up properly, never miss these events and do not check continuously. When a button press is detected, the microcontroller (or processor) is notified and immediately stops the execution of our main code, executes a function that we created (interrupt/event handler), and then continues with our main code. You can easily observe that the interrupt handler must be really fast and usually is only one line of code. This is important in order for the interrupt to be handled fast and the execution of the main code to be stopped only for a few nanoseconds.

In order to set up an interrupt, start by creating the following circuit. Connect one side of the button to the GP0 and the other to 3.3VOUT.



For the code, we begin by importing the Pin class from the machine module. Next, we create the interrupt handler function which only prints "Button Pressed". After that, we have to declare a Pin for the button as input and also enable the pull-down resistor of the Raspberry Pi Pico. The pull-down resistor is essential because when we do not press the button the GP0 pin of the Raspberry Pi Pico is left hanging and can easily pick interference from the environment, making it go HIGH or LOW unexpectedly. The pull-down resistor prevents that by pulling GP0 to LOW. This resistor is weak so it can easily be overwritten by the 3.3V of the button. The last thing we have to do is to actually enable the interrupt on the button pin. We do that by passing to the method button.irq() the name of our interrupt handler and also the trigger mode which can be Pin.IRQ_RISING or Pin.IRQ_FALLING. In our case, when we press the button, the level starts from LOW and then becomes HIGH so we are going to use Pin.IRQ_RISING.

```
from machine import Pin

def handle_button_press(button):
    print("Button Pressed")

button = Pin(0, Pin.IN, Pin.PULL_DOWN)
button.irq(handler=handle_button_press, trigger=Pin.IRQ_RISING)
```

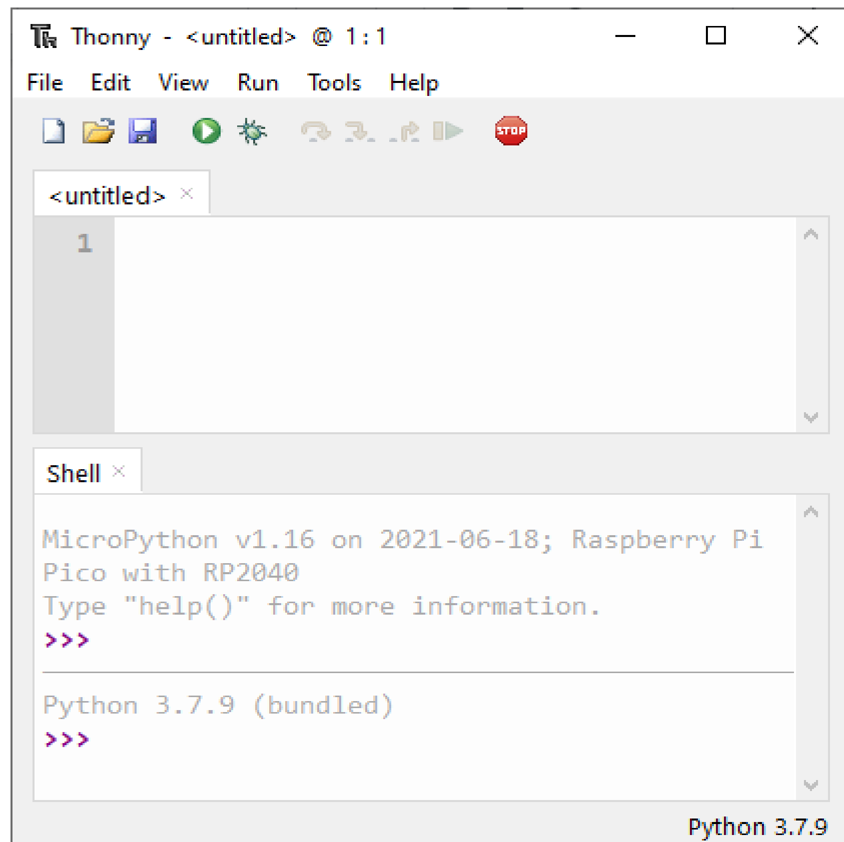
To upload the above code to Raspberry Pi Pico and try it, read the following subsection first.

6.3.7 Thonny Python IDE

In order to program the Raspberry Pi Pico, we need a code editor where we can write our code and then upload it to the board. We will use the Thonny editor. To download it, go to https://thonny.org/. Once you have finished downloading and installing Thonny, open it and you should see the following interface. Some useful icons are shown in the toolbar just under the menu bar. The first toolbar icon is used to create a new file, the second one is used to open an existing file, the third one is used to save the current file, the fourth one is used to execute the

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

code of the current file and the last one is used to stop the execution of any code. The rest of the buttons are used to debug the code of the currently opened file, i.e., to execute it step-by-step to inspect its operation so that you can find any bugs. Sometimes the fourth icon may be disabled, which means that there is a code being executed right now and in order to enable it you have to press the stop button first.



To proceed, we need to connect the Raspberry Pi Pico to our computer. Next, we have to inform Thonny that we are writing code for the Raspberry Pi Pico. To do that, we press click on the bottom right corner where it says Python 3.7.9 or some other version number. From there we select the option "MicroPython (Raspberry Pi Pico)". In case this option is not available, select "Configure interpreter…". Under the "Which interpreter or device should Thonny use for running your code?" there is a drop-down list from which you can select "MicroPython (Raspberry Pi Pico)" and then press OK.

Now we are ready for our first code. If you look at the pinout diagram, on the top of the Raspberry Pi Pico there is a pin named LED (GP25). This pin connects to an integrated LED on the board which we can control from GP25. First, we have to import the Pin class from the machine module, in order to control any GPIO.

```
from machine import Pin
```

After that, we have to declare which pin we are going to use and also its direction. GPIO pins have two directions, OUT or IN and, in order to easily differentiate between them, we should think if the current gets OUT of the pin or IN the pin. In our case, the current gets OUT of the pin in order to power the LED, so we declare the pin 25 as output by creating an object and assigning it to a variable as follows:

```
led = Pin(25, Pin.OUT)
```


The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

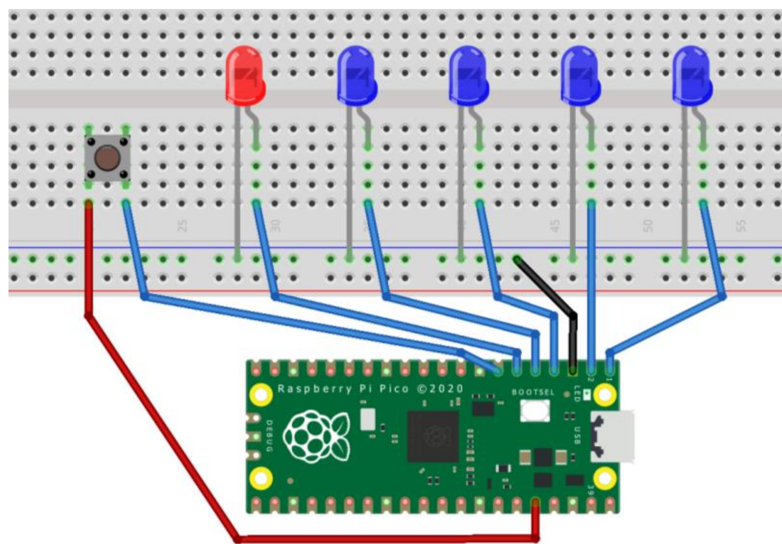
Now, in order to turn the led on or off, we can simply call the on or off methods of the led object.

```
led.on()
led.off()
```

We can now press the play button in order to run our code on the Raspberry Pi Pico and observe that the onboard LED lights up when led.on() is executed. The LED is turned off when led.off() is executed. When we are prompted to save the file, we can choose to save it on our computer.

6.4 Practice

We will start this learning activity by creating a circuit that connects one button, one red LED, and four blue LEDs to the Raspberry Pi Pico board. The LEDs are connected to pins GP0, GP1, GP2, GP3, and GP4, and the button to the pin GP5.



The first thing we have to do in our code is to import the Pin class and sleep function from machine and time modules respectively. We need those in order to control the GPIO pins and to create some delays whenever it is needed.

```
from machine import Pin
from time import sleep
```

Next, we have to create the classes for all of the components we are going to use. The first class we are going to create is for the Led. It will have an initializer that will take the pin number and the LED color as arguments. Also, it will have two methods for turning the LED on and off. In the initializer we also declare the Pin for the LED as output and save it to a member variable named pin.

```
class Led:
    def __init__(self, pin_number, color):
        self.color = color
        self.pin = Pin(pin_number, Pin.OUT)

    def on(self):
        self.pin.on()

    def off(self):
```


The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
        self.pin.off()
```

We can test the above code by creating an object for the onboard LED which is connected to GP25 and turning it on and off.

```
led = Led(25, "Green")
led.on()
sleep(1)
led.off()
```

Another class we have to create is for the button. It will have an initializer that takes the pin_number as an argument. Also, this class will have a method that returns if the button was pressed (as we had with micro:bit) and the interrupt handler. The initializer declares the button pin as input and enables the pull-down resistor on that pin. It also initializes the pressed variable to False since the button has not yet been pressed and finally sets up the interrupt. The was_pressed method returns True if the button was pressed, otherwise, it returns False. Before it returns True, it has to reset the state of the pressed member variable. The handle_button_pressed method just changes the state of the pressed member variable to True.

```
class Button:
    def __init__(self, pin_number):
        self.pin = Pin(pin_number, Pin.IN, Pin.PULL_DOWN)
        self.pressed = False
        self.pin.irq(handler=self.handle_button_pressed, trigger=Pin.IRQ_RISING)

    def was_pressed(self):
        if (self.pressed):
            self.pressed = False
            return True
        return False

    def handle_button_pressed(self, pin):
        self.pressed = True
```

The last class we have to create is called Display. The actual display consists of 5 LEDs which we want to control from this class. Consequently, the initializer of this new class just takes a list of LED objects as an argument and initializes the position of the LED to be turned on to 0. Also, it has a method that turns off the currently lit LED and turns on the next one. If it has reached the last LED, it continues with the first one again. In this whole process, the member variable position is updated in order to keep in memory which LED is currently lit. Another method is used in order to retrieve the color of the currently lit LED. Two more methods light all LEDs on or off.

```
class Display:
    def __init__(self, leds):
        self.leds = leds
        self.position = 0

    def next_led(self):
        self.leds[self.position].off()
        self.position += 1
        if self.position >= len(self.leds):
            self.position = 0
        self.leds[self.position].on()
```

```
    def color(self):
        return self.leds[self.position].color

    def all_on(self):
        for led in self.leds:
            led.on()

    def all_off(self):
        for led in self.leds:
            led.off()
```

Now that every class we are going to need is ready, we can continue to the main part of our code. We begin by creating a list of Led objects giving the *pin_number* and color of each Led we have connected. Next, we create a Button passing the pin number of the button and a Display object passing the list of Led objects:

```
leds = [Led(0, "Blue"), Led(1, "Blue"), Led(2, "Blue"), Led(3, "Blue"), Led(4, "Red")]
button = Button(5)
display = Display(leds)
```

For the last part of our code, we create a *while True* loop so our game can run forever. Inside, while the button is not pressed, we light up the next LED of the display after 0.1 seconds. Once the button is pressed, we check if the color of the currently lit LED is "Blue" and, if it is, we light all the LEDs of the display for 2 seconds to indicate that the player lost. Otherwise, if the LED's color is "Red", we keep it on for 2 seconds to indicate that the player has stopped the game at the desired point/LED.

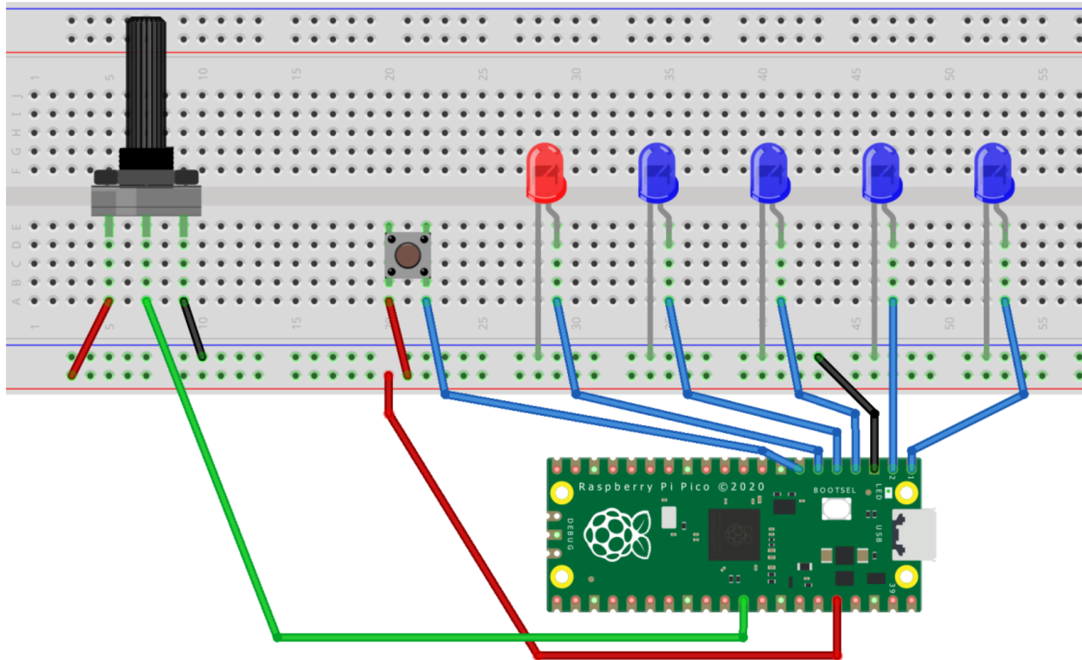
```
while True:
    while not button.was_pressed():
        display.next_led()
        sleep(0.1)
    if display.color() == "Blue":
        display.all_on()
        sleep(2)
        display.all_off()
    else:
        sleep(2)
```

Extension 1:

We can extend our game by adding a potentiometer, with which we will control the speed of the LEDs flashing. Start by adding a potentiometer to your circuit as shown below. Connect the left most pin of the potentiometer to 3.3V, the rightmost pin to the GND, and the middle one to the ADC0 (GP26) pin of Raspberry Pi Pico.

Co-funded by the
Erasmus+ Programme
of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



In our code, we have to create a new class for the potentiometer, named Potentiometer. Inside it we will have an initializer that takes the pin number of the potentiometer as an argument. The class will also have a method that reads and returns the position of the potentiometer.

```
from machine import Pin, ADC


class Potentiometer:
        def __init__(self, pin_number):
                self.pin = ADC(Pin(pin_number))

        def value(self):
                return self.pin.read_u16()
```

The read_u16 method returns a value from 0 to 65535, which is a really big range if we want to use it for the delay as is. To solve this in our code, we will divide the value returned from the method by 100000, which will result in a range from 0 to 0.65535. This way we can replace the 0.1 second delay with the above expression as explained.

```
leds = [Led(0, "Blue"), Led(1, "Blue"), Led(2, "Blue"), Led(3,                "Blue"), Led(4,
"Red")]
button = Button(5)
display = Display(leds)
pot = Potentiometer(26)

while True:
   while not button.was_pressed():
      display.next_led()
      sleep(pot.value()/100000)
   if display.color() == "Blue":
      display.all_on()
      sleep(2)
      display.all_off()
   else:
```

Co-funded by the
Erasmus+ Programme
of the European Union

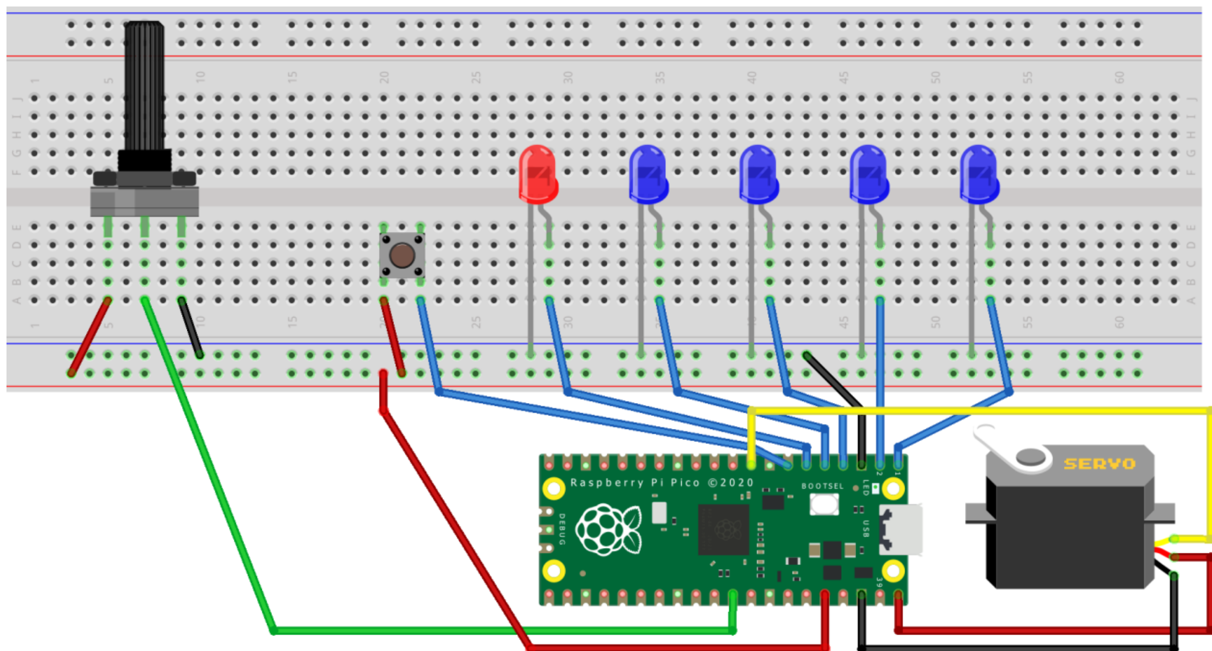
The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

```
        sleep(2)
```

Upload the code to the Raspberry Pi Pico and try adjusting the speed of the game using the potentiometer.

Extension 2:

We can extend our game even more by creating a scoreboard using a servo motor and a paper (or 3D printed) board with distinct scores drawn in a semi-circle. Each time the player hits the right LED, the servo moves clockwise, and otherwise it turns counterclockwise, indicating the current score. First, create the following circuit using the SG90 servo and by connecting the brown wire to GND, the red wire to VBUS, and the orange/yellow wire to GP6.



Now we can continue by creating a Servo class in our code. This class will have an initializer and a method called angle for controlling the position of the servo's axle. The initializer takes as an argument the *pin_number*, creates a PWM/Pin instance, then sets the frequency of PWM to 50Hz and initializes the servo position at 0. The angle method takes as an argument the desired angle, which must be between 0 and 180. Otherwise, if it is bigger, it will be set to 180 and, if it is smaller, it will be set to 0. To set the duty cycle of our PWM signal we use the Pin's method *duty_u16* which takes a value from 0 to 65535 as an argument. To convert the angle given to us into a value for the method we use the formula *(angle*38)+1311*.

```
class Servo:
    def __init__(self, pin_number):
        self.pin = PWM(Pin(pin_number))
        self.pin.freq(50)
        self.angle(0)

    def angle(self, angle):
        if angle<0:
            angle = 0
        elif angle>180:
            angle = 180
        self.pin.duty_u16((angle*38)+1311)
```

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Next, we have to create a class named Points that keeps and updates the score. The initializer of this class takes a Servo class instance as an argument and initializes the member variable named points to 0. The class also has two methods named won and lost that handle point winning and losing and also update the servo position. Since the 180 distinct positions are too many, we can increase or decrease the points by 10 which will translate to 18 distinct positions.

```
class Points:
    def __init__(self, servo):
        self.points = 0
        self.servo = servo

    def won(self):
        self.points += 10
        self.servo.angle(self.points)

    def lost(self):
        self.points -= 10
        self.servo.angle(self.points)
```

To use our new classes, we first create a Servo instance giving the pin number 6 as an argument and then we pass the Servo instance to the initializer of the Points class.

```
servo = Servo(6)
points = Points(servo)
```

The last thing we have to do is to call the won method of Points when we stop the game on the correct LED. Otherwise, we call the lost method. The main loop of our code will now look like this.

```
while True:
    while not button.was_pressed():
        display.next_led()
        sleep(pot.value()/100000)
    if display.color() == "Blue":
        points.lost()
        display.all_on()
        sleep(2)
        display.all_off()
    else:
        points.won()
        sleep(2)
```

When creating the code, pay attention that on top we put the imports, and below them we define the classes followed by the instance creation and then finally our main loop.

6.5 Time for fun

Here are some ideas for further extension of the game:

1. Replace the rightmost blue LED with a red one. Revise the code so that the flashing of the LEDs goes from right to left and then from left to right. The player has to press the button when any of the two red LEDs are on in order to win.

2. Change the previous extension by putting two blue LEDs at the left and the right end and a red LED in the middle. In order to win, the player has to press the button when the red LED is on. Revise the code to achieve this way of operation with the new setting of the LEDs.

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

3. Put a time limit to the game duration. The player can continue playing until the time limit is reached.

6.6 Self check

1. The initializer of a Class is a method named:
   a. initializer
   b. init
   c. __init__
   d. Class
2. An LED:
   a. Stands for Light Emitting Diode
   b. May or may not need a resistor in series, depending on the input voltage
   c. Has polarity
   d. All of the above
3. Potentiometers are:
   a. Variable resistors
   b. Variable capacitors
   c. Fixed value resistors
   d. Fixed value capacitors
4. Buttons:
   a. Are used like switches that intercept a circuit
   b. Need a resistor parallel to them
   c. Need a capacitor parallel to them
   d. None of the above
5. Servos rotate to 0 degrees and 180 degrees when they receive a pulse width of:
   a. 0ms and 180ms respectively
   b. 1ms and 2ms respectively
   c. 0.5ms and 1.5ms respectively
   d. 0ms and 1ms respectively
6. Interrupts:
   a. Stop the normal execution of our code to run usually a single line of code before returning to normal execution
   b. Occur after an event, for example, button press
   c. Both a. and b.
   d. Occur only when a signal goes from LOW to HIGH
7. To define a Pin instance as an input, we pass to the initializer, together with the pin number:

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

a. The value INPUT
b. The value Pin.INPUT
c. The value Pin.OUT
d. The value Pin.IN

8. To turn a Pin instance on and off, we call the methods:

a. On and off
b. High and low
c. Open and close
d. Up and down

9. To make sure that a pin connected to a button is not left hanging, we usually enable a:

a. Pull-up resistor
b. Pull-down resistor
c. Parallel capacitor
d. None of the above

10. Classes:

a. Are prototypes of objects
b. Are objects
c. Must always have an initializer
d. None of the above

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

References

MAKERS website: https://makers-project.eu/

Tutorials for learning Python: https://www.learnpython.org/

Teaching programming using concepts of cooking and cooking recipes:
http://www.xandaschofield.com/2016/02/teaching-computer-science-with-cookies.html

Micro:bit site: https://microbit.org/

Scratch site: https://scratch.mit.edu/

Raspberry Pi Pico: https://www.raspberrypi.com/products/raspberry-pi-pico/

Github repository with the complete code presented in this module:
https://github.com/tucmakers/moduleA

Co-funded by the
Erasmus+ Programme
of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.