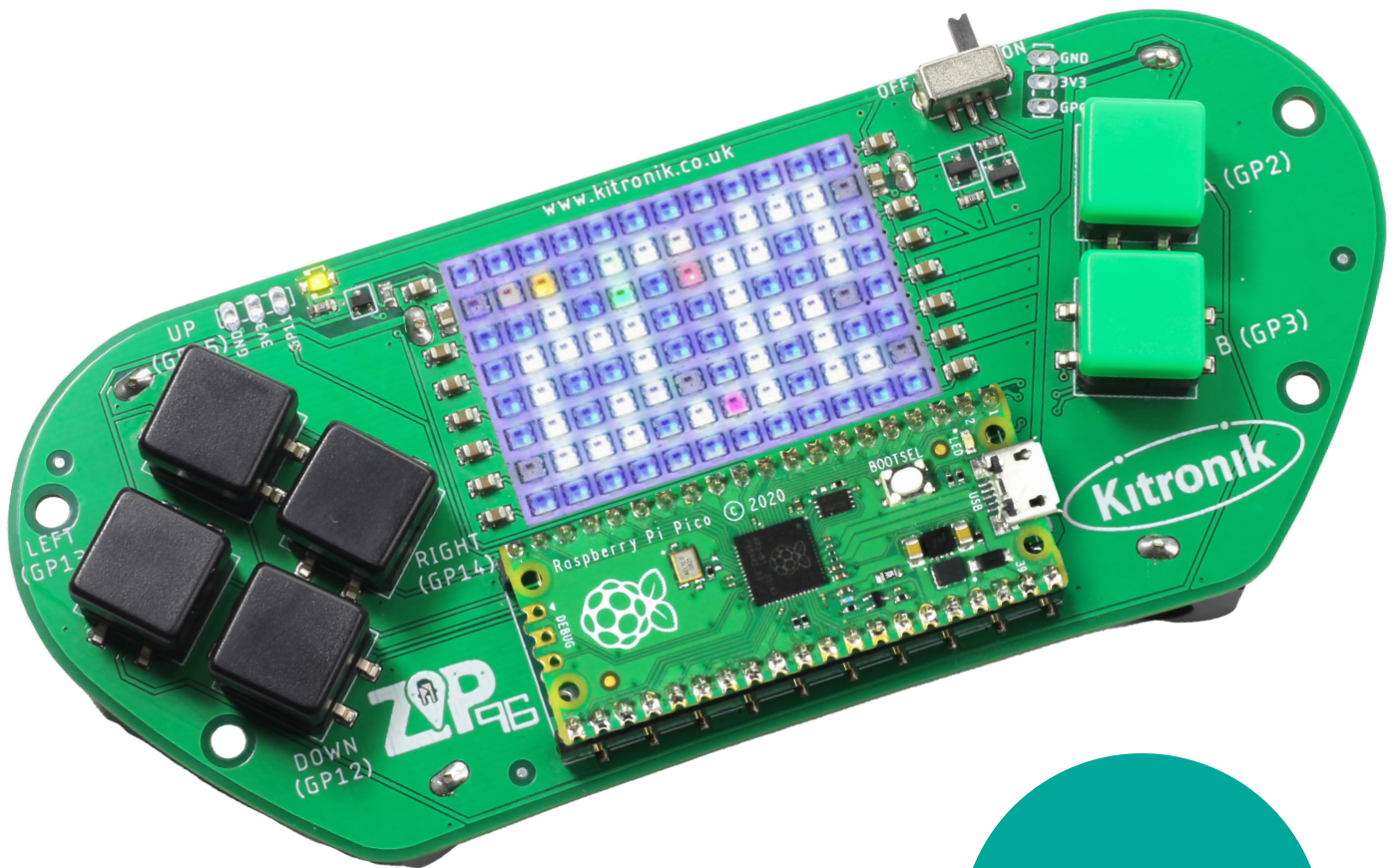


LESSON GUIDE TO THE PICO ZIP96



LESSON 4: THE A-MAZING GAME



This lesson includes curriculum mapping, practical exercises and a linked PowerPoint presentation.

14+

www.kitronik.co.uk

TEACH YOUR STUDENTS HOW TO CREATE GAMES WITH CODE!

LESSON 4

INTRODUCTION & SETUP



This is the third lesson in the 'A-mazing Game' series for Pico ZIP96. Building on the movement functionality added before, this lesson will look at taking inputs from the Pico ZIP96 and using them to control the player.



CLASSROOM SETUP



Students will be working in pairs. They will need:

- Pen & Paper
- A computer/laptop with a USB port and Internet access
- Raspberry Pi Pico H
- Kitronik Pico ZIP96
- 3 x AA batteries
- A micro USB cable
- A copy of the Kitronik ZIP96 library added in step 4 of the Thonny setup, or ZIP96Pico.py in Lessons Code folder
- A copy of last lesson's code (ZIP96Pico - A-Mazing Game - Lesson 03.py in Lessons Code folder)

The teacher will be writing on the board as well as demonstrating code on a projected board (if available).



Curriculum mapping

- Understanding tools for writing programs. Using sequence, variables, data types, inputs and outputs.
- Apply iteration in program designs using loops.
- Make decisions in programs, making use of arithmetic, logic and Boolean expressions and operators. Created nested selection statements.
- Decompose problems into smaller components, and make use of subroutines to build up well-structured programs.

KEYWORDS:

TRANSLATORS, INTEGRATED DEVELOPMENT ENVIRONMENTS (IDES), ERRORS, SEQUENCE, VARIABLES, DATA TYPES, INPUTS, OUTPUTS, ITERATION, WHILE LOOPS, FOR LOOPS, NESTED STATEMENTS, DESIGN PROGRAMS, SELECTION, CONTROL STRUCTURES, LOGIC, BOOLEAN, NESTED STATEMENTS, SUBROUTINES, PROCEDURES, FUNCTIONS, MODULES, LIBRARIES, VARIABLE SCOPE, WELL-DESIGNED PROGRAMS

LESSON 4

WHAT IS OUR A-MAZING GAME?

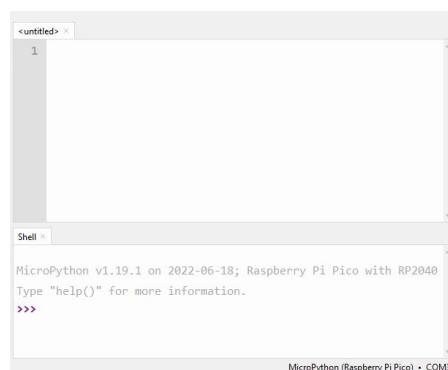
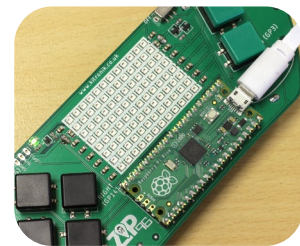


Our A-Mazing Game is a maze-based game where the player runs through a maze collecting gems. Once you have collected all the gems in the maze then you have won! But it won't be that easy, as there are enemies in the maze trying to catch you before you collect all of the gems. The three enemies each have their own level of difficulty. The first enemy moves randomly. The second enemy tries to move towards you, without trying to avoid the maze walls. The third and final enemy is smart and moves around the maze walls finding the best path to you. When an enemy catches you, you lose a life and everyone in the game is reset back to their starting positions. You have three lives to collect all the gems, and if you don't, then you lose.

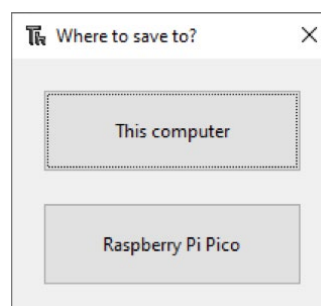
SETUP

Students will need to work in pairs, having one device per pair.

- 1 Start by having the students setup the ZIP96 Pico by connecting it to a computer and opening up Thonny.
- 2 The Pico device will appear in the bottom right corner of Thonny.
 - If the device does not load automatically, try pressing the STOP icon at the top of the screen.
 - If the shell does not load automatically, turn it on by checking **View > Shell**.



- 3 Create a new file by clicking **File > New** and save this to your Pico as main.py by clicking **File > Save as** and selecting **Raspberry Pi Pico**.



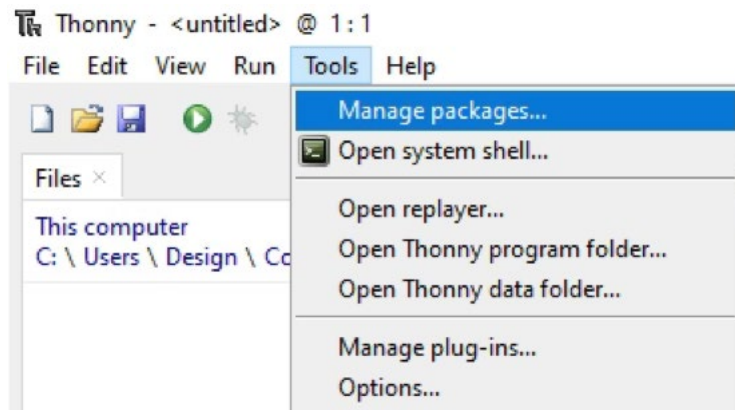
LESSON 4

SETUP CONTINUED

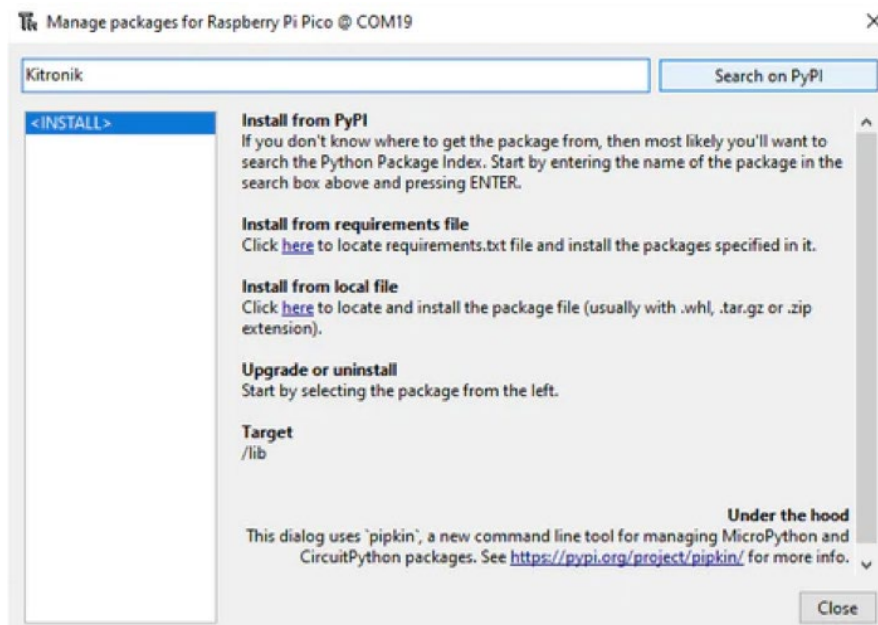
- 4 Now we can install the ZIP96Pico library onto our Pico.



- To do this we need to click **Tools > Manage Packages...** from the drop down menu.



- With the Manage packages window open we can now search for Kitronik in the text box at the top, and click the Search on PyPI button. Thonny will search the Python Package Index for all the Kitronik packages.



- Click on KitronikPicoZIP96 from the search results list. This will show us details about the package and from here we can click the Install button to add the package to our Pico. Thonny may ask you to confirm that you would like to install this package and we want to select Yes, we do want to install the package.

LESSON 4

MAIN LESSON



Curriculum mapping

Decompose problems into smaller components, and make use of subroutines to build up well-structured programs.



CREATE GAME CLASS: *WALL*

Now that we have a working Player class, let's start adding walls to our maze. In the Wall class we will again need to define a constructor. The Wall will need to know:

- its x and y position on the screen,
- the colour used for its LED,
- the screen object to draw the player on the ZIP96.

So let's take these as input parameters in the Wall constructor and set them as variables stored inside the object. Then create a draw function inside the Wall class in the same way we did in the Player class. We can again add a call to the draw function inside the constructor for our Wall class.



```
# Class to store our Wall functionality
class Wall():
    # Wall object constructor
    def __init__(self, x, y, colour, screen):
        self.x = x
        self.y = y
        self.colour = colour
        self.screen = screen
        #Draw the starting position
        self.draw()

    # Draw the current position on the screen
    def draw(self):
        self.screen.setLEDMatrix(self.x, self.y, self.colour)
```



Curriculum mapping

Learn how to handle strings, and simplify solutions by making use of lists and arrays. Apply iteration in program designs using loops.

LESSON 4

CREATE GAME WALL OBJECTS



To setup the walls for our maze we need to first decide where we want them to go. Above where we create our player object, let's create an array wallsXY of the x and y coordinates for each Wall object in our maze. To help understand the layout we can use the diagram below.

A-Mazing Game Walls Layout

	0	1	2	3	4	5	6	7	8	9	10	11
0	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue
1	White	White	White	Blue	White	White	White	Blue	White	White	White	White
2	Blue	Blue	White	Blue	White	Blue	White	White	White	Blue	White	Blue
3	Blue	White	White	White	White	Blue	Blue	White	Blue	Blue	White	Blue
4	Blue	White	Blue	Blue	White	Blue	Blue	White	White	White	White	Blue
5	Blue	White	Blue	White	White	White	Blue	White	Blue	White	Blue	Blue
6	White	White	White	White	Blue	White	White	White	Blue	White	White	White
7	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue

This is how our walls layout will look in our wallsXY array.



```
# List to store the coordinates of our walls
wallsXY = [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0), (8, 0), (9, 0), (10, 0), (11, 0),
           (3, 1), (7, 1),
           (0, 2), (1, 2), (3, 2), (5, 2), (9, 2), (11, 2),
           (0, 3), (5, 3), (6, 3), (8, 3), (9, 3), (11, 3),
           (0, 4), (2, 4), (3, 4), (5, 4), (6, 4), (11, 4),
           (0, 5), (2, 5), (6, 5), (8, 5), (10, 5), (11, 5),
           (4, 6), (8, 6),
           (0, 7), (1, 7), (2, 7), (3, 7), (4, 7), (5, 7), (6, 7), (7, 7), (8, 7), (9, 7), (10, 7), (11, 7)]
```

LESSON

4

With the coordinates for the walls stored, we can now create an empty array walls to store all of the Wall objects. We will fill the walls array by looping through each x and y pair stored in wallsXY, using these coordinates to create a Wall object for each pair and append it to the end of the walls array. To do this we'll use a for each loop which in Python can be setup using the for keyword, followed by a new variable name wallXY, the keyword in and finally the thing we want to loop through. In our case we are looping through an array of x, y coordinate pairs in wallsXY and so each time we loop, the wallXY variable will be set to a different x, y coordinate pair from our wallsXY array.



```
# List to store our wall objects
walls = []
# Loop to create wall objects using the wallsXY coordinates
for wallXY in wallsXY:
    # Add the wall objects to our walls list
    walls.append(Wall(wallXY[0], wallXY[1], gamer.Screen.BLUE, screen))
```

Next we are going to need access to the walls inside the Player class, so let's add the walls array as an input parameter in the Player constructor.



```
def __init__(self, x, y, colour, walls, screen, screenWidth, screenHeight):
    self.x = x
    self.y = y
    self.colour = colour
    self.walls = walls
```

TASK: TEST WALL CONSTRUCTOR

With our Wall class written we can now test our maze with the walls added.

Note: Don't forget to update the player object's inputs to include the walls array.

```
# Create an object for our player
player = Player(0, 1, gamer.Screen.YELLOW, walls, screen, screenWidth, screenHeight)
```



What is the problem with the maze setup at the moment? How can it be fixed?



CREATE GAME FUNCTION: *WALL.COLLISION*



To stop our player from being able to move onto a Wall object we need to add a function collision to the Wall class. This function should accept an x and a y value as inputs and check them against its own coordinates to return if they match, meaning they would be in the same position. In this function we use Boolean logic which says that only when `self.x == x` is True and `self.y == y` is True we should return True. In all other cases we should return False.

```
# Check if the given x and y are the same as our current position
def collision(self, x, y):
    return self.x == x and self.y == y
```

We can then add calls to the `Wall.collision` function inside of our `Player.move` function to check if we move onto a Wall object. Let's do this underneath the line in the move function where we check if y is off the bottom edge of the screen.

To perform the check we need to look through every Wall object in the walls array that we take as an input in the Player constructor. For this we'll use another for each loop. We'll use the variable name `wall`, and we want to loop through the walls array.



Curriculum mapping

Make decisions in programs, making use of arithmetic, logic and Boolean expressions and operators.

Inside the loop we can use `wall` to access our Wall object and so we want to call the collision function with the player object's new position. When there is a collision with a Wall object then we need to undo the move made, which we can do using the `--` operator on our object's position variables. This is similar to how we update the object's position but instead of adding the input values x and y we want to minus them to reset the object's position back to what it was before. Then when we have collided with a wall we cannot collide with another, so we don't need to check the rest of the walls in the list. To skip the rest of the list we'll use the `break` command which exits the loop, even when there are more walls to check.

```
if (self.y >= self.screenHeight): self.y = 0

# Check each wall in our list
for wall in self.walls:
    # If the player is colliding with a wall
    if wall.collision(self.x, self.y):
        # Undo the new position using the x and y parameters
        self.x -= x
        self.y -= y
        # Cannot collide with more than one wall so leave loop
        break
```



TASK: TEST *WALL.COLLISION* FUNCTION

Now try moving the player object around the maze and see if you can still move onto Wall objects. To best test our new function, try moving onto a Wall in every direction, up, down, left and right.

LESSON
4

CONCLUSION

LESSON 04 CONCLUSION



In this lesson, you have:

- Created Wall class, constructor and methods
- Used for loop, if statement and Boolean logic to check for interaction between Player and Wall objects

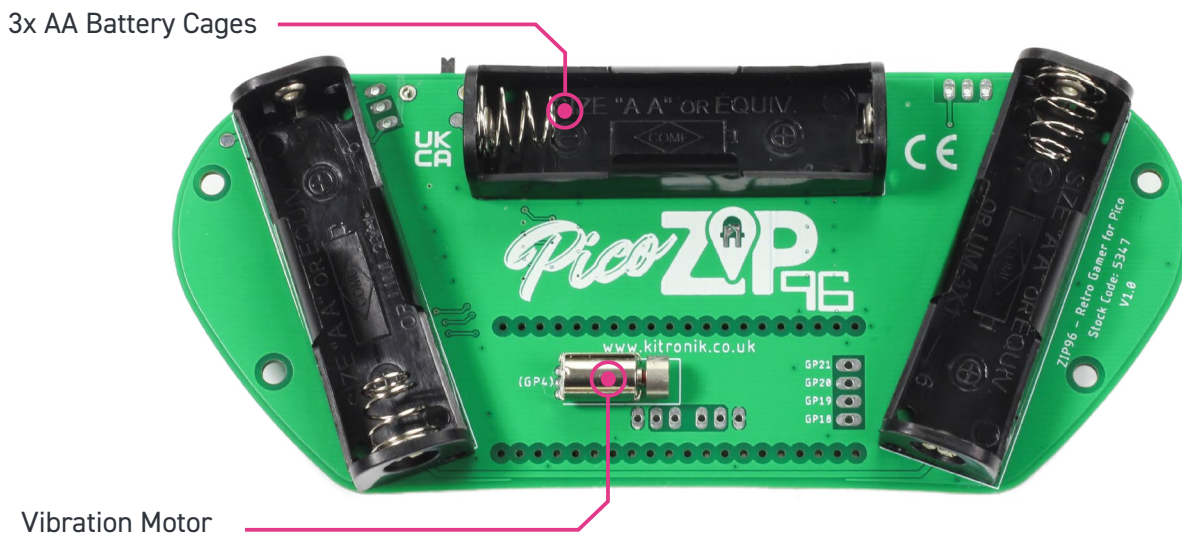
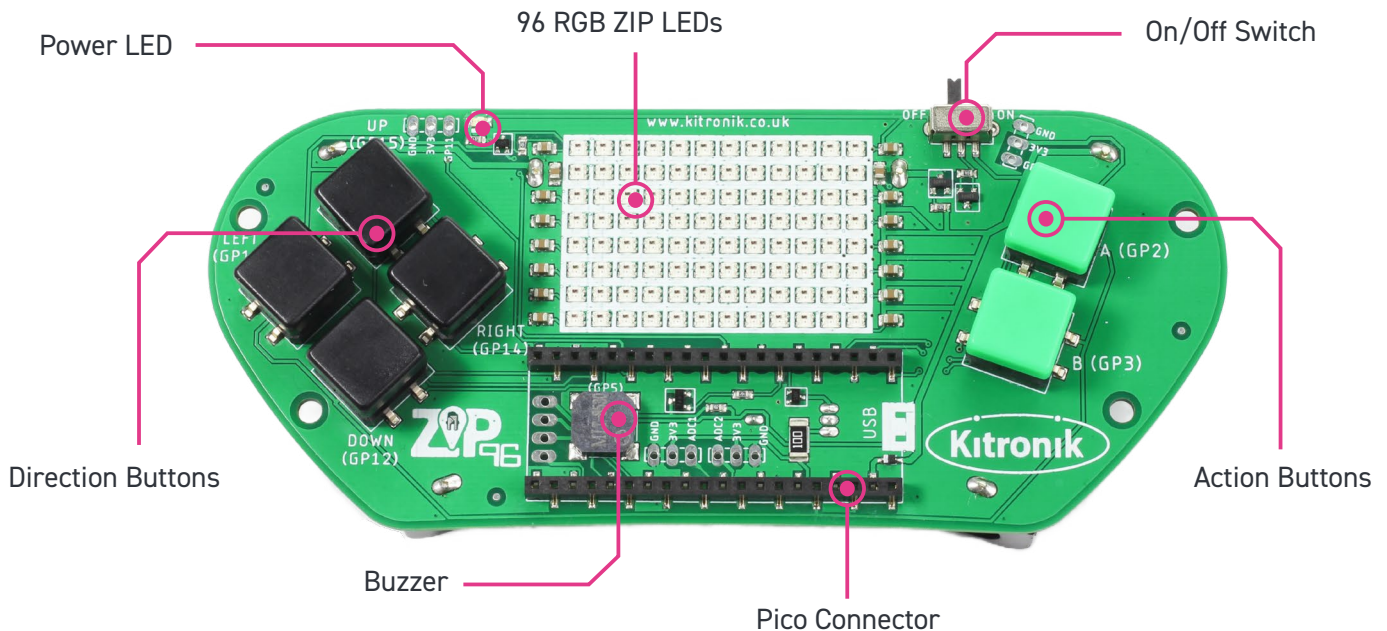
FULL LESSON CODE

The full code for each lesson can be found in the Lessons Code folder.



THE ZIP96 IS A PROGRAMMABLE RETRO GAMEPAD FOR THE RASPBERRY PI PICO.

It features 96 colour addressable LEDs arranged in a 12 x 8 display, a buzzer for audio feedback, a vibration motor for haptic feedback, and 6 input buttons. It also breaks out GP1, GP11, ADC1 and ADC2, along with a set of 3.3V and GND for each, to standard 0.1" footprints. GP18 to 21 are also broken out on a 0.1" footprint underneath the Pico. The Pico is connected via low profile 20-way pin sockets.



T: 0115 970 4243

W: www.kitronik.co.uk

E: support@kitronik.co.uk



[kitronik.co.uk/twitter](https://twitter.com/kitronik)



[kitronik.co.uk/facebook](https://www.facebook.com/kitronik)



[kitronik.co.uk/youtube](https://www.youtube.com/kitronik)



[kitronik.co.uk/instagram](https://www.instagram.com/kitronik)



Designed & manufactured
in the UK by **Kitronik**



For more information on RoHs and CE please visit kitronik.co.uk/rohs-ce. Children assembling this product should be supervised by a competent adult. The product contains small parts so should be kept out of reach of children under 3 years old.