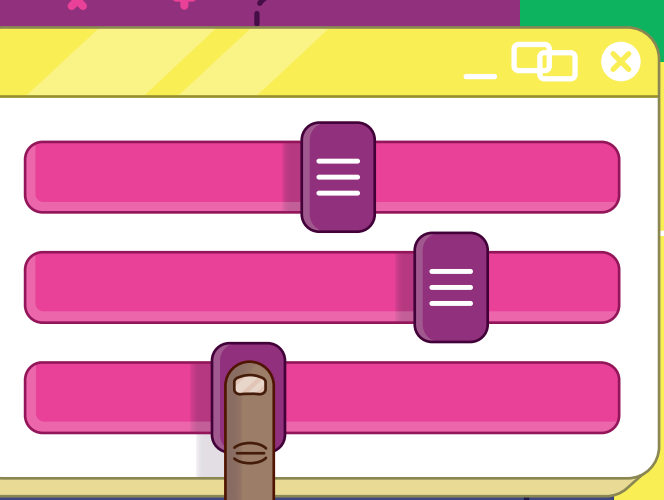
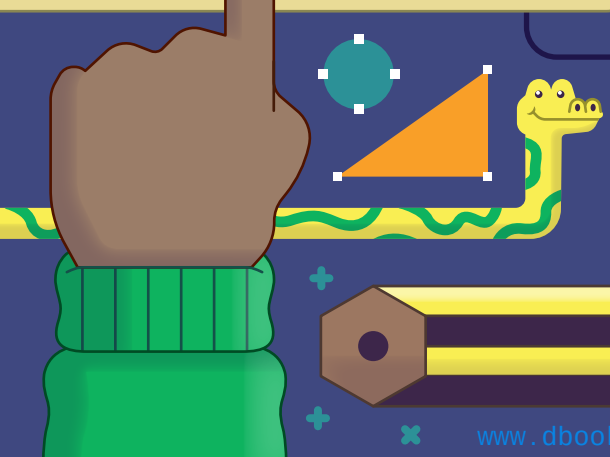
Laura Sach
Martin O'Hanlon

Create Graphical User Interfaces with Python

How to build windows, buttons, and widgets for your Python projects





Chapter 8

Flood It

Create a more complex Waffle-based puzzle game

'Flood It' is a game where the aim is to flood the board with all squares the same colour. Beginning with the top-left square, players choose a colour to flood into. It offers a slightly more complex Waffle-based game.

Aim of the game

In this example (**Figure 1**), the top-left square is blue. The player could either choose to flood into the single purple square below, or to flood into the yellow square to the right.

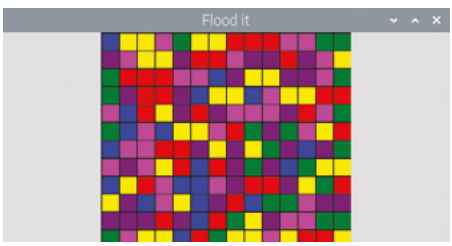


Figure 1 Flood the squares with one colour

Flooding the yellow square would be a better move because all adjoining yellow squares would also be flooded, and the player is only allowed a limited amount of moves before the game ends.

Set up

Download (from **magpi.cc/floodit**) and open the starter file, **floodit_starter.py**. Save it in a sensible place.

In the variables section, give the variables some values:

- `colours` – a list of six colours as strings. These can either be common colour names or hex colours. The colour names "white", "black", "red", "green", "blue", "cyan", "yellow", and "magenta" will always be available.
- `board_size` – the width/height of the board as an integer; we chose 14. The board is always a square.
- `moves_limit` – how many moves the player is allowed before they lose, as an integer; we chose 25.

In the app section, create an App widget and give it a title.

```
app = App("Flood it")
app.display()
```

Running this will result in a standard labelled window (**Figure 2**).

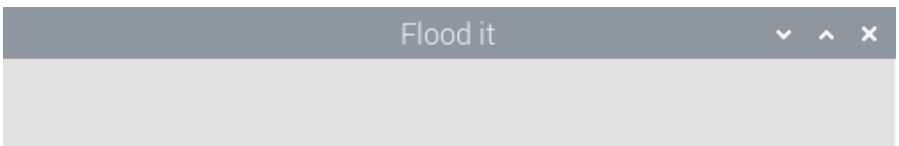


Figure 2 The usual labelled window

Create the board

The board is a grid of squares, each containing a randomly selected colour from the list you created earlier.

Inside the app, add a Waffle widget. This will create a grid which will be the board.

```
board = Waffle(app)
```

Figure 3 The grid squares are too small

Run your program and you will see that the grid is a bit too small (**Figure 3**).

Add to the line of code you just wrote to specify parameters for the width and height of the Waffle, and make the padding between the grid squares zero.

```
board = Waffle(app, width=board_size, height=board_size, pad=0)
```

That's better (**Figure 4**).

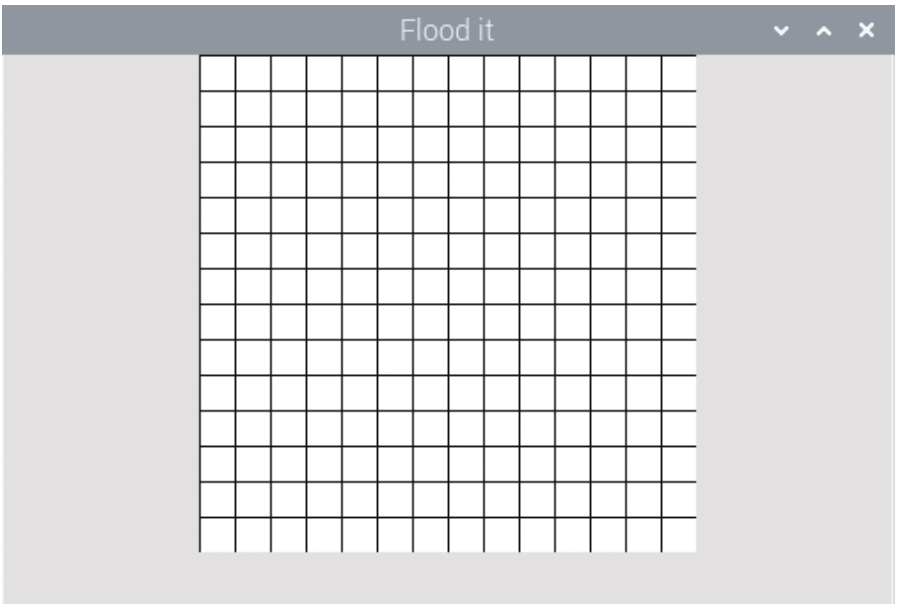


Figure 4 A grid of the correct board size, with no padding

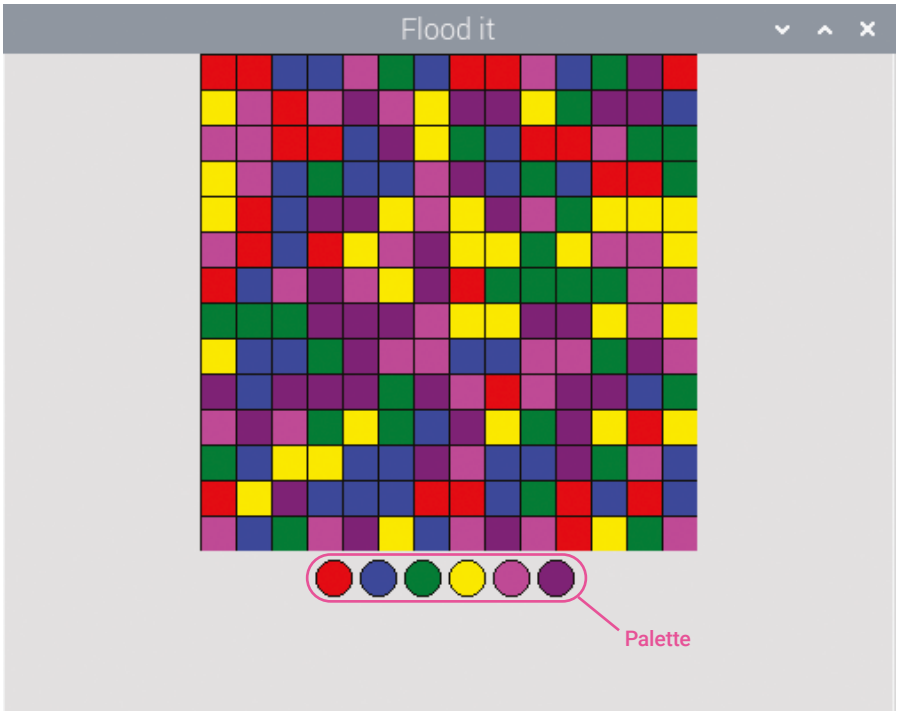


Figure 5 You'll need a palette for the player to choose a colour

Create the palette

The palette shows the player which colours they can click on to flood the board. They will click on these colours to play the game. The palette from the finished game is shown in **Figure 5**.

On the line after you created the board, create another Waffle, but this time it should be called `palette`.

```
palette = Waffle(app)
```

Remember when you added the parameters to the board Waffle in the previous step? This time, add these parameters to the `palette` Waffle with each one separated by a comma:

`width = 6` (the number of colours we have)
`height = 1`
`dotty = True` (this makes the squares into circles)

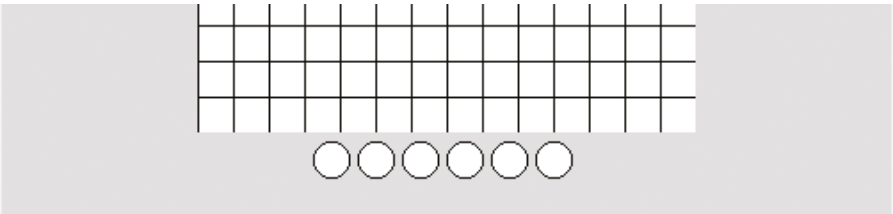


Figure 6 A blank palette

So, now you should have:

```
palette = Waffle(app, width=6, height=1, dotty=True)
```

Run the code to see a blank palette (**Figure 6**).

Colour in the board

The board should start off with each square as a randomly chosen colour from the colours list you created earlier.

On the line below your palette, write a call to a function

```
fill_board()
```

Find the functions section in your program, and begin writing the code for this new function:

```
def fill_board():
```

You can write a nested loop to loop through every row and column in the board. Each pixel will be coloured with a randomly chosen colour from the list. To colour in a pixel, you will use this code, where the `?` symbols will be replaced with the x, y co-ordinates of the pixel:

```
board.set_pixel(?, ?, random.choice(colours))
```

Try to write the code yourself using what you have learnt about nested loops in the previous chapters – the solution is provided on page 83 if you get stuck.

Hint: *Use the* `board_size` *variable to know how many times to loop.*

When you run your code, you should see a colourful board. If you see a white board, double-check that you put in the function call to `fill_board()` (**Figure 7**).

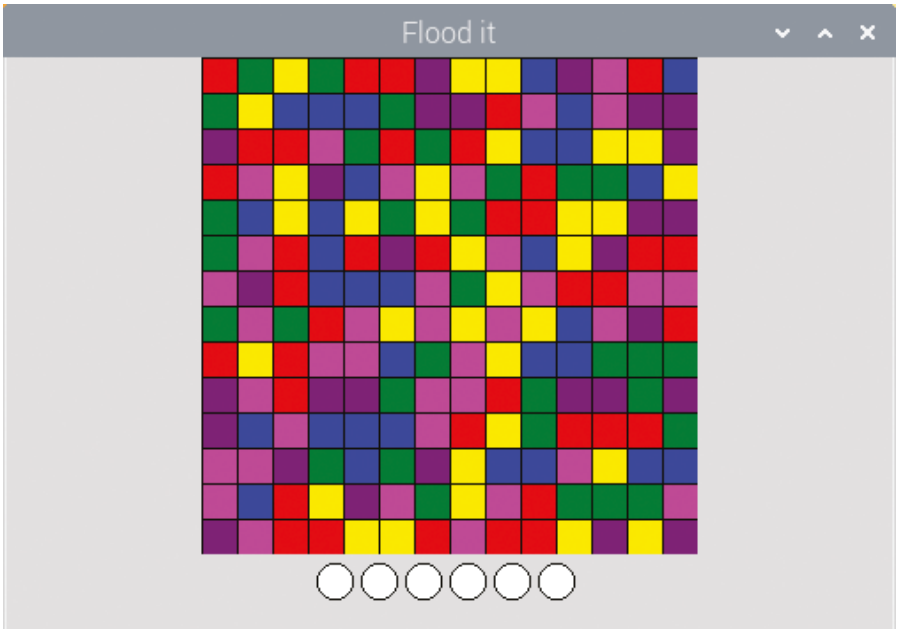


Figure 7 Each square of the board is coloured randomly

Here is one solution, but there are many ways you could do this:

```
def fill_board():
    for x in range(board_size):
        for y in range(board_size):
            board.set_pixel(x, y, random.choice(colours))
```

An alternative solution which uses a more advanced feature called a list comprehension:

```
def fill_board():
    [board.set_pixel(x, y, random.choice(colours)) for y in
range(board_size) for x in range(board_size)]
```

Colour in the palette

Now that you have a colourful board, let's colour in the palette.

On the line below your `fill_board()` code, write a call to a function:

```
init_palette()
```

Find the functions section in your program, and begin writing the code for this new function:

```
def init_palette():
```

The idea here is to loop through all of the colours in the list, assigning one to each of the circles in the palette. You can use the same `set_pixel` method as you used for the board to change the colour of the circles in the palette.

Have a go at writing the code yourself. If you get stuck, some possible solutions are shown in the 'How many ways can you colour the palette' box.

Hint: *All of the circles in the palette are in row 0 of the Waffle.*

HOW MANY WAYS CAN YOU COLOUR THE PALETTE?

Here is a solution which uses a loop and a variable to keep track of which column you are colouring in:

```
def init_palette():
    column = 0
    for colour in colours:
        palette.set_pixel(column, 0, colour)
        column += 1
```

Here is a similar solution which uses a range inside the `for` loop instead of a counter variable:

```
def init_palette():
    for x in range(len(colours)):
        palette.set_pixel(x, 0, colours[x])
```

Here is a different solution which uses the index function `colours.index(colour)`. This code says 'In the colours list, find me the position in the list of colour'. So, for example if your list was `["green", "blue", "red"]` then the index of green would be 0, the index of blue would be 1, etc., remembering that we count starting from zero.

```
def init_palette():
    for colour in colours:
        palette.set_pixel(colours.index(colour), 0, colour)
```

You can use any of these solutions, or you may have come up with a different way by yourself. None of them is the 'right answer': there are often many different ways of coding a solution.

Start the flood

When the player clicks on a colour in the palette, the board should flood with that colour, beginning with the top-left square.

In the functions section, create a new function called `start_flood` in exactly the same way as you did for the last two functions. This function needs to take two parameters which will be the x, y co-ordinates of the square that was clicked on. Add these between the brackets so that you end up with your code looking like this:

```
def start_flood(x, y):
```

Add a line of code (indented) to the function get the name of the colour that was clicked on:

```
    flood_colour = palette.get_pixel(x,y)
```

This will be the colour to flood the board with.

Add a line of code to get the current colour of the starting pixel – this is always the pixel in the top left of the board, at co-ordinates 0, 0.

```
    target = board.get_pixel(0,0)
```

Now call the `flood` function, which has already been written for you in the starter file. This function starts at 0,0 and floods all the pixels connected to the top-left pixel that are the same colour with the `flood_colour`.

```
    flood(0, 0, target, flood_colour)
```

This function should run whenever someone clicks on a colour in the palette, so find the line of code where you created the palette.

```
palette = Waffle(app, width=6, height=1, dotty=True)
```

Add another parameter which is a command. When a circle on the palette is clicked, this command will be executed. The command is the function `start_flood`, so your code should now look like this:

```
palette = Waffle(app, width=6, height=1, dotty=True,
command=start_flood)
```

Test out your code by clicking on the circles on the palette.

The top-left square is green (**Figure 8**). If you click purple on the palette, the top-left square will turn purple and connect to the purple square below (**Figure 9**).

Now there are five purple squares connected to the top-left square. Let's click pink to connect up the pink squares underneath (**Figure 10**).

Now there is a large chain of pink squares. Continue the game by pressing different colours in the palette. The aim is to eventually get all of the squares the same colour.

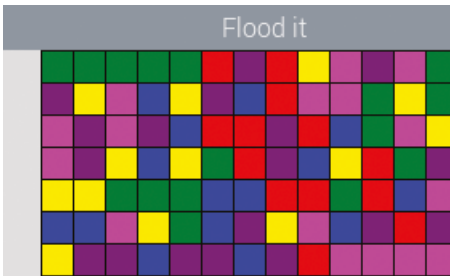


Figure 8 Here, the top-left square is green

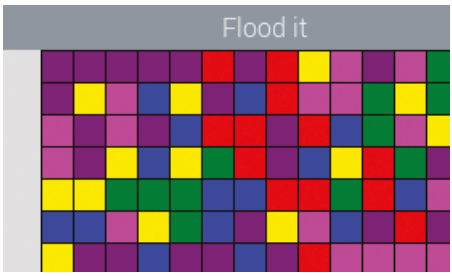


Figure 9 Clicking purple turns it purple

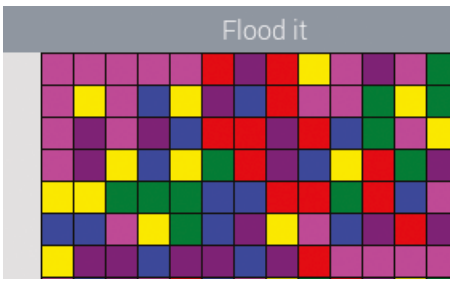


Figure 10 Click pink for a chain of pink

Winning the game

At the moment, if the player manages to get all of the squares in the grid the same colour, nothing happens. The player is also allowed an infinite number of turns, as the number of moves they have taken is not tracked.

First let's add a piece of text to the GUI to display whether the player has won or lost. The text will start off blank.

Underneath the code for the palette, add a Text widget called `win_text`.

```
win_text = Text(app)
```

In the variables section, add another variable called `moves_taken` and set it to 0.

Now create a function called `win_check` to check after each move whether the player has won.

First, you need to specify that you would like to be allowed to change the value of the global variable `moves_taken`.

```
global moves_taken
```

Then add 1 to the `moves_taken` variable – each time this function is called, we will add one more move.

```
    moves_taken += 1
```

Check if the `moves_taken` is less than the `moves_limit` or not:

```
    if moves_taken < moves_limit:

    else:
```

If the `moves_taken` is not within the limit, this means the playher has run out of moves, so update the text to say that they lost:

```
    if moves_taken < moves_limit:

    else:
        win_text.value = "You lost :("
```

If the number of moves taken *is* less than the limit, check whether all of the squares are the same colour by calling the function already written for you in the starter file. Make sure the following code is indented below the first `if` statement:

```
      if all_squares_are_the_same():
          win_text.value = "You win!"
```

The completed piece of code should look like this:

```
def win_check():
    moves_taken += 1
    if moves_taken <= moves_limit:
        if all_squares_are_the_same():
            win_text.value = "You win!"
    else:
        win_text.value = "You lost :("
```

Finally, you must call the `win_check` function whenever a square is clicked on. The easiest way to do this is to add the function call at the end of the `start_flood` function.

Now it's time to test the game. An example code listing is shown in **08-floodit.py**, overleaf.

Test your game

You can test whether the game works by playing it; however, it might take a long time to test whether you can win! An easier way to check is to change the `board_size` variable to something small such as 5, and then play the game on a much smaller grid to see whether you can win.

You can easily test whether the game causes you to lose properly by clicking on the same colour 25 times!

CHALLENGE

- If the player wins or the player loses, disable the palette to prevent them clicking on it any more and causing an error. The code to disable the palette is `palette.disable()`.
- Display how many moves are left as a piece of text on the GUI.
- Add a button which displays instructions for how to play.
- Add a reset button to let the player start a new game. Don't forget, you will also have to reset the colours on the board, reset the `moves_taken` variable, and re-enable the palette if you disabled it (`palette.enable()`).

08-floodit.py / Python 3

DOWNLOAD
magpi.cc/guizerocode

```
# ----------------------------
# Imports
# ----------------------------

from guizero import App, Waffle, Text, PushButton, info
import random

# ----------------------------
# Variables
# ----------------------------

colours = ["red", "blue", "green", "yellow", "magenta", "purple"]
board_size = 14
moves_limit = 25
moves_taken = 0

# ----------------------------
# Functions
# ----------------------------

# Recursively floods adjacent squares
def flood(x, y, target, replacement):
    # Algorithm from https://en.wikipedia.org/wiki/Flood_fill
    if target == replacement:
        return False
    if board.get_pixel(x, y) != target:
        return False
    board.set_pixel(x, y, replacement)
    if y+1 <= board_size-1:   # South
        flood(x, y+1, target, replacement)
    if y-1 >= 0:              # North
        flood(x, y-1, target, replacement)
    if x+1 <= board_size-1:    # East
        flood(x+1, y, target, replacement)
    if x-1 >= 0:              # West
        flood(x-1, y, target, replacement)

# Check whether all squares are the same
def all_squares_are_the_same():
    squares = board.get_all()
    if all(colour == squares[0] for colour in squares):
        return True
```

08-floodit.py (cont.) / Python 3

```
    else:
        return False

def win_check():
    global moves_taken
    moves_taken += 1
    if moves_taken <= moves_limit:
        if all_squares_are_the_same():
            win_text.value = "You win!"
    else:
        win_text.value = "You lost :("

def fill_board():
    for x in range(board_size):
        for y in range(board_size):
            board.set_pixel(x, y, random.choice(colours))

def init_palette():
    for colour in colours:
        palette.set_pixel(colours.index(colour), 0, colour)

def start_flood(x, y):
    flood_colour = palette.get_pixel(x,y)
    target = board.get_pixel(0,0)
    flood(0, 0, target, flood_colour)
    win_check()

# ------------------------------
# App
# ------------------------------

app = App("Flood it")

board = Waffle(app, width=board_size, height=board_size, pad=0)
palette = Waffle(app, width=6, height=1, dotty=True,
command=start_flood)

win_text = Text(app)

fill_board()
init_palette()

app.display()
```

Chapter 9

Emoji Match

Create a fun picture-matching game

You are going to build an emoji picture-matching game (Figure 1). The object of the game is to spot the one emoji that appears in two different sets. You get a point for each correct match and lose a point for an incorrect match.

Loading emojis

To create the game, you will need emojis. You can use the emojis created for Twitter (**twemoji.twitter.com**). Download the **emojis.zip** file from **magpi.cc/guizeroemojis**, open the zip file, and copy the **emojis** folder to the folder where you save your code.

The game will need to choose nine emojis at random and arrange them into a grid. A simple way to do this is to put all of the emojis into a list and randomly shuffle them.

The following code creates a shuffled list of items, each in the form **path/emoji_file_name**.

Create a new program with the usual commented lines for different sections (Imports, Variables, Functions, App). Under imports, add:

```
import os
from random import shuffle
```

Then, under variables, enter this code which creates a shuffled list of emojis, each in the form **path/emoji_file_name**.

```
# set the path to the emoji folder on your computer
emojis_dir = "emojis"
emojis = [os.path.join(emojis_dir, f) for f in os.listdir(emojis_
dir)]
shuffle(emojis)
```

The `emojis_dir` variable is the path of the emojis on your computer; it will tell the code that loads the emojis where to find them.

Test your program. Try printing the `emojis` list to the screen with `print(emojis)`. You should see a long list of file names. The list should be in a different order each time you run it.

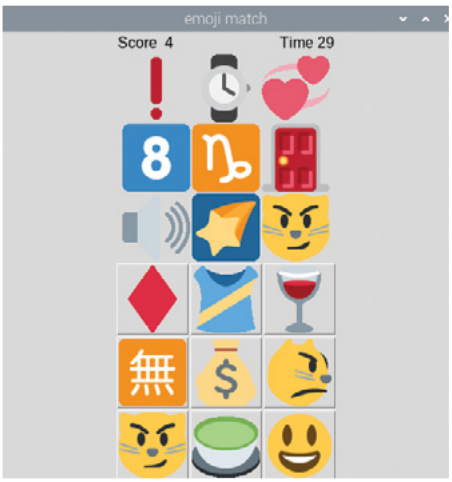


Figure 1 The finished game

Displaying the emojis

Next, the code needs to create two 3×3 grids of Picture and PushButton widgets which will show the emojis.

Modify your program to create a guizero app and a Box to hold the picture widgets using a `"grid"` layout. In the imports section, add this line to import the required widgets:

```
from guizero import App, Box
```

In the app section, add the following code:

```
app = App("emoji match")

pictures_box = Box(app, layout="grid")
```

The Box widget is really useful for laying out your GUI. It's an invisible area of your GUI where you can group widgets together. A Box can have its own layout, size, and bg

(background). They can also be hidden or shown, meaning you can easily make a collection of widgets invisible.

If you wish to *see* the Box, you can add a border by setting the parameter to True.

```
pictures_box = Box(app, layout="grid", border=True)
```

Now, add the Picture widget to your imports:

```
from guizero import App, Box, Picture
```

In the app section, add in the code to create the Picture widgets and add them to a list.

```
pictures = []

for x in range(0,3):
    for y in range(0,3):
        picture = Picture(pictures_box, grid=[x,y])
        pictures.append(picture)
```

To assign co-ordinates to each Picture widget, two `for` loops are used. They both run through the range 0–2; one assigns its value to the variable `x` and the other to the variable `y`. The grid position of each widget is set using the `x` and `y` values. The widgets are appended to a list so they can be referenced later in the game.

Do the same for PushButton widgets to create the second 3×3 grid. First, add the widget to your imports:

```
from guizero import App, Box, Picture, PushButton
```

In the app section, add lines so it looks like this:

```
app = App("emoji match")

pictures_box = Box(app, layout="grid")
buttons_box = Box(app, layout="grid")

pictures = []
buttons = []

for x in range(0,3):
    for y in range(0,3):
```

```
        picture = Picture(pictures_box, grid=[x,y])
        pictures.append(picture)

        button = PushButton(buttons_box, grid=[x,y])
        buttons.append(button)
```

In the functions section, create a function to set up each round of the game.

```
def setup_round():
    for picture in pictures:
        picture.image = emojis.pop()

    for button in buttons:
        button.image = emojis.pop()
```

To assign each `picture` and `button` widget an emoji, the `image` property is set to an item from the `emojis` list. Emojis are selected using `pop()`, which chooses the last item in a list and then removes it from the list. I've used this function because it will prevent any emoji appearing in the game more than once.

At the bottom of your program, call the `setup_round` function and display the app.

```
setup_round()

app.display()
```

Your program should now resemble **emoji1.py** (page 99). Test it and you should see two grids of nine emojis.

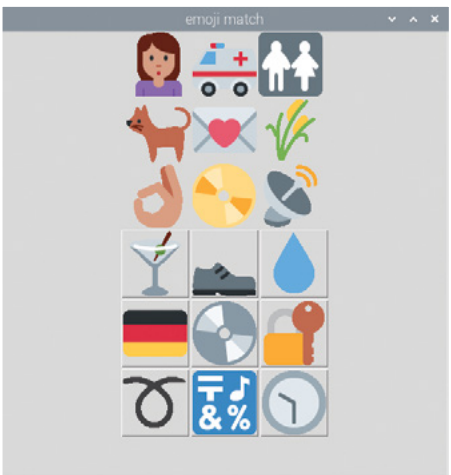


Figure 2 No matching emojis

Matching emojis

At the moment, all of the emojis in your app will be different (**Figure 2**). In the next step, you will pick another emoji to match, and update one picture and one button so they have the same matching emoji.

Add `randint` to your `random` import line. This is used to obtain a number from 0 to 8 for each picture and button.

```
from random import shuffle, randint
```

Then add this code (indented) to the bottom of the `setup_round` function to pop another emoji from the list and set it to be the image of a random picture and button.

```
matched_emoji = emojis.pop()

random_picture = randint(0,8)
pictures[random_picture].image = matched_emoji

random_button = randint(0,8)
buttons[random_button].image = matched_emoji
```

Your code should now look like **emoji2.py**. Run your program now; one of the emojis should match. Look carefully – the matching emoji can be hard to spot.

Check the guess

Each time one of the PushButtons is pressed, it will need to check if this is the matching emoji and put the result 'correct' or 'incorrect' on the screen. After the player's guess, a new round will be set up and different set of emojis displayed.

Your app will need a Text widget display the result. Add it to your imports:

```
from guizero import App, Box, Picture, PushButton, Text
```

Add this line in your app section:

```
result = Text(app)
```

Create a new function which will be called when one of the emoji buttons is pressed. It will display 'correct' or 'incorrect' and call `setup_round` to create the next set of emojis.

```
def match_emoji(matched):
    if matched:
        result.value = "correct"
    else:
        result.value = "incorrect"

    setup_round()
```

The incorrect emoji buttons will pass False to the `match_emoji` function; the matching emoji will pass True.

Update the `setup_round` function so that all the 'incorrect' buttons call the `match_emoji` function.

```
    for button in buttons:
        button.image = emojis.pop()
        button.update_command(match_emoji, args=[False])
```

The `update_command` method sets the function which will be called when the button is pressed. The `args` list `[False]` will be used as the parameters to the `match_emoji` function.

Finally, update the command for the matching button so it calls `match_emoji`, but this time passes True as the argument.

```
    buttons[random_button].update_command(match_emoji, [True])
```

Your code should now resemble **emoji3.py**. Play the game. In each round there will be a matching emoji – press the matching picture button. Did you get it right?

Adding a score and timer

At the moment, the game continues forever (or until you run out of emojis in the list). Add a score and a timer which counts down to the end of the game to give a challenge.

In the app section, create two Text widgets to show the score and the timer.

```
score = Text(app, text="0")
timer = Text(app, text="30")
```

The timer is set to `"30"`, which will be the number of seconds in each round.

Modify the `match_emoji` function to either add or subtract 1 to/from the player's score.

```
def match_emoji(matched):
    if matched:
        result.value = "correct"
        score.value = int(score.value) + 1
    else:
        result.value = "incorrect"
        score.value = int(score.value) - 1
```

To create the timer, you will use a feature of guizero which allows you to ask the application to continuously call a function every 1 second.

Create a function which will reduce the value of the timer by 1.

```
def reduce_time():
    timer.value = int(timer.value) - 1
```

Before the app is displayed, use the `app.repeat()` function to call the `reduce_time` function every second (1000 milliseconds).

```
app.repeat(1000, reduce_time)

app.display()
```

Running your game now, you will notice that the timer counts down from 30. Unfortunately, it will continue counting down past 0 and never stop.

Update the `reduce_time` function to check if the timer is less than zero and then stop the game.

```
def reduce_time():
    timer.value = int(timer.value) - 1
    # is it game over?
    if int(timer.value) < 0:
        result.value = "Game over! Score = " + score.value
        # hide the game
        pictures_box.hide()
        buttons_box.hide()
        timer.hide()
        score.hide()
```

When the timer is less than 0, the message 'game over' is displayed and the game's widgets are hidden so the user can no longer play.

See **emoji4.py** to get an idea of how your code should now look. Run it and play the emoji match game. Challenge a friend or family member to a game.

You may want to put the score and timer widgets into a Box so they can be laid out better (**Figure 3**) – see the complete **09-emoji-match.py** listing for how to do this.

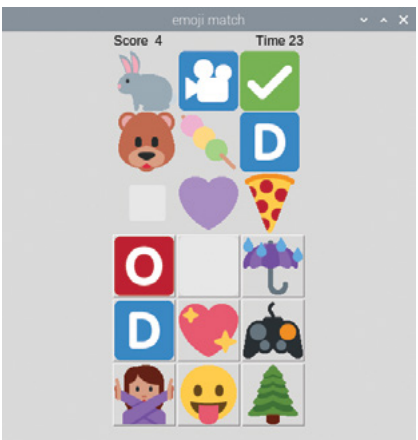


Figure 3 With box for score and timer

CHALLENGE

At the moment, the only way to start a new round of the game is restart the program. Can you change the code to introduce a button to start a new round?

USING OTHER IMAGES

The emoji match game uses picture buttons to allow the user to pick which emoji matches. You can make any PushButton widget into a picture button by setting the image parameter; for example:

```
button = PushButton(app, image="my_picture.gif")
```

The button will scale to fit the size of your image. The type of image you can use is determined by your operating system and how you installed guizero, although any setup will support GIF images. To find the image file types supported by your setup, you can run:

```
from guizero import system_config
print(system_config.supported_image_types)
```

You can find out more about image support in guizero at **lawsie.github.io/guizero/images**.

emoji1.py / Python 3

DOWNLOAD
magpi.cc/guizerocode

```
# ------------------------------
# Imports
# ------------------------------

import os
from random import shuffle
from guizero import App, Box, Picture, PushButton


# ------------------------------
# Variables
# ------------------------------

# set the path to the emoji folder on your computer
emojis_dir = "emojis"
```

emoji1.py (cont.) / Python 3

```
emojis = [os.path.join(emojis_dir, f) for f in os.listdir(emojis_
dir)]
shuffle(emojis)


# ------------------------------
# Functions
# ------------------------------

def setup_round():
    for picture in pictures:
        picture.image = emojis.pop()

    for button in buttons:
        button.image = emojis.pop()


# ------------------------------
# App
# ------------------------------

app = App("emoji match")

pictures_box = Box(app, layout="grid")
buttons_box = Box(app, layout="grid")

pictures = []
buttons = []

for x in range(0,3):
    for y in range(0,3):
        picture = Picture(pictures_box, grid=[x,y])
        pictures.append(picture)

        button = PushButton(buttons_box, grid=[x,y])
        buttons.append(button)

setup_round()

app.display()
```

emoji2.py / Python 3

```
# ------------------------------
# Imports
# ------------------------------

import os
from random import shuffle, randint
from guizero import App, Box, Picture, PushButton


# ------------------------------
# Variables
# ------------------------------

# set the path to the emoji folder on your computer
emojis_dir = "emojis"
emojis = [os.path.join(emojis_dir, f) for f in os.listdir(emojis_
dir)]
shuffle(emojis)


# ------------------------------
# Functions
# ------------------------------

def setup_round():
    for picture in pictures:
        picture.image = emojis.pop()

    for button in buttons:
        button.image = emojis.pop()

    matched_emoji = emojis.pop()

    random_picture = randint(0,8)
    pictures[random_picture].image = matched_emoji

    random_button = randint(0,8)
    buttons[random_button].image = matched_emoji
```

emoji2.py (cont.) / Python 3

```
# ------------------------------
# App
# ------------------------------

app = App("emoji match")

pictures_box = Box(app, layout="grid")
buttons_box = Box(app, layout="grid")

pictures = []
buttons = []

for x in range(0,3):
    for y in range(0,3):
        picture = Picture(pictures_box, grid=[x,y])
        pictures.append(picture)

        button = PushButton(buttons_box, grid=[x,y])
        buttons.append(button)

setup_round()

app.display()
```

emoji3.py / Python 3

```
# ------------------------------
# Imports
# ------------------------------

import os
from random import shuffle, randint
from guizero import App, Box, Picture, PushButton, Text


# ------------------------------
# Variables
# ------------------------------

# set the path to the emoji folder on your computer
```

emoji3.py (cont.) / Python 3

```
emojis_dir = "emojis"
emojis = [os.path.join(emojis_dir, f) for f in os.listdir(emojis_
dir)]
shuffle(emojis)


# ------------------------------
# Functions
# ------------------------------

def setup_round():
    for picture in pictures:
        picture.image = emojis.pop()

    for button in buttons:
        button.image = emojis.pop()
        button.update_command(match_emoji, args=[False])

    matched_emoji = emojis.pop()

    random_picture = randint(0,8)
    pictures[random_picture].image = matched_emoji

    random_button = randint(0,8)
    buttons[random_button].image = matched_emoji

    buttons[random_button].update_command(match_emoji, [True])

def match_emoji(matched):
    if matched:
        result.value = "correct"
    else:
        result.value = "incorrect"

    setup_round()


# ------------------------------
# App
# ------------------------------
```

emoji3.py (cont.) / Python 3

```
app = App("emoji match")

pictures_box = Box(app, layout="grid")
buttons_box = Box(app, layout="grid")

pictures = []
buttons = []

for x in range(0,3):
    for y in range(0,3):
        picture = Picture(pictures_box, grid=[x,y])
        pictures.append(picture)

        button = PushButton(buttons_box, grid=[x,y])
        buttons.append(button)

result = Text(app)

setup_round()

app.display()
```

emoji4.py / Python 3

```
# ------------------------------
# Imports
# ------------------------------

import os
from random import shuffle, randint
from guizero import App, Box, Picture, PushButton, Text


# ------------------------------
# Variables
# ------------------------------

# set the path to the emoji folder on your computer
emojis_dir = "emojis"
emojis = [os.path.join(emojis_dir, f) for f in os.listdir(emojis_
```

emoji4.py (cont.) / Python 3

```
dir)]
shuffle(emojis)


# ----------------------------
# Functions
# ----------------------------

def setup_round():
    for picture in pictures:
        picture.image = emojis.pop()

    for button in buttons:
        button.image = emojis.pop()
        button.update_command(match_emoji, args=[False])

    matched_emoji = emojis.pop()

    random_picture = randint(0,8)
    pictures[random_picture].image = matched_emoji

    random_button = randint(0,8)
    buttons[random_button].image = matched_emoji

    buttons[random_button].update_command(match_emoji, [True])

def match_emoji(matched):
    if matched:
        result.value = "correct"
        score.value = int(score.value) + 1
    else:
        result.value = "incorrect"
        score.value = int(score.value) - 1

    setup_round()

def reduce_time():
    timer.value = int(timer.value) - 1
    # is it game over?
    if int(timer.value) < 0:
        result.value = "Game over! Score = " + score.value
```

emoji4.py (cont.) / Python 3

```
        # hide the game
        pictures_box.hide()
        buttons_box.hide()
        timer.hide()
        score.hide()


# ------------------------------
# App
# ------------------------------

app = App("emoji match")

score = Text(app, text="0")
timer = Text(app, text="30")

pictures_box = Box(app, layout="grid")
buttons_box = Box(app, layout="grid")

pictures = []
buttons = []

for x in range(0,3):
    for y in range(0,3):
        picture = Picture(pictures_box, grid=[x,y])
        pictures.append(picture)

        button = PushButton(buttons_box, grid=[x,y])
        buttons.append(button)

result = Text(app)

setup_round()

app.repeat(1000, reduce_time)

app.display()
```

09-emoji-match.py / Python 3

```
# ------------------------------
# Imports
# ------------------------------

import os
from random import shuffle, randint
from guizero import App, Box, Picture,  PushButton, Text


# ------------------------------
# Variables
# ------------------------------

# set the path to the emoji folder on your computer
emojis_dir = "emojis"
emojis = [os.path.join(emojis_dir, f) for f in os.listdir(emojis_
dir)]
shuffle(emojis)


# ------------------------------
# Functions
# ------------------------------

def setup_round():
    for picture in pictures:
        picture.image = emojis.pop()

    for button in buttons:
        button.image = emojis.pop()
        button.update_command(match_emoji, args=[False])

    matched_emoji = emojis.pop()

    random_picture = randint(0,8)
    pictures[random_picture].image = matched_emoji

    random_button = randint(0,8)
    buttons[random_button].image = matched_emoji

    buttons[random_button].update_command(match_emoji, [True])
```

09-emoji-match.py (cont.) / Python 3

```
def match_emoji(matched):
    if matched:
        result.value = "correct"
        score.value = int(score.value) + 1
    else:
        result.value = "incorrect"
        score.value = int(score.value) - 1

    setup_round()

def reduce_time():
    timer.value = int(timer.value) - 1
    # is it game over?
    if int(timer.value) < 0:
        result.value = "Game over! Score = " + score.value
        # hide the game
        game_box.hide()


# ------------------------------
# App
# ------------------------------

app = App("emoji match")

game_box = Box(app, align="top")

top_box = Box(game_box, align="top", width="fill")
Text(top_box, align="left", text="Score ")
score = Text(top_box, text="4", align="left")
timer = Text(top_box, text="30", align="right")
Text(top_box, text="Time", align="right")

pictures_box = Box(game_box, layout="grid")
buttons_box = Box(game_box, layout="grid")

pictures = []
buttons = []

for x in range(0,3):
```

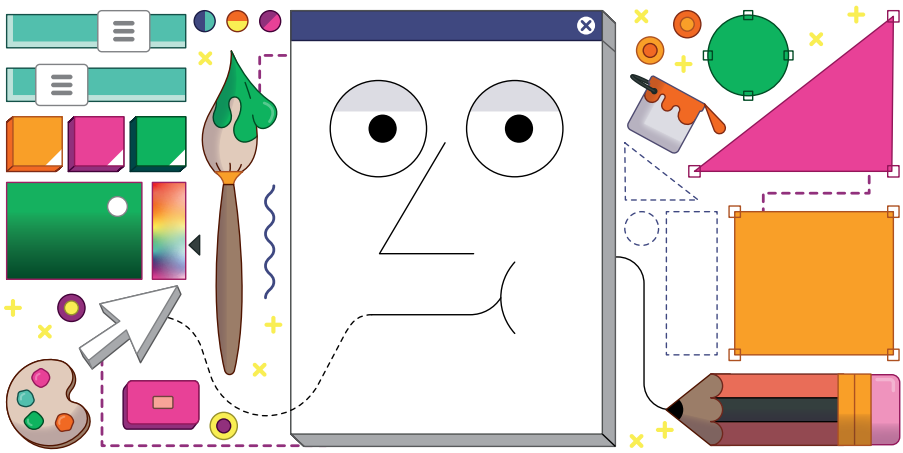
09-emoji-match.py (cont.) / Python 3

```
    for y in range(0,3):
        picture = Picture(pictures_box, grid=[x,y])
        pictures.append(picture)

        button = PushButton(buttons_box, grid=[x,y])
        buttons.append(button)

result = Text(app)

setup_round()

app.repeat(1000, reduce_time)

app.display()
```

Chapter 10

Paint

Create a simple drawing application



You are going to build a simple application which will allow you to paint using lines and shapes (Figure 1). You will create your paint application in four stages:

- drawing dots which follow the mouse
- draw lines between the dots
- adding colours and line width modifier
- drawing shapes

Note that you can style your application anyway you want – it doesn't have to look like this one.

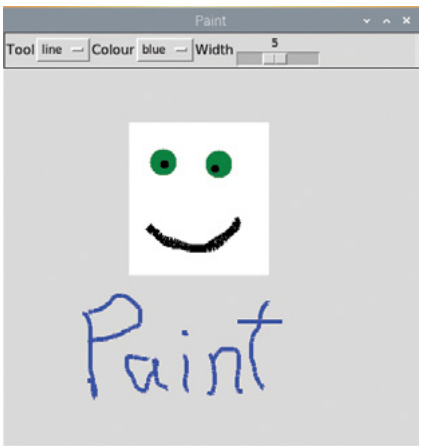


Figure 1 Our simple paint application

Drawing dots

The first step is to create a simple application which will use the Drawing widget and the `when_mouse_dragged` event to draw dots (or ovals on the screen).

In the imports section of your otherwise blank program, add the widgets:

```
from guizero import App, Drawing
```

Create a new function:

```
def draw(event):
    painting.oval(
        event.x - 1, event.y - 1,
        event.x + 1, event.y + 1,
        color="black")
```

Add this code to the app section:

```
app = App("Paint")

painting = Drawing(app, width="fill", height="fill")

painting.when_mouse_dragged = draw

app.display()
```

Your code should resemble **paint1.py** (page 116). The Drawing widget fills all the available space on the window. When the mouse is dragged across the drawing, the function `draw` is called which draws ovals on the `painting`.

The `draw` function is called each time an event is raised. The event which contains the x and y position of the mouse is passed as a variable to the function.

There is a problem, though. Unless you move your mouse very slowly, a series of dots is drawn by your program, not a continuous line (**Figure 2**). It's not a very good paintbrush! There are gaps between the dots because an event is not raised for every pixel the mouse crosses.

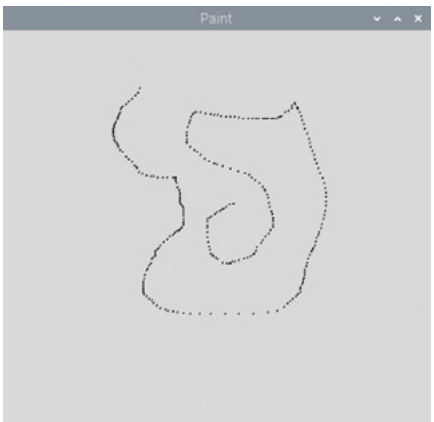


Figure 2 Not a very good paintbrush

Lines between the dots

To solve this problem, you are going to change the program to draw lines between the dots. That way the line made will be continuous and be more like a pen or paintbrush.

You will need to use a `when_left_button_pressed` event to store the position of where the line starts. Then draw a straight line between where the line starts and next position the mouse was dragged too.

Create a new function which will be called when the mouse is pressed.

```
    def start(event):
        painting.last_event = event
```

Add this to the app section:

```
    painting.when_left_button_pressed = start
```

The position of where the line starts is stored in the `last_event` variable.

Modify the `draw` function to draw a line between where the line starts and where the mouse has been dragged to.

```
    def draw(event):
        painting.line(
            painting.last_event.x, painting.last_event.y,
            event.x, event.y,
            color="black",
            width=3
        )

        painting.last_event = event
```

By updating the `last_event` variable to be the current position of the mouse, the next time the mouse is dragged, it will draw another line between this point and the next. Your program should look like **paint2.py**. Test it and make sure your paintbrush now works properly.

Change the line width and colour

You only have one colour and thickness for your paintbrush, which limits the drawing you can create. Next, you will amend your GUI so you can pick different colours and line widths.

Add two widgets to the GUI capture a colour and width for the line.

```
from guizero import App, Drawing, Combo, Slider
```

Add these line to the app section:

```
color = Combo(app, options=["black", "white", "red", "green", "blue"])
width = Slider(app, start=1, end=10)
```

You may also want to change the background colour of your painting to be different. Also, we have used a Combo and a Slider, but you could choose different widgets.

Modify the `draw` function to use the colour and width values when painting the line.

```
    painting.line(
        painting.last_event.x, painting.last_event.y,
        event.x, event.y,
        color=color.value,
        width=width.value
    )
```

Test your code, which should be like **paint3.py**, and you can now select the colour and line width.

Drawing shapes

You are going to extend your paint application so you can draw filled rectangles. When the mouse is pressed, the rectangle will appear and grow as the mouse is dragged across the screen. When the mouse button is released, the rectangle will drawn onto the screen.

To do this, you will modify your program to continuously draw and delete rectangles until the mouse button is released. Let's add a widget to your GUI so you can select whether to draw a line or a rectangle. Add this to the app section:

```
shape = Combo(app, options=["line", "rectangle"])
```

Modify the `draw` function to only draw lines if the `"line"` option is selected.

```
    if shape.value == "line":
        painting.line(
            painting.last_event.x, painting.last_event.y,
            event.x, event.y,
            color=color.value,
            width=width.value
        )
```

Test your program to make sure that the line still works and nothing happens when `"rectangle"` is selected.

Create two new variables to keep track of the first event and the last shape drawn when the mouse button is pressed.

```
def start(event):
    painting.last_event = event
    painting.first_event = event
    painting.last_shape = None
```

These variables will be used when drawing and deleting the rectangle before the mouse button is released.

Add this code to your draw function to draw a rectangle when the mouse is dragged.

```
    if shape.value == "rectangle":

        if painting.last_shape is not None:
            painting.delete(painting.last_shape)

        rectangle = painting.rectangle(
            painting.first_event.x, painting.first_event.y,
            event.x, event.y,
            color=color.value
        )

        painting.last_shape = rectangle
```

The program will continually draw a rectangle, then delete it, then draw it again until you release the button.

Your complete program should look similar to **paint4.py**. Have fun trying it out – what pictures can you create?

ADD OVALS

When you first created the Paint application, you used ovals to draw dots across the screen. Can you modify your program to draw ovals again, using a similar process to how rectangles are drawn? Hint: see the **10-paint.py** listing, which also styles up the tools and aligns them neatly in a box.

The Drawing widget also supports drawing triangles and polygons. Take a look at the documentation (**lawsie.github.io/guizero/drawing**) and see how you might use this function to create other shapes.

CUSTOM EVENTS

To get your paint application to react to the mouse position, you have used custom events. The events work very similar to the normal widget command parameter in that you set them to a function, which is called when that event occurs.

When your function is called, a variable is passed which contains information about the event that has occurred, such as the x and y co-ordinates of the mouse. Most widgets, including the App itself, support the following events:

- when clicked – **`when_clicked`**
- when the left mouse button is pressed – **`when_left_button_pressed`**
- when the left mouse button is released – **`when_left_button_released`**
- when the right mouse button is pressed – **`when_right_button_pressed`**
- when the right mouse button is released – **`when_right_button_released`**
- when a key is pressed – **`when_key_pressed`**
- when a key is released – **`when_key_released`**
- when the mouse enters a widget – **`when_mouse_enters`**
- when the mouse leaves a widget – **`when_mouse_leaves`**
- when the mouse is dragged across a widget – **`when_mouse_dragged`**

These events can be used to make your GUIs more interactive.

paint1.py / Python 3

DOWNLOAD
magpi.cc/guizerocode

```
# simple paint app, just draw dots

# ------------------------------
# Imports
# ------------------------------

from guizero import App, Drawing



# ------------------------------
# Functions
# ------------------------------

def draw(event):
    painting.oval(
        event.x - 1, event.y - 1,
        event.x + 1, event.y + 1,
        color="black")


# ------------------------------
# App
# ------------------------------

app = App("Paint")

painting = Drawing(app, width="fill", height="fill")

painting.when_mouse_dragged = draw

app.display()
```

paint2.py / Python 3

```
# drawing lines by tracking when the mouse is clicked

# ------------------------------
# Imports
# ------------------------------

from guizero import App, Drawing


# ------------------------------
# Functions
# ------------------------------

def start(event):
    painting.last_event = event

def draw(event):
    painting.line(
        painting.last_event.x, painting.last_event.y,
        event.x, event.y,
        color="black",
        width=3
    )

    painting.last_event = event


# ------------------------------
# App
# ------------------------------

app = App("Paint")

painting = Drawing(app, width="fill", height="fill")

painting.when_left_button_pressed = start
painting.when_mouse_dragged = draw

app.display()
```

paint3.py / Python 3

```
# widgets to set the color and width

# ------------------------------
# Imports
# ------------------------------

from guizero import App, Drawing, Combo, Slider


# ------------------------------
# Functions
# ------------------------------

def start(event):
    painting.last_event = event

def draw(event):
    painting.line(
        painting.last_event.x, painting.last_event.y,
        event.x, event.y,
        color=color.value,
        width=width.value
    )


    painting.last_event = event


# ------------------------------
# App
# ------------------------------

app = App("Paint")

color = Combo(app, options=["black", "white", "red", "green",
"blue"])
width = Slider(app, start=1, end=10)

painting = Drawing(app, width="fill", height="fill")

painting.when_left_button_pressed = start
painting.when_mouse_dragged = draw

app.display()
```

paint4.py / Python 3

```
# adding different drawing shapes

# ------------------------------
# Imports
# ------------------------------

from guizero import App, Drawing, Combo, Slider


# ------------------------------
# Functions
# ------------------------------

def start(event):
    painting.last_event = event
    painting.first_event = event
    painting.last_shape = None

def draw(event):
    if shape.value == "line":
        painting.line(
            painting.last_event.x, painting.last_event.y,
            event.x, event.y,
            color=color.value,
            width=width.value
        )

    if shape.value == "rectangle":

        if painting.last_shape is not None:
            painting.delete(painting.last_shape)

        rectangle = painting.rectangle(
            painting.first_event.x, painting.first_event.y,
            event.x, event.y,
            color=color.value
        )

        painting.last_shape = rectangle

    painting.last_event = event


# ------------------------------
# App
```

paint4.py (cont.) / Python 3

```
# ----------------------------

app = App("Paint")

color = Combo(app, options=["black", "white", "red", "green",
"blue"])
width = Slider(app, start=1, end=10)
shape = Combo(app, options=["line", "rectangle"])

painting = Drawing(app, width="fill", height="fill")

painting.when_left_button_pressed = start
painting.when_mouse_dragged = draw

app.display()
```

10-paint.py / Python 3

```
# styled up

# ------------------------------
# Imports
# ------------------------------

from guizero import App, Drawing, Combo, Slider, Box, Text


# ------------------------------
# Functions
# ------------------------------

def start(event):
    painting.last_event = event
    painting.first_event = event
    painting.last_shape = None

def draw(event):
    if shape.value == "line":
        painting.line(
            painting.last_event.x, painting.last_event.y,
            event.x, event.y,
            color=color.value,
            width=width.value
        )

    else:
        if painting.last_shape is not None:
                painting.delete(painting.last_shape)

        if shape.value == "rectangle":

            painting.last_shape = painting.rectangle(
                painting.first_event.x, painting.first_event.y,
                event.x, event.y,
                color=color.value
            )

        if shape.value == "oval":

            painting.last_shape = painting.oval(
                painting.first_event.x, painting.first_event.y,
                event.x, event.y,
                color=color.value
```

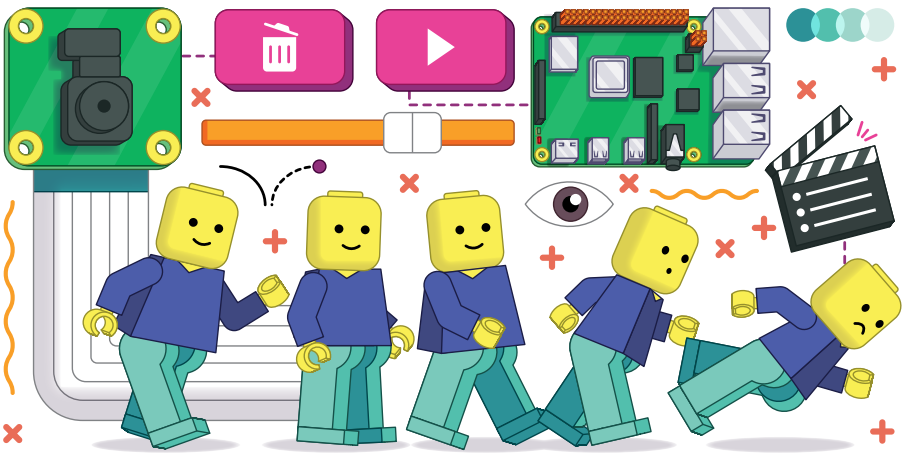
10-paint.py (cont.) / Python 3

```
            )

    painting.last_event = event


# ------------------------------
# App
# ------------------------------

app = App("Paint")
app.font = "impact"

tools = Box(app, align="top", width="fill", border=True)

Text(tools, text="Tool", align="left")
shape = Combo(tools, options=["line", "rectangle", "oval"],
align="left")

Text(tools, text="Colour", align="left")
color = Combo(tools, options=["black", "white", "red", "green",
"blue"], align="left")

Text(tools, text="Width", align="left")
width = Slider(tools, start=1, end=10, align="left")

painting = Drawing(app, width="fill", height="fill")

painting.when_left_button_pressed = start
painting.when_mouse_dragged = draw

app.display()
```

Chapter 11

Stop-frame Animation

Build your own stop-frame animated GIF creator



This projects uses a Raspberry Pi Camera Module and guizero to make a stop-frame animation application (Figure 1). To complete this project, you will need a Raspberry Pi with an official Camera Module (or High Quality Camera). If you need help connecting up the Camera Module, take a look at the 'Getting started with the Camera Module' guide at **rpf.io/picamera**.

You will need guizero installed with the optional 'images' functionality, which you can install by running this command in the terminal:

```
pip3 install guizero[images]
```

If using the Thonny IDE, you may also need to switch to Regular Mode, go to Tools > Manager packages, select guizero, click on the '...' button, check the box for 'Upgrade dependencies', and click on Install.

This project is broken down into stages:

1. Taking a picture with the camera and displaying it on a GUI
2. Taking multiple pictures and saving them to a GIF file
3. Allowing the user to change the GIF
4. Tidying up the GUI

Take a picture

Start by creating this program.

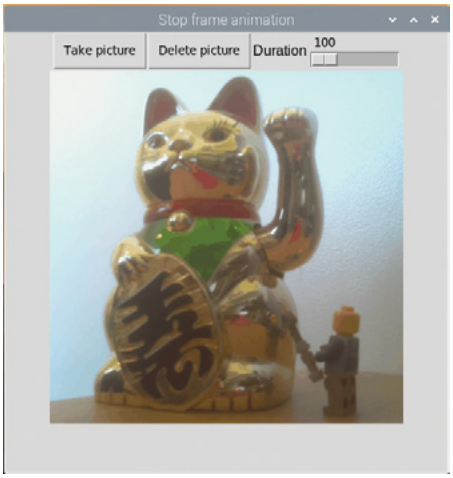


Figure 1 A simple stop-frame animation

```
# Imports ---------------
from guizero import App, Picture, PushButton
from picamera import PiCamera

# Functions -------------
def capture_image():
    camera.capture("frame.jpg")
    viewer.image = "frame.jpg"

# Variables -------------
camera = PiCamera(resolution="400x400")

# App -------------------
app = App(title="Stop frame animation")

take_next_picture = PushButton(app, text="Take picture",
command=capture_image)
viewer = Picture(app)

app.display()
```

Note that the higher the resolution, the greater the processing time. 400×400 is small but really quick to process.

The GUI contains a PushButton and Picture. When the button is pressed, the `capture_image` function is called. The function uses the camera to capture an image and save it as **frame.jpg**. The picture is then displayed in the Picture widget.

Test the program (**stopframe1.py**, page 131). When you click the 'Take picture' button, the image should be displayed on the GUI (**Figure 2**).

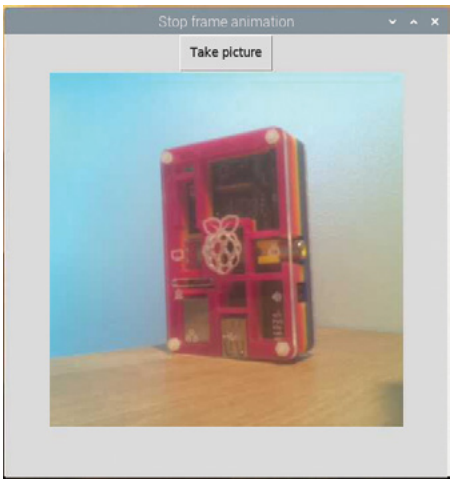


Figure 2 Take a picture

Take multiple images and save to a GIF

An animation is made of multiple pictures, known as frames. Each frame in the animation will be slightly different to the last and when played together at speed, the animation will appear to move.

In this step, you will change your GUI to keep a list of all the frames taken and use PIL (Python Imaging Library) to save the frames as an animated GIF which will be displayed in the viewer. At the top of your program, import the Image module from PIL:

```
from PIL import Image
```

Create a list to store the frames of your animation:

```
frames = []
```

To keep track of how many frames have been taken, import a Text widget, add it to your app, and set it to 0.

```
from guizero import App, Picture, PushButton, Text

total_frames = Text(app, text="0")
```

Each time a new image is captured, you will need to open it and append it to your list of frames:

```
def capture_image():
    camera.capture("frame.jpg")
    viewer.image = "frame.jpg"

    frame = Image.open("frame.jpg")
    frames.append(frame)
    total_frames.value = len(frames)
```

The `len` (length) of the `frames` list is then used to update the text in `total_frames`.

Your program should now look similar to **stopframe2.py**. Test it and make sure the number of frames increases each time you take a picture.

Save as a GIF

You can PIL to save all the frames as one animated GIF. Create a new `save_animation` function to save the frames as **animation.gif**.

```
def save_animation():
    if len(frames) > 0:
        viewer.show()
        frames[0].save(
            "animation.gif",
            save_all=True,
            append_images=frames[1:])
        viewer.image = "animation.gif"
    else:
        viewer.hide()
```

There is a lot happening here, but by breaking down the code you can see how this works.

If the number of frames in the list is greater than 0, then the viewer is shown, otherwise it is hidden:

```
    if len(frames) > 0:
        viewer.show()
        ...
    else:
        viewer.hide()
```

The frames are then saved to a file called **animation.gif**. The first frame (`frames[0]`) is saved, the remaining frames (`frames[1:]`) are appended, and all are saved to the animated GIF:

```
        frames[0].save(
            "animation.gif",
            save_all=True,
            append_images=frames[1:])
```

The **animation.gif** is then shown in the viewer:

```
        viewer.image = "animation.gif"
```

Call the `save_animation` function at the *end* of the `capture_image` function to create and display the animation:

```
    def capture_image():
        camera.capture("frame.jpg")
        viewer.image = "frame.jpg"

        save_animation()
```

Your code should now be similar to **stopframe3.py**. Test it out.

Delete the last frame

At the moment, if you make a mistake while creating your animated GIF, you have to start again from the beginning.

You should modify your GUI to allow the last frame taken to be deleted, so if a mistake is made you can undo the change.

Create a new function which will remove or pop the last frame from the list, save the changed animation, and then display it.

```
    def delete_frame():
        if len(frames) > 0:
            frames.pop()
            total_frames.value = len(frames)

        save_animation()
```

The length of the `frames` list is checked before attempting to pop the last item. An error would be raised if you tried to pop an item from an empty list.

Add a PushButton to the GUI to call the `delete_frame` function, by inserting this code in your app:

```
 delete_last_picture = PushButton(controls, align="left",
text="Delete last", command=delete_frame)
```

Note: *You could also modify the GUI to allow you to delete any frame, not just the last one.*

Changing the timing

Each frame is displayed for the default duration time of 100 milliseconds. Include a Slider widget in your GUI to allow the duration to be changed.

Add it to the list of imports.

```
 from guizero import App, Picture, PushButton, Text, Slider
```

Then create the widget in the app.

```
 Text(app, text="Duration")
 duration = Slider(app, start=100, end=1000, command=save_
animation)
```

The `start` and `end` parameters will be the minimum and maximum times you can set for the frame duration.

Each time the slider is changed, the `save_animation` function will be run.

Update the `save_animation` function to use the duration value when saving the GIF.

```
    frames[0].save(
        "animation.gif",
        save_all=True,
        append_images=frames[1:],
        duration=duration.value)
```

Your code should now resemble **stopframe4.py**. Try it out.

CONSISTENT CAPTURES

As the camera is using `"auto"`, each time a image is captured, the setting used may change. This will cause each image to be slightly different and will cause a flickering in your animation.

By fixing the camera settings when the program starts, you can stop this from happening.

The required settings will depend on the lighting where you are taking picture.

This article from the picamera documentation provides more information and example settings: **rpf.io/picamera-consistent**.

Align the controls

At the moment, the controls are taking up a lot of room stacked at the top of the GUI (**Figure 3**). Create a Box and align it to the top of the GUI to hold the controls, first adding it to the imports.

```
from guizero import App, Picture, PushButton, Text, Slider, Box

controls = Box(app, align="top")
```

Modify the widgets so that they are in the controls box and set the align parameter to `"left"`. For example:

```
total_frames = Text(controls, text="0", align="left")
```

Aligning widgets to the left inside the box will make them stack up next to each other.

Repeat this for rest of the controls so they are all put into the top box and lined up next to each other.

Your complete program should look similar to **11-stop-frame.py**.

Figure 3 Controls stacked at the top

stopframe1.py / Python 3

DOWNLOAD
magpi.cc/guizerocode

```
# Imports ---------------

from guizero import App, Picture, PushButton
from picamera import PiCamera


# Functions -------------

def capture_image():
    camera.capture("frame.jpg")
    viewer.image = "frame.jpg"


# App -------------------

app = App(title="Stop frame animation")

camera = PiCamera(resolution="400x400")
take_next_picture = PushButton(app, text="Take picture",
command=capture_image)
viewer = Picture(app)

app.display()
```

stopframe2.py / Python 3

```
# Imports ---------------

from guizero import App, Picture, PushButton, Text
from picamera import PiCamera
from PIL import Image


# Functions -------------

def capture_image():
    camera.capture("frame.jpg")
    viewer.image = "frame.jpg"
```

stopframe2.py (cont.) / Python 3

```
    frame = Image.open("frame.jpg")
    frames.append(frame)
    total_frames.value = len(frames)


# Variables -------------

frames = []

camera = PiCamera(resolution="400x400")


# App -------------------

app = App(title="Stop frame animation")

total_frames = Text(app, text="0")
take_next_picture = PushButton(app, text="Take picture",
command=capture_image)

viewer = Picture(app)

app.display()
```

stopframe3.py / Python 3

```
# Imports ---------------

from guizero import App, Picture, PushButton, Text
from picamera import PiCamera
from PIL import Image


# Functions -------------

def capture_image():
    camera.capture("frame.jpg")
    viewer.image = "frame.jpg"

    frame = Image.open("frame.jpg")
```

stopframe3.py (cont.) / Python 3

```
    frames.append(frame)
    total_frames.value = len(frames)

    save_animation()

def save_animation():
    if len(frames) > 0:
        viewer.show()
        frames[0].save(
            "animation.gif",
            save_all=True,
            append_images=frames[1:])
        viewer.image = "animation.gif"
    else:
        viewer.hide()


# Variables -------------

frames = []

camera = PiCamera(resolution="400x400")


# App -------------------

app = App(title="Stop frame animation")

total_frames = Text(app, text="0")
take_next_picture = PushButton(app, text="Take picture",
command=capture_image)

viewer = Picture(app)

app.display()
```

stopframe4.py / Python 3

```
# Imports ---------------

from guizero import App, Picture, PushButton, Text, Slider
from picamera import PiCamera
from PIL import Image


# Functions -------------

def capture_image():
    camera.capture("frame.jpg")
    viewer.image = "frame.jpg"

    frame = Image.open("frame.jpg")
    frames.append(frame)
    total_frames.value = len(frames)

    save_animation()

def save_animation():
    if len(frames) > 0:
        viewer.show()
        frames[0].save(
            "animation.gif",
            save_all=True,
            append_images=frames[1:],
            duration=duration.value)
        viewer.image = "animation.gif"
    else:
        viewer.hide()

def delete_frame():
    if len(frames) > 0:
        frames.pop()
        total_frames.value = len(frames)

    save_animation()


# Variables -------------

frames = []

camera = PiCamera(resolution="400x400")
```

stopframe4.py (cont.) / Python 3

```
# App -------------------

app = App(title="Stop frame animation")

total_frames = Text(app, text="0")
take_next_picture = PushButton(app, text="Take picture",
command=capture_image)
delete_last_picture = PushButton(app, text="Delete last",
command=delete_frame)
Text(app, text="Duration")
duration = Slider(app, start=100, end=1000, command=save_
animation)

viewer = Picture(app)

app.display()
```

11-stop-frame.py / Python 3

```
# Imports ---------------

from guizero import App, Picture, PushButton, Text, Slider, Box
from picamera import PiCamera
from PIL import Image


# Functions -------------

def capture_image():
    camera.capture("frame.jpg")
    viewer.image = "frame.jpg"

    frame = Image.open("frame.jpg")
    frames.append(frame)
    total_frames.value = len(frames)

    save_animation()

def save_animation():
    if len(frames) > 0:
        viewer.show()
        frames[0].save(
```

11-stop-frame.py (cont.) / Python 3

```
            "animation.gif",
            save_all=True,
            append_images=frames[1:],
            duration=duration.value)
        viewer.image = "animation.gif"
    else:
        viewer.hide()

def delete_frame():
    if len(frames) > 0:
        frames.pop()
        total_frames.value = len(frames)

    save_animation()


# Variables -------------

frames = []

camera = PiCamera(resolution="400x400")


# App -------------------

app = App(title="Stop frame animation")

controls = Box(app, align="top")
total_frames = Text(controls, text="0", align="left")
take_next_picture = PushButton(controls, align="left", text="Take
picture", command=capture_image)
delete_last_picture = PushButton(controls, align="left",
text="Delete last", command=delete_frame)
Text(controls, align="left", text="Duration")
duration = Slider(controls, align="left", start=100, end=1000,
command=save_animation)

viewer = Picture(app)

app.display()
```

Appendix A

Setting up

Learn how install Python and an IDE

Here we will show you how to set everything up on your computer in order to create Python programs with graphical user interfaces. To be able to run and edit the applications in this book, you'll need three things:

1. The Python interpreter – this is the software that allows you to run programs written in Python.
2. An integrated development environment (IDE) – software which includes a code editor and the ability to run a program from that editor. Python comes bundled with an IDE called IDLE, but you might choose to use a different IDE.
3. The guizero Python library – instructions for installing this are given in Chapter 1, but if you are using Thonny please refer to the section below as the instructions are slightly different.

There are many IDEs available; here we're going to look at two of them – IDLE and Thonny. IDLE is a very simple IDE which comes bundled with Python for Windows and Mac, and is installed by default on some versions of Raspberry Pi OS. Thonny has some additional features, but it is still geared towards beginners.

Occasionally, errors can occur while trying to get everything installed and running – especially on older computers. If you experience errors while trying to use a particular IDE or version of Python, try another IDE or Python version.

Installing Python and IDLE

Windows

Windows does not come with Python 3 pre-installed. If you think you may have installed Python previously, you can check this by looking for 'Python' in the Start menu or under 'Apps and Features' within Settings. If you intend to use Thonny as your IDE, you can skip ahead to the 'Thonny' section as Python is automatically installed alongside it.

Go to **python.org**, mouse-over Downloads and click on Windows. Choose the option to directly download the latest stable Python 3 release (most people will need the one labelled *Windows x86-64 executable installer*, but this may vary depending on your computer). Once the download is complete, run the program either via your web browser or from your Downloads folder. Click 'Install now' to install using the default options. IDLE will also be installed and can be opened via the Start menu or by searching for it by name.

Alternatively, if you have Windows 10, you could download the Microsoft Store package of the latest Python version (currently 3.8). If you have any difficulties, full installation instructions can be found at **rpf.io/python-windows**.

Mac

Although most versions of macOS come with a Python interpreter, it's version 2.7 which is not compatible with guizero. You will need to install Python 3 alongside the existing installation.

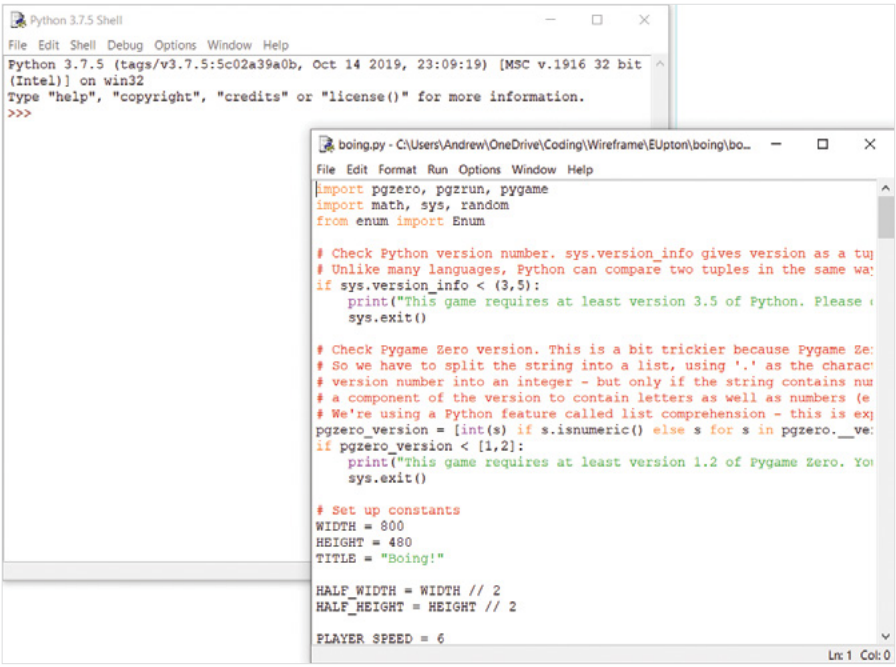
Go to **python.org**, mouse-over Downloads and click on Mac OS X. Choose the latest stable release, and click on *macOS 64-bit installer*. Once the download is complete, run the program either via your web browser or from your Downloads folder. Install using the default options. IDLE will also be installed and can now be opened from the Launchpad or Applications folder.

Raspberry Pi

Raspberry Pi OS (previously known as Raspbian) comes with Python already installed. However, recent versions of the OS come with the Thonny but do not include IDLE. To install IDLE, make sure you are connected to the internet, then open a Terminal and type:

```
sudo apt-get install idle3
```

If you have any problems getting the code in this book to run, try upgrading to the most recent version of Raspberry Pi OS.



IDLE usually comes pre-installed with Python

IDEs

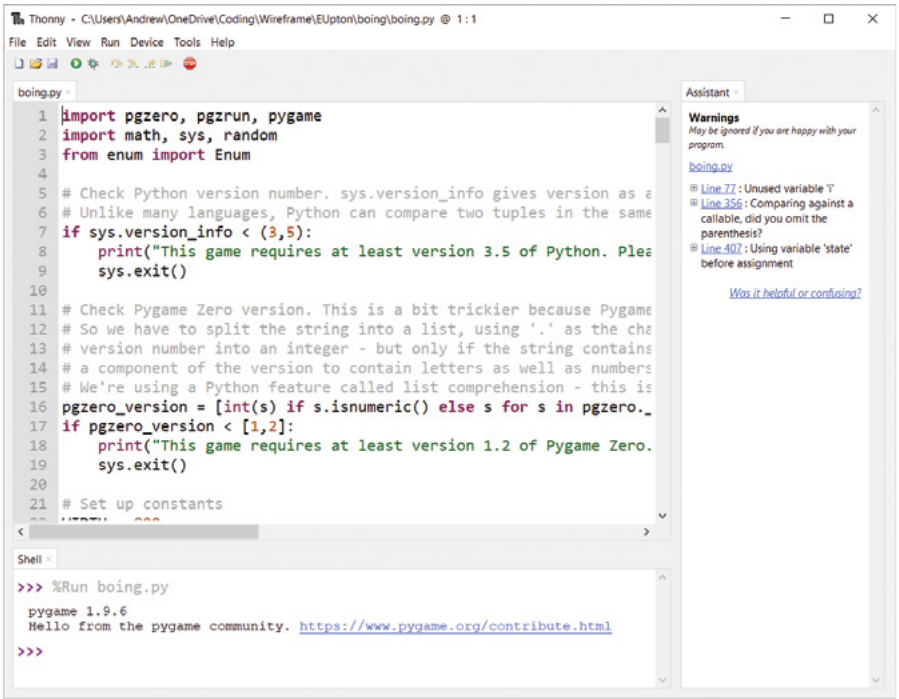
IDLE

IDLE is a basic IDE which is usually automatically installed alongside Python. Once IDLE starts, the first thing you'll see is a window titled 'Python 3.8.5 Shell' (the number may be different depending on which exact version you have). This window is called the *shell* and you can use it to type a line of Python code and see the code run straight away.

For example, try typing the following:

```
6 + 2
```

Use the File menu > New file to open a Python file. A file allows you to type multiple lines of code and then run them all, rather than each line running immediately when you press **ENTER**. You can run the program by going to the Run menu and choosing Run Module – or by pressing **F5** on the keyboard. If an error occurs, you will see any error messages in the shell window.



Thonny includes a debugger which may prove useful

Thonny

Thonny comes installed with recent versions of Raspberry Pi OS. For Windows and Mac computers, you can download and install it from **thonny.org**. By default, Thonny uses a version of Python which comes packaged with it, so even if you have already installed Python, Thonny will ignore that version and use its own.

Use the File menu > New to open a Python file. You will type your code in the top white box which has a number 1 to the left side. You can run the program by selecting the 'Run current script' button, or by pressing **F5** on the keyboard. If an error occurs, you will see any error messages in the shell area at the bottom of the screen.

Thonny includes a debugger which allows you to step through the code one line at a time and see how the variables change.

Because Thonny uses its own Python installation, you will need to install guizero from inside Thonny in order for Thonny to be able to access it. Make sure you are connected to the internet, then click on the Tools menu > Manage packages. In the window that appears, type guizero in the box and click 'Find package from PyPI'. Thonny will locate the package for you; click Install to install guizero within Thonny's own Python environment.

Appendix B

Get started with Python

If you're a complete beginner, here's how to start coding in Python

Unlike a visual, block-based coding environment like Scratch, Python is text-based: you write instructions, using a simplified language and specific format, which the computer then carries out. Python is a great next step for those who have already used Scratch, offering increased flexibility and a more 'traditional' programming environment. In the following examples, we're using the Thonny IDE (integrated development environment), but you can use an alternative IDE if you prefer (see **Appendix A**).

First program

The top white box in the Thonny window is where you write your program script. Click in this box and type the following code:

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

Now click the Run icon in the Thonny toolbar and you will be asked to save your program first; type a descriptive name, like 'Hello World', and click the Save button. Once your program has saved, you'll see two messages appear in the Python shell area:

```
>>> %Run 'Hello World.py'
Hello, World!
```

Congratulations, you have successfully written and run a Python script! You will use the same method for all of the programs in this book – write the code in the script area and then run it.

Loops and code indentation

Just as Scratch uses stacks of jigsaw-like blocks to control which bits of the program are connected to which other bits, Python has its own way of controlling the sequence in which its programs run: indentation.

Create a new program by clicking on the New icon in the Thonny toolbar. You won't lose your existing program; instead, Thonny will create a new tab above the script area. Type in the following code:

```
print ("Loop starting!")
for i in range(10):
    print ("Loop")
```

Click the Run icon in the Thonny toolbar, save your program with the name 'Indentation', and watch the shell area for its output. See if you can work out what is happening.

The first line prints a message to the shell, just like your Hello World program. The second tells Python to start a loop which runs 10 times – the number of times the loop runs is controlled by the `range(10)` instruction. The third line is indented, which means it is pushed inwards compared to the other lines. This indentation is how Python tells the difference between instructions outside the loop and instructions inside the loop. In Scratch, the instructions to be included in the loop are placed within the C-shaped block, and in Python they are indented. So this means that the instruction to print the word 'Loop' is repeated 10 times.

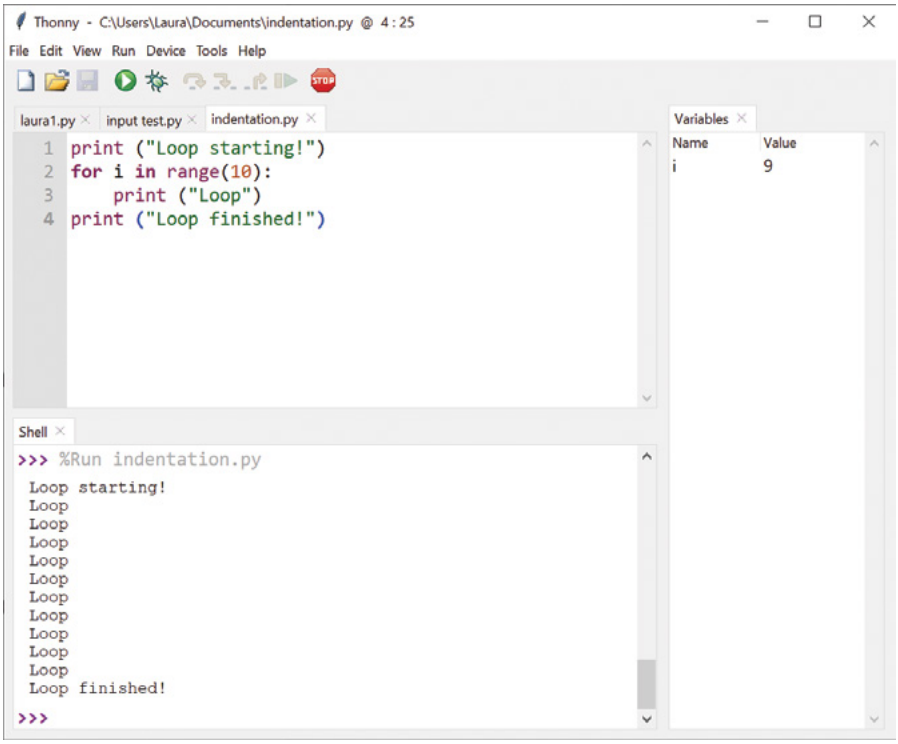
You'll notice that when you pressed **ENTER** at the end of the third line, Thonny automatically indented the next line, assuming it would be part of the loop. To remove this, just press the **BACKSPACE** key once and then type a fourth and final line:

```
print ("Loop finished!")
```

Your four-line program is now complete. The first line sits outside the loop, and will only run once; the second line sets up the loop; the third sits inside the loop and will run once for each time the loop loops; and the fourth line sits outside the loop once again.

Run the program again. If you haven't made the Thonny window larger, you may need to use the scroll bar to the right of the shell area to see its full output:

```
Loop starting!
Loop
Loop
Loop
Loop
Loop
Loop
Loop
Loop
Loop
Loop
Loop finished!
```



Run the program and see the result in the shell area below

Indentation is a powerful part of Python, and one of the most common reasons for a program to not work as you expected. When looking for problems in a program, always double-check the indentation.

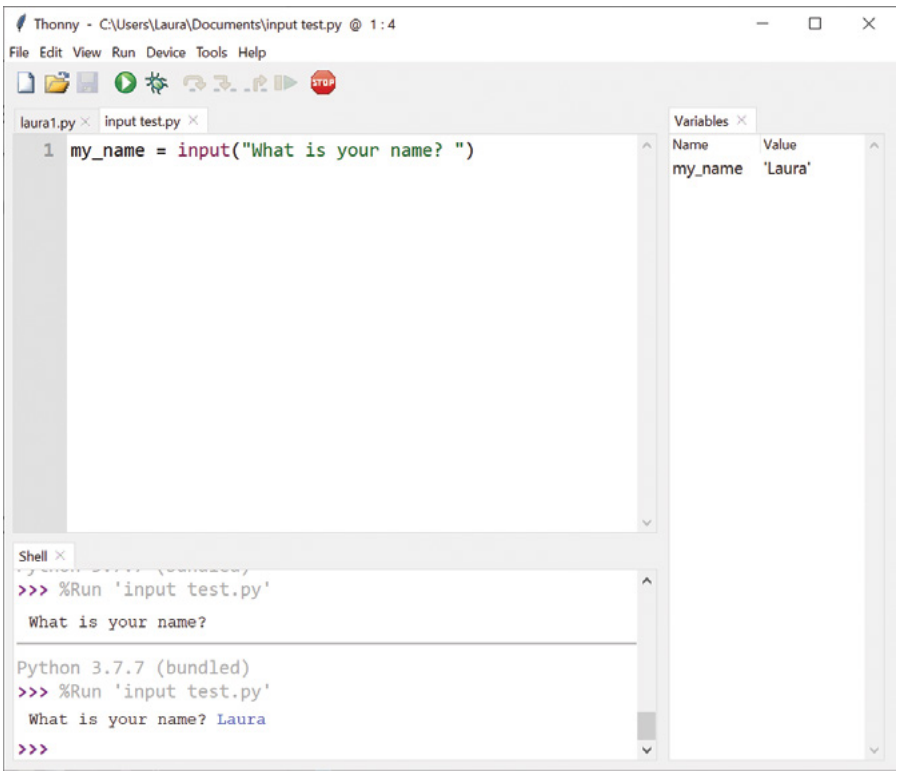
Conditionals and Variables

You can create variables in your program to store values; for example, you might want to store some data the user typed in, or the result of a calculation.

Start a new program by clicking the New icon on the Thonny menu, then type the following into the script area:

```
my_name = input("What is your name? ")
```

Click the Run icon, save your program with the name 'Input test', and watch what happens in the shell area: you'll be asked for your name. Type your name and press **ENTER**. The variables area to the right of the Thonny window will automatically display the name of the variable



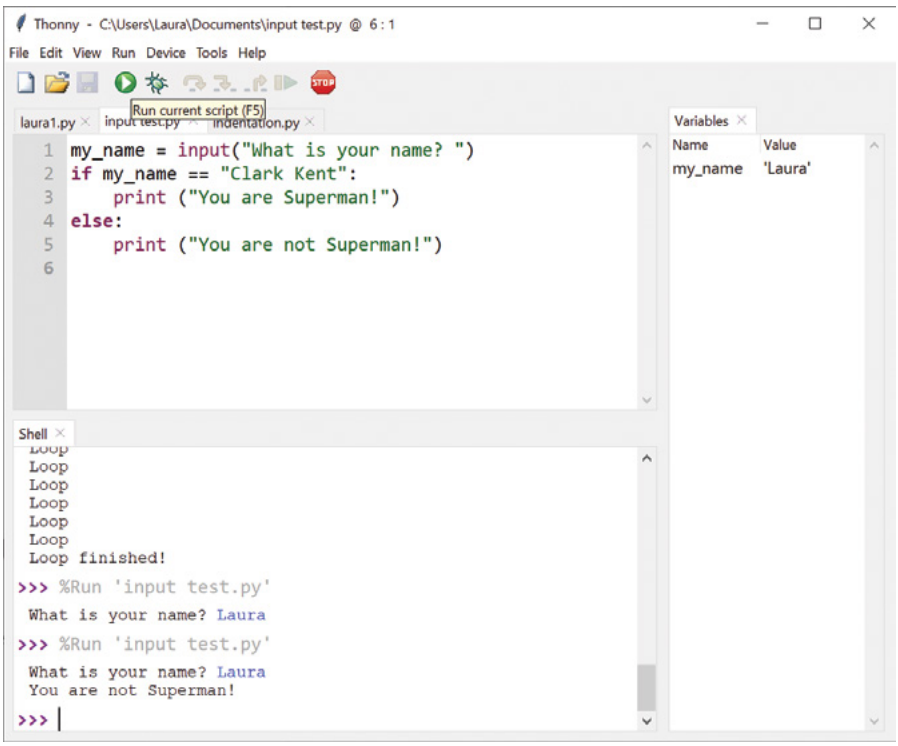
Variable names and values are shown in the area on the right

(my_name) and its value (e.g. 'Laura'). If you can't see the variables area, check on the View menu that there is a tick next to Variables. This information remains displayed even when the program isn't running, making it easy to see what your program has been doing.

The program has saved what you typed as your name as the value of the variable called `my_name`. Using the `input` command is useful for basic Python programs, but when you attempt the GUI programs in this book, you will learn about other ways to capture input from the user and store it in variables to be used in your program.

To make your program do something useful with the name, add a conditional statement by typing the following:

```
if my_name == "Clark Kent":
    print ("You are Superman!")
else:
    print ("You are not Superman!")
```



Unless you enter your name is entered as Clark Kent, you're not Superman

Remember that when Thonny sees that your code needs to be indented, it will do so automatically – but it doesn't know when your code needs to stop being indented, so you'll have to delete the spaces yourself.

Click the Run icon and type your name into the shell area, as before. Unless your name happens to be Clark Kent, you'll see an additional message 'You are not Superman!' in the shell area. Click Run again, and this time type in the name 'Clark Kent' – making sure to write it exactly as in the program, with a capital C and a capital K. This time, the program recognises that you are, in fact, Superman.

The `==` symbol tells Python to compare two values. In this case it will look to see if the value of the variable `my_name` matches the text `"Clark Kent"`.

USING = AND ==

We have now seen two different operators – the single equals sign (=) and the double equals sign (==). They mean different things and it is important to know the difference. The single equals (=) means "it IS equal to this value," while a double equals (==) means "IS IT equal to this value?" The first is used to assign a value to a variable, and the second is used to compare two values.

When you are creating the programs in this book, you will collect input from the user, store data in variables, display information on the screen, and use loops. The examples we have worked through in this section are very basic and only allow the user to interact with the program via text. We hope that now you know the basics of how to write a Python program you will enjoy creating GUIs as a more graphical way of interacting with your program.

Appendix C

Widgets in guizero

An overview of the widgets used in guizero

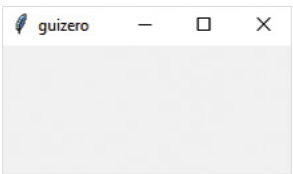
Widgets in guizero are how you create your graphical user interface. They are the things which appear on the GUI, everything from the app itself to text boxes, buttons and pictures.

Note: *This is an overview of the widgets in guizero. Be sure to view the specific online documentation for each widget for more information:* ***lawsie.github.io/guizero***.

Widgets

App

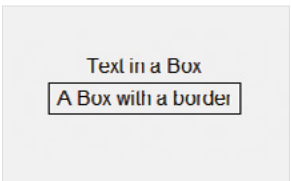
The App object is the basis of all GUIs created using guizero. It is the main window which contains all of the other widgets.




```
app = App()
app.display()
```

Box

The Box object is an invisible container which can contain other widgets.

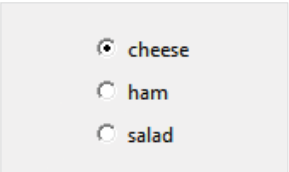



```
box = Box(app)
box = Box(app, border=True)
```

ButtonGroup

The ButtonGroup object displays a group of radio buttons, allowing the user to choose a single option.

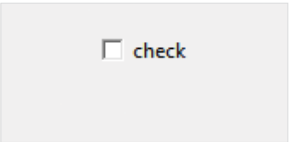
```
choice = ButtonGroup(app,
options=["cheese", "ham", "salad"])
```



CheckBox

The CheckBox object displays a check box to allow an option to be ticked or un-ticked.

```
checkbox = CheckBox(app, text="salad ?")
```



Combo

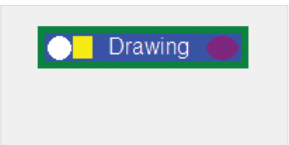
The Combo object displays a drop-down box allowing a single item to be selected from a list of options.

```
combo = Combo(app, options=["cheese",
"ham", "salad"])
```

Drawing

The Drawing object allows shapes, images, and text to be created.

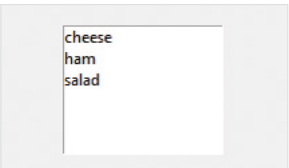
```
drawing = Drawing(app)
```



ListBox

The ListBox object displays a list of items from which either single or multiple items can be selected.

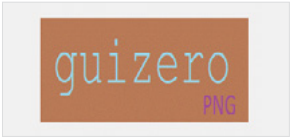
```
listbox = ListBox(app, items=["cheese",
"ham", "salad"])
```



Picture

The Picture object displays an image.

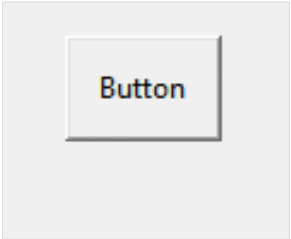
```
picture = Picture(app, image="guizero.
png")
```



PushButton

The PushButton object displays a button with text or an image, which calls a function when pressed.

```
def do_nothing():
    print("button pressed")

button = PushButton(app, command=do_nothing)
```



Slider

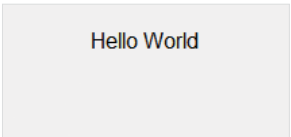
The Slider object displays a bar and selector which can be used to specify a value in a range.

```
slider = Slider(app)
```

Text

The Text object displays non-editable text in your app – useful for titles, labels, and instructions.

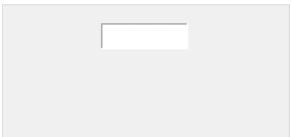
```
text = Text(app, text="Hello World")
```



TextBox

The TextBox object displays a text box which the user can type in.

```
textbox = TextBox(app)
```



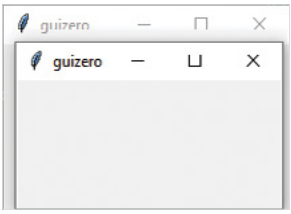
Waffle

The Waffle object displays an n×n grid of squares with custom dimensions and padding.

```
waffle = Waffle(app)
```

Window

The Window object creates a new window in guizero.




```
window = Window(app)
```

Properties

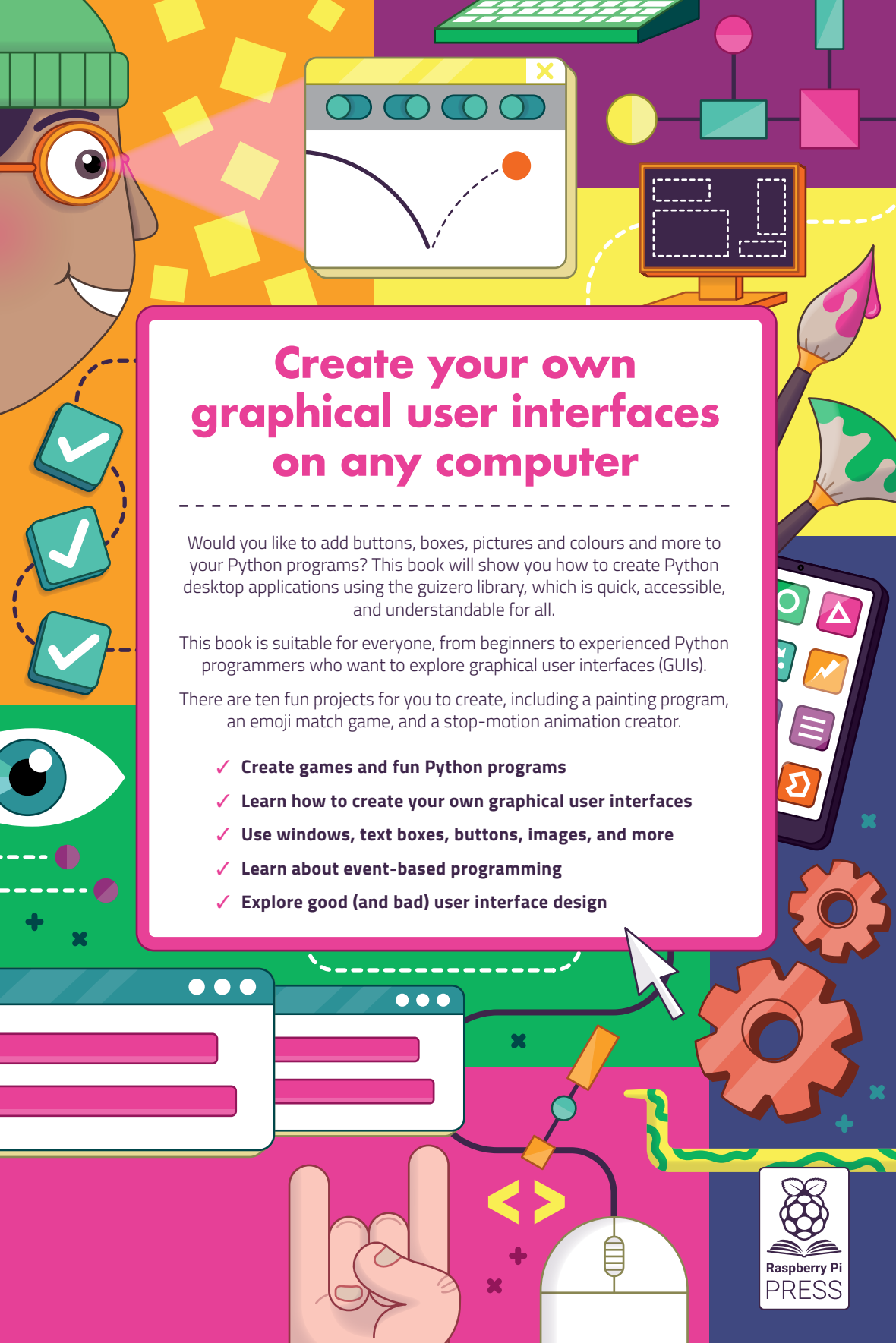
All widgets are customisable through their properties. These properties are typical for most widgets. Check a widget's online documentation (e.g. **lawsie.github.io/guizero/app**) for details.

PROPERTY	DATA TYPE	DESCRIPTION
`align`	string	The alignment of this widget within its container
`bg`	string, List	The background colour of the widget
`enabled`	boolean	True if the widget is enabled
`font`	string	The font of the text
`grid`	List	[x,y] co-ordinates of this widget if in a 'grid'
`height`	int, string	The height of the widget
`master`	App, Window, Box	The container to which this widget belongs
`value`	int, string, bool	The widget's current 'value', e.g. the text in a TextBox
`visible`	boolean	If this widget is visible
`width`	size	The width of the widget
`ext_size`	int	The size of the text
`ext_color`	color	The colour of the text

Methods

Widgets can be interacted with through their methods. The methods supported are dependent on the widget, so check the documentation. These methods are typical across most widgets.

METHOD	DESCRIPTION
`after(time, command, args=None)`	Schedules a single call to command after time milliseconds
`cancel(command)`	Cancels a scheduled call to command
`destroy()`	Destroys the widget
`disable()`	Disables the widget so that it cannot be interacted with
`enable()`	Enables the widget
`focus()`	Gives focus to the widget
`hide()`	Hides the widget from view
`repeat(time, command, args=None)`	Schedules a call to command every time milliseconds
`resize(width, height)`	Sets the width and height of the widget
`show()`	Displays the widget if it was previously hidden
`update_command(command, args=None)`	Updates the function to call when the widget is used



Create your own graphical user interfaces on any computer

Would you like to add buttons, boxes, pictures and colours and more to your Python programs? This book will show you how to create Python desktop applications using the guizero library, which is quick, accessible, and understandable for all.

This book is suitable for everyone, from beginners to experienced Python programmers who want to explore graphical user interfaces (GUIs).

There are ten fun projects for you to create, including a painting program, an emoji match game, and a stop-motion animation creator.

- ✓ **Create games and fun Python programs**
- ✓ **Learn how to create your own graphical user interfaces**
- ✓ **Use windows, text boxes, buttons, images, and more**
- ✓ **Learn about event-based programming**
- ✓ **Explore good (and bad) user interface design**

Raspberry Pi PRESS