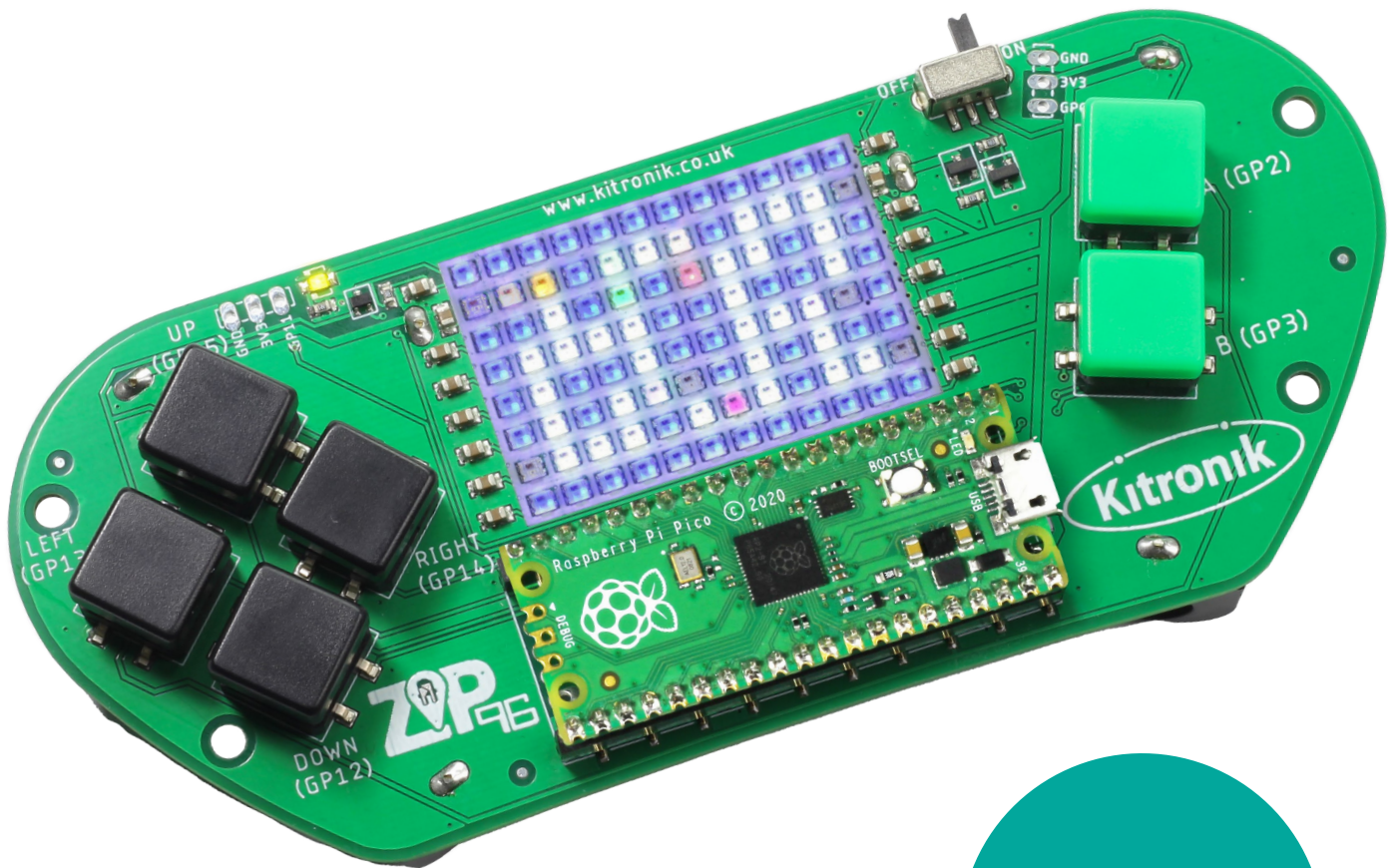


# LESSON GUIDE TO THE PICO ZIP96



## LESSON 11: THE A-MAZING GAME



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

14+

[www.kitronik.co.uk](http://www.kitronik.co.uk)

**TEACH YOUR STUDENTS HOW TO CREATE GAMES WITH CODE!**

# LESSON 11

## INTRODUCTION & SETUP



This is the eleventh lesson in the 'A-mazing Game' series for Pico ZIP96. With the first, randomly moving enemy now providing some extra challenge for the player, this lesson adds a second enemy with the ability to move towards 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 (ZIP96Pico.py in Lessons Code folder)
- A copy of last lesson's code (ZIP96Pico - A-Mazing Game - Lesson 09.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.
- Decompose problems and solve them using algorithms. Explore different searching and sorting algorithms.
- Decompose problems into smaller components, and make use of subroutines to build up well-structured programs.
- Learn how to handle strings, and simplify solutions by making use of lists and arrays.
- Apply iteration in program designs using loops.
- Make decisions in programs, making use of arithmetic, logic and Boolean expressions and operators. Create nested selection statements.

### KEYWORDS:

TRANSLATORS, 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, STRINGS, LISTS, STRING HANDLING, ARRAYS (1D, 2D), MANIPULATION, ITERATION, ALGORITHMS, DECOMPOSITION, ABSTRACTION, DESIGN METHODS, TRACE TABLES, SEARCHING AND SORTING ALGORITHMS

# LESSON 11

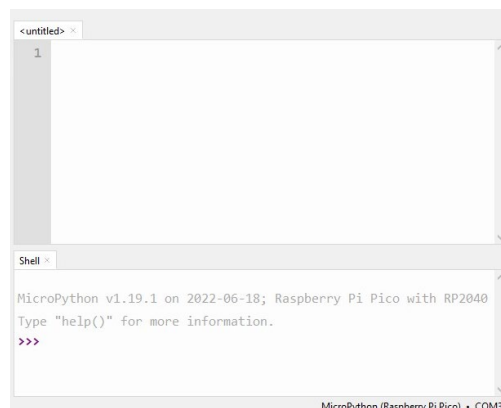
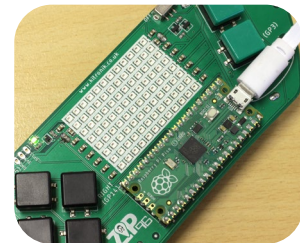
## WHAT IS OUR A-MAZING GAME?



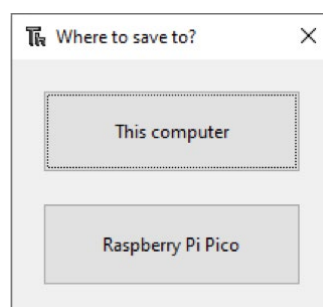
Our A-Mazing Game is a maze-based game where the player runs through a maze collecting gems. Once you have collected all the gems in the maze then you have won! But it won't be that easy, as there are enemies in the maze trying to catch you before you collect all of the gems. The three enemies each have 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

- 1 Start by having the student's 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**.

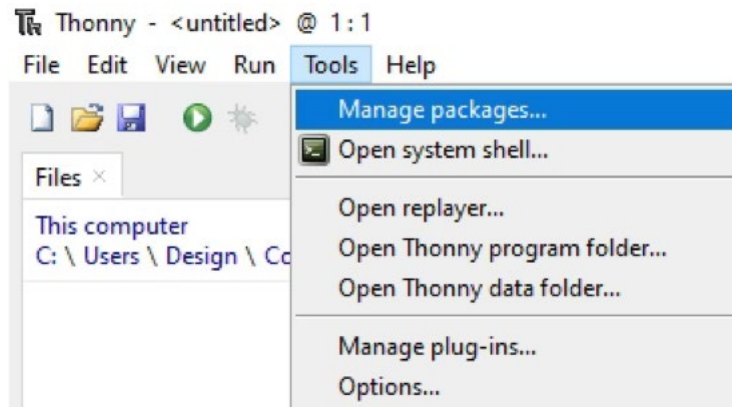


## SETUP CONTINUED

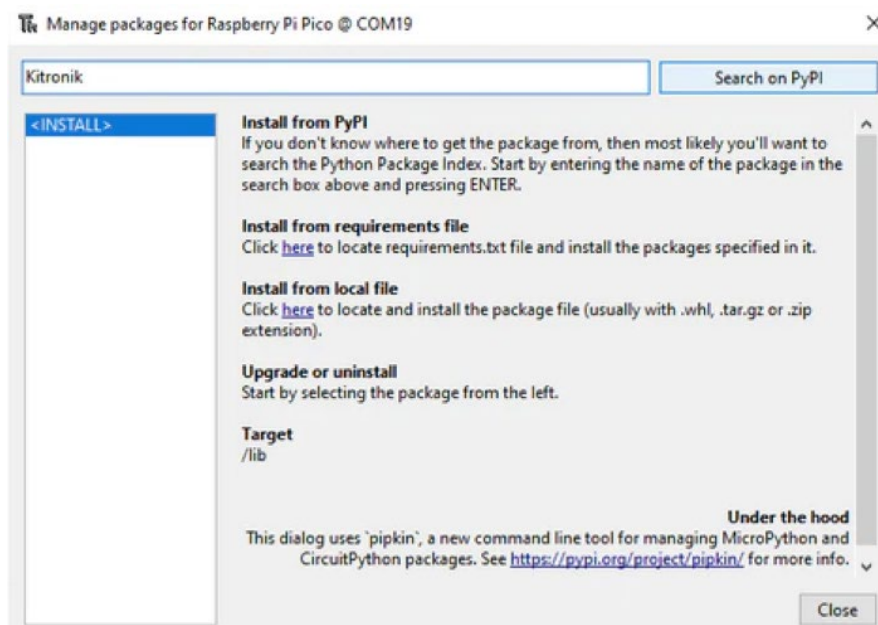
- 4 Now we can install the ZIP96Pico library onto our Pico.



- 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 11

## MAIN LESSON



### Curriculum mapping

Decompose problems and solve them using algorithms. Explore different searching and sorting algorithms.

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



### CREATE GAME CLASS: *POSITION*

The last thing we have left to do for our game is to write the code for a smart Enemy that moves around the maze walls finding the best path to the player. To create this final Enemy we will use the algorithm we set out in the last lesson. As part of the algorithm we use positions in the maze that store the x and y location along with a distance travelled from the start. Let's create a new class Position to store these values.

```
# Class to store our Position functionality. Used by Enemy.moveSmart()
class Position():
    # Position object constructor
    def __init__(self, x, y, travelled):
        self.x = x
        self.y = y
        self.travelled = travelled
```

### CREATE GAME FUNCTION: *ENEMY.MOVESMART*



With our Position class setup we can create a new move function in the Enemy class to have the Enemy move along the shortest path towards the player. The start of the moveSmart function is the same as the moveRandom and moveNormal functions. We want to reset the current position on the screen and set hitPlayer to False.

```
# Update the current position towards Player x, y
def moveSmart(self):
    # Reset the current position on the screen
    self.drawEmpty()
    # Reset hitPlayer
    self.hitPlayer = False
```




### Curriculum mapping

Using sequence, variables, data types, inputs and outputs.

Learn how to handle strings, and simplify solutions by making use of lists and arrays.

## LESSON 11

Next we setup some variables to use in our search algorithm. The first is `current`, which we use to store a `Position` object at the location in the maze of the player object, with a distance travelled of zero. Then create an array `toVisit` of `Position` objects we want to check. We'll initialise this with the `current` variable we just created. Finally, we are going to need a 2 dimensional array `visited` that is the size of our maze and can contain all the `Position` objects we have already checked. The `Position` objects stored in `visited` will eventually reveal to us the shortest path between the player and the smart Enemy object. 

To start, we are going to initialise the `visited` array with the `None` object for every element. The `None` object is used to say there is no value saved. It is helpful to show us the positions we have been to, when we have a `Position` object stored there, and positions we haven't been to, when we have a `None` object stored there. We initialise the 2-dimensional array using two for loops. We can use for loops to setup a range of elements in an array. The inner arrays will have a length equal to the ZIP96 screen width, while the main outer array will have a length equal to the ZIP96 screen height. This means each inner array will contain 12 `None` objects and the outer array will contain 8 of the inner arrays.


```
# Create new Position at Player x, y with travelled distance zero
current = Position(self.player.x, self.player.y, 0)
# List of Positions to check
self.toVisit = [current]
# List of Positions already checked
self.visited = [[None for x in range(self.screenWidth)] for y in range(self.screenHeight)]
```



### Curriculum mapping

Apply iteration in program designs using loops.

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

To perform our search we need to use a while loop with the condition being to continue until there are no more `Position` objects in the `toVisit` array. Inside the loop we first want to get the `Position` object with the shortest distance travelled from the `toVisit` array. We can do this using the `pop` function which removes and returns the last element from the array and stores it in `current`. Then let's add the `current` `Position` object to the `visited` array using its x and y coordinates. Next, we'll call `collision` on the smart Enemy object to check if we have got a path back to its location. When we are back at the smart Enemy we should break out of the loop to stop the search. 

```
# While we have Positions to visit
while self.toVisit:
    # Take the end Position from toVisit
    # Lowest distance travelled from Player at the end of the list
    current = self.toVisit.pop()
    # Add the Position to the visited list
    self.visited[current.y][current.x] = current

    # Check if we have returned to the Enemy from the Player
    if self.collision(current.x, current.y):
        break
```



## LESSON 11

As our search makes a path from the player back to our smart Enemy object, we'll need to add the collision function into the Enemy class. This works like before, checking the given x and y input values against the Enemy coordinates to return if they match.



```
# 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
```

### CREATE GAME FUNCTION: *ENEMY.MOVESMARTUPDATE*

We are going to create a new function moveSmartUpdate in the Enemy class to check possible new positions and add them to the toVisit array if we haven't seen them before. At the start of the moveSmartUpdate function we'll do some standard checks we have been performing to tell when a location in the maze is valid. The difference with these checks is that we should just stop checking the location when it's not valid, rather than undo the move like we have done previously.



#### Curriculum mapping

Created nested selection statements.



There are four conditions which need checking, two for the x coordinate and two for the y coordinate. Rather than using four separate if statements, what can be used to reduce it to two?



To do this we'll add the if statements that check when a position is off the edge of the screen. Here we can reduce the number of if statements needed by using the bit wise or operator. This returns when the first condition is True or the second condition is True or both of them are True. Next we'll perform checks on the position to return when it's inside of the walls.



```
# Updates toVisit list with a valid Position
def moveSmartUpdate(self, x, y, travelled):
    # Check the new x position is valid (not off the edge of the screen)
    if (x < 0 or x >= self.screenWidth): return
    # Check the new y position is valid (not off the edge of the screen)
    if (y < 0 or y >= self.screenHeight): return

    # Check each Wall in our list
    for wall in self.walls:
        # If the Enemy is colliding with a Wall
        if wall.collision(x, y):
            # Not a valid move so return
            return
```

## LESSON 11

When we know the Position is valid we can check if we have visited it before. If we haven't seen it then we want to add it as a Position we want to visit by inserting it into the toVisit list. We want the toVisit list to be ordered with the longest distance travelled at the start, at index 0, and the shortest distance travelled at the end. To do this we are going to essentially use the insertion sort algorithm.



### Students take note

Insertion sort is an algorithm used to sort an array of elements. Insertion sort works by splitting the array of elements into two sections. The first section is the sorted part and is at the start of the array. The second section is the unsorted part and is at the end of the array. Insertion sort works by looking at the first element in the unsorted section and moving into its correct position in the sorted section of the array. This is done by looping through the elements in the sorted array from the largest value down to the smallest, and comparing the values of the unsorted element to the sorted elements. When the value of the sorted element is less than the unsorted element, we know the unsorted element should be placed here in the array. The sorted section now includes an extra element, and we repeat this process until there are no more elements in the unsorted section.



Our use of insertion sort uses the core principles from the actual algorithm, but instead we'll do an on the fly insertion sort. What this means is we'll sort the toVisit array by adding an element into their sorted position, rather than having a random array of elements and having to sort them.



We'll do this by looping through the toVisit array from the start where our Position with the largest distance travelled will be. At each Position we'll check if it is less than the new Position and when it is less we can break out of the loop. Outside of the loop we can now add the new Position into the array. We do this outside of the loop so that if our new Position has the smallest distance travelled then it is added onto the end of the array.

```
# If Position at x, y doesn't exists (is None)
if not self.visited[y][x]:
    i = 0

    # Check each Position in toVisit
    for i in range(len(self.toVisit)):
        # If Position in toVist travelled less distance leave loop
        if self.toVisit[i].travelled < travelled: break

    # Add new Position toVisit
    # Lowest distance travelled from Player at the end of the list
    self.toVisit.insert(i, Position(x, y, travelled))
```



## UPDATE GAME FUNCTION: *ENEMY.MOVESMART*



Back inside of the moveSmart function we are going to call the moveSmartUpdate function we just created on the positions around the current Position. Doing this allows us to test each of the different moves this Enemy object can make from the current Position. We want to add these function calls to the end of the while self.toVisit: loop. Inside these function calls the Position objects surrounding current will be checked to see if they are valid then added to the toVisit array when they are. We also want to add one onto the distance travelled of the current object, as any of its surrounding Position objects will take one extra move to get there for the current Position.

```
# Add the Positions around the current Position into the toVisit list,
# and increase their travelled distance by 1
self.moveSmartUpdate(current.x + 1, current.y, current.travelled + 1)
self.moveSmartUpdate(current.x - 1, current.y, current.travelled + 1)
self.moveSmartUpdate(current.x, current.y + 1, current.travelled + 1)
self.moveSmartUpdate(current.x, current.y - 1, current.travelled + 1)
```

That's all we need to do inside of the while loop, so we can now use the visited array to determine the best move the smart Enemy object can make from its current Position. To do that let's first get the Position object at the Enemy object's current location from the visited array. Using current we can also set the distance travelled from the Enemy to the player to determine when a move has a shorter path to the player.



```
# Get the current Enemy Position from the visited list
current = self.visited[self.y][self.x]
# Set the Enemy travelled distance
self.travelled = current.travelled
```

To check for the move with the shortest path to the player we should first check whether the surrounding position is off the edge of the maze. This prevents us from accessing a position outside of the array which would cause an error while our maze game is running.



```
# Checks if a position is the shorted path to the Player
def moveSmartCheck(self, x, y):
    # Check the new x position is valid (not off the edge of the screen)
    if (x < 0 or x >= self.screenWidth): return
    # Check the new y position is valid (not off the edge of the screen)
    if (y < 0 or y >= self.screenHeight): return
```

Now we know we aren't accessing out of bounds elements in our array, let's get the Position object we want to check from the visited array and store it in checked. Then we'll need to use an if statement to see whether checked stores an object or not. When it does store a Position object, the if statement will then see whether the distance travelled of checked is less than the distance travelled by the smart Enemy object's current Position. When it is we can update the Enemy object to move into this Position and update its travelled variable in case we have any more surrounding positions to check.



```
# Get Position at x, y from visited
checked = self.visited[y][x]

# If Position exists (is not None)
# And the distance travelled from Player is less
if checked and checked.travelled < self.travelled:
    # Update the lowest distance travelled
    self.travelled = checked.travelled
    # Change the current position to the x and y parameters
    self.x = x
    self.y = y
```

## UPDATE GAME FUNCTION: *ENEMY.MOVESMART*



At the end of our search in moveSmart we then call the moveSmartCheck function we just created on the positions around the Enemy, as these are the different moves this Enemy object can make. Inside these function calls the Enemy object's Position will be updated to be along the path with the shortest distance to the player.

```
# Check the Positions around the current Enemy Position,
# to see which has the shortest distance from the Player
self.moveSmartCheck(current.x + 1, current.y)
self.moveSmartCheck(current.x - 1, current.y)
self.moveSmartCheck(current.x, current.y + 1)
self.moveSmartCheck(current.x, current.y - 1)
```

After selecting the move we clean up our arrays by calling clear on both of them to empty all the elements they have stored. Then we want to check when the Enemy catches the player and draw the updated position to the screen.



```
# Empty our lists of Positions
self.toVisit.clear()
self.visited.clear()

# If the Enemy is colliding with the Player
if self.player.collision(self.x, self.y):
    # Set hitPlayer
    self.hitPlayer = True

# Update the new position on the screen
self.draw()
```

## TASK: TEST THE *ENEMY.MOVESMART* GAME FUNCTION



Let's now test our final game function `Enemy.moveSmart` by adding a new `Enemy` object to the `enemies` array. We can set the start position for this `Enemy` to be on the left side of the screen, far away from the player, where we have an empty space and its colour to purple.

```
enemies = [Enemy(6, 2, gamer.Screen.RED, walls, gems, player, screen, screenWidth, screenHeight)
            Enemy(5, 5, gamer.Screen.GREEN, walls, gems, player, screen, screenWidth, screenHeight)
            Enemy(10, 4, gamer.Screen.PURPLE, walls, gems, player, screen, screenWidth, screenHeight)]
```

Again, let's add the new `Enemy` object into the game loop underneath where we call `moveNormal` on the second `Enemy`. For all the other `Enemy` objects, we select them by adding an `else` statement to the bottom of our `if` statements in the `enemies` loop, and we want to call the new `moveSmart` function.

```
elif i == 1: enemies[i].moveNormal()
# Third Enemy finds a path to the Player
else: enemies[i].moveSmart()
```

Don't forget to reset the new `Enemy` in the `livesUpdate` function. Try testing the new smart `Enemy` and see if you can still win the A-Mazing Game!

```
enemies[1].reset(5, 5)
enemies[2].reset(10, 4)
```

# CONCLUSION

## LESSON 11 CONCLUSION

In this lesson, you have:

- Setup an array and 2D array
- Implemented a basic 2D search algorithm
- Implemented an insertion sort based sorting algorithm
- Used for loop, `if` statement and Boolean logic to check for interaction between `Enemy` and other objects



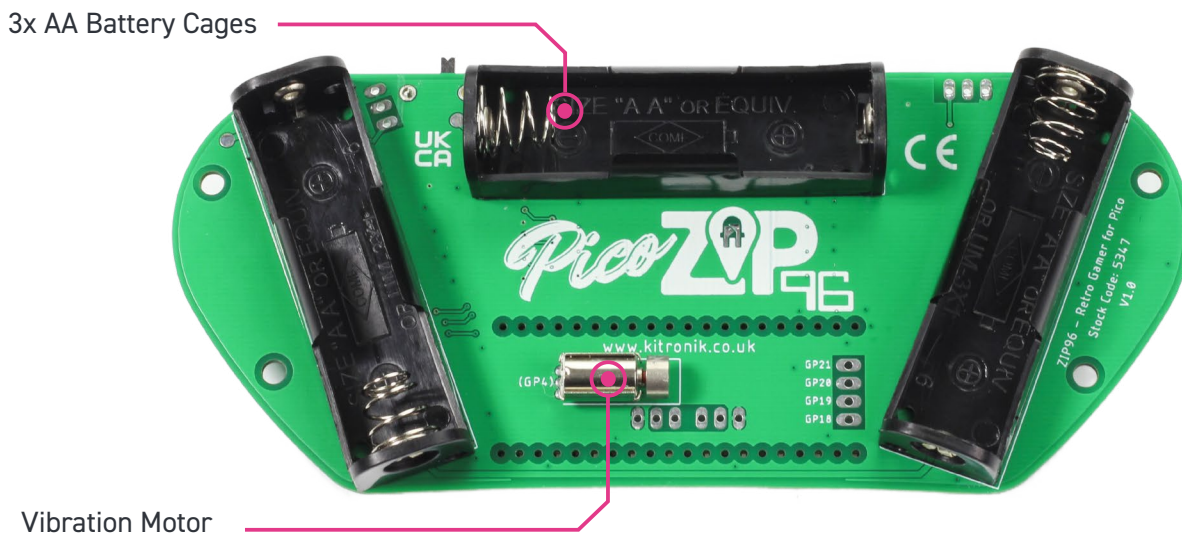
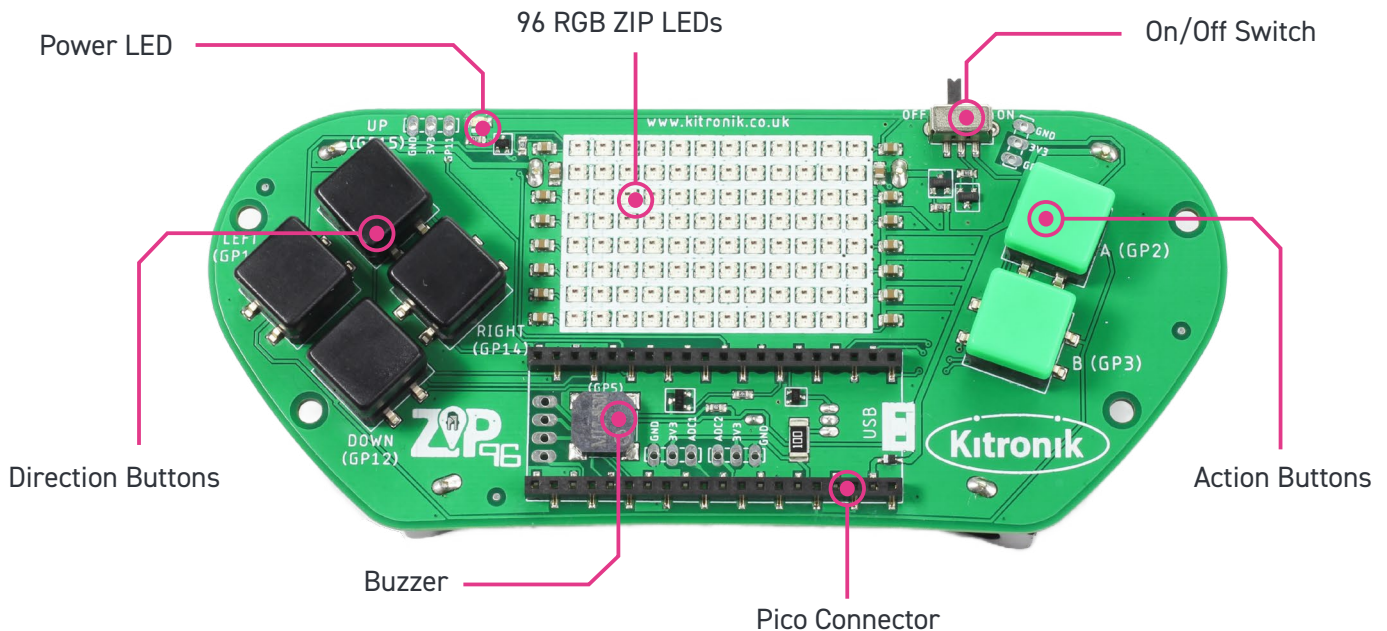
## FULL LESSON CODE

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](http://www.kitronik.co.uk)

E: [support@kitronik.co.uk](mailto: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](http://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.