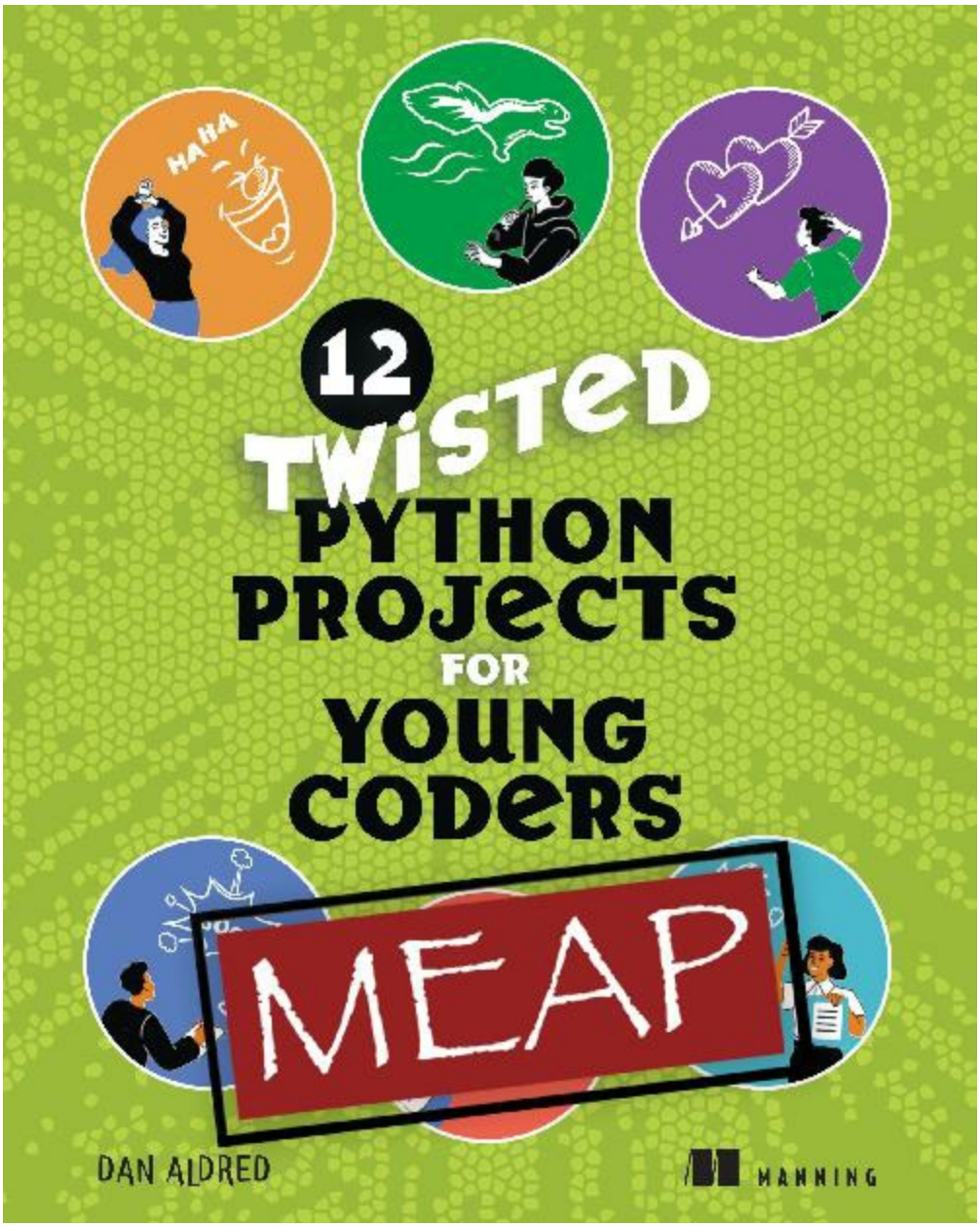

# 12 Twisted Python Projects for Young Coders


DAN ALDRED

MANNING


HA HA
12
TWISTED
PYTHON
PROJECTS
FOR
YOUNG
CODERS
MEAP
DAN ALDRED
MANNING

# 12 Twisted Python Projects for Young Coders

# welcome

Thank you for your purchase of the MEAP for *Twisted Python Projects*! For people who are new to Python, it's important to have fun while learning. I understand that, and that’s why I’ve combined Python with guizero, a simple program for creating apps, to bring you 12 twisted and entertaining projects. From a joke machine to a fart soundboard, a number guessing game, a prank quiz, a pixel painter, and more, each chapter covers everything you need to know to build these fun and unusual apps. And don’t worry, there’s even a sensible chapter where you’ll learn to build an app to check the security of passwords.

Throughout the book, I guide you through each stage of the code, explaining what it does and why we use it. Through a combination of diagrams, callouts, graphics, and code listings, you’ll learn essential Python and guizero programming skills. Plus, at the end of each chapter, you’ll find extra challenges to make your app even more twisted, with all the code included for you to experiment with.

With the skills you’ll learn, you’ll be able to create your own fun apps beyond the ones presented in the book. Thank you again for your interest and support. I’m excited to hear your feedback on the platform and look forward to helping you become a twisted Python programmer!

Please be sure to post any questions, comments, or suggestions you have in the liveBook discussion forum. I look forward responding to you on the platform.

—Dan Aldred (TeCoEd)

**In this book**

# 1 Hello GUI


## This chapter covers

- Understanding what Python is
- Knowing what a GUI is
- Installing the required software
- Building your first GUI

Welcome to *Twelve Twisted Python Projects for Young Coders*, eleven outrageous projects and one sensible one. This book walks you through building these fun projects, such as the F.A.R.T soundbox, a personal timer, a BAE or Bust calculator, even a pixel painter. Each chapter presents a new and exciting project that is fun in its own right and teaches you programming skills in Python and guizero. Python is a popular and versatile coding language, and guizero is an easy way to create interactive graphical user interfaces, GUIs. You can find GUIs everywhere, for example, the menus and buttons that enable you to interact with your smartphone or tablet. The apps and games that you can download and play, the Operating System on your computer. GUIs are everywhere. The software we will use to create our GUIs is called guizero. More on this later.

Each chapter covers building a new GUI app with different features. When

you’re done with each project you can combine your new programming skills and techniques from each chapter to create your own or new versions of the GUIs.

This first chapter is a comfortable introduction to creating your first GUI using the Python programming language. You will be walked through every step of building a simple project, from installing the software, creating a file, writing the program code, testing your code, and then making improvements.

Most of this book’s projects use the Microsoft Windows platform; however, Python and guizero are compatible with Linux and Apple computers . Appendix A covers installing Python on a Windows computer. Appendix B covers how to install the required software on a Linux or Apple computer.

# Just what is Python?

Python is one of the fastest-growing programming languages. You can use Python to create a wide variety of programs, such as websites and games, or for automating tasks, data analysis and so much more.

#### Tell me more about Python

Python was created by Guido van Rossum as a side project and first appeared in February 1991, which is a long time ago! When asked why his project was called Python, Guido said that at the time he was reading scripts from a 1970s BBC TV comedy program called Monty Python's Flying Circus. He needed a short and memorable name for his new programming language and chose to call it Python. Although Python has a funny name, it is used professionally and used as a main language to teach coding in many schools and colleges across the world.

So why is Python so popular? One reason is because Python can be used to write programs for so many different purposes. However, its real strength lies in the simplicity of using the code. For example, consider the beginners “hello world” program, which in the coding world is a well-known first program to create when learning a new language. The program prints out the words “hello world.”

In some programming languages a “hello world” program would look like this:

```
void main() {
  print('Hello, World!');
}
```

In Python it is simply

```
print ("Hello World")
```

The advantage of Python code being so simple to write is that you are less likely to make errors. Some programming languages require special punctuation and spacing. Python still does, but it is less complex. However, do not assume Python is a basic language. You can still create really interesting and engaging GUIs as you will see throughout the chapters; it is just that the Python code is easier to write than other programming languages.

If you already have Python installed, then you can move on to section 1.2. Remember to check out appendix E, which covers commonly used Python features and techniques. If you need to first install Python, then go to Appendix A before reading 1. 2.

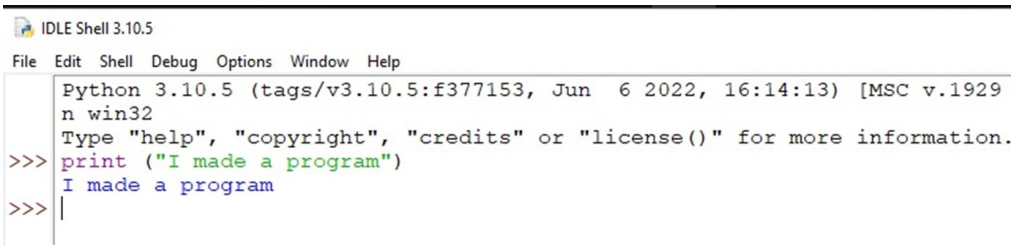
Once Python is installed, you can test that Python is working correctly by,

1. Opening the Python IDLE program.
2. In the window that opens (known as the Shell Window) type out the line of code as shown in see figure 1.1

```
print ("I made a program").
```

3. Press the **Enter** key on the keyboard and the program will print out the message, *I made a program*.

**Figure 1.1 Testing that Python works**

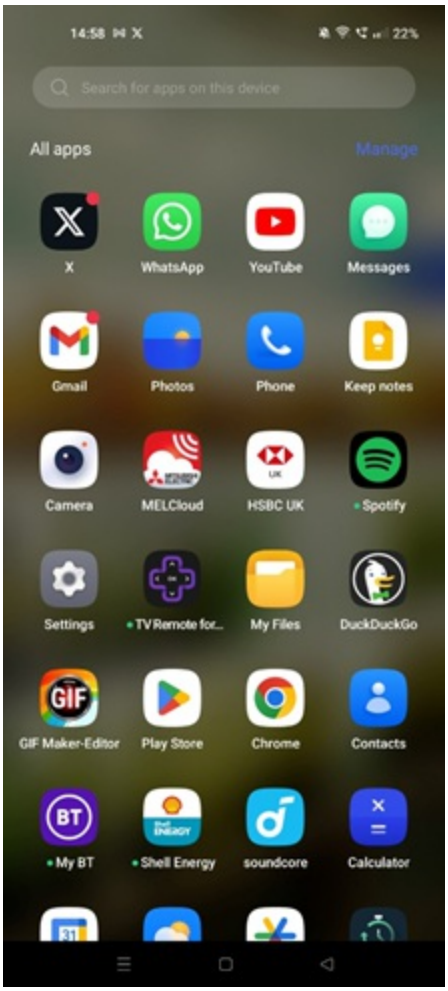



# What is a GUI and what is guizero?

GUI stands for Graphical User Interface. It's pronounced "gooey," like the inside of a Cadburys' Cream Egg. You are probably already familiar with several computing devices. Such as laptops, game consoles and your smartphone. All these devices work using a text-based language that we call a programming language or code. Using text to create code is challenging. It is easy to make mistakes or omit a required part of the code even if you are concentrating really hard. This results in glitches—that is, bugs or the program failing.

An easier way to interact with your computer or device is to use graphics and images; in short, a GUI. (figure 1.2). The Home Screen on your smartphone is GUI, if you want to make a new folder using a GUI, you press the folder icon. If you want to take a photo, you press the camera icon. This is a simpler method of interaction and easier to remember rather than having to write the code to create the folder or to take a picture. Icons are easier to remember and use, and this is why GUIs are found everywhere, from Web Browsers, ATMs, self-service checkout tills, GPS satnav, and so on.

**Figure 1.2 A modern phone GUI**




So, you now know what a GUI is, but what is guizero? Guizero was created by Laura Sach and Martin O'Hanlon and they describe it as, "a Python 3 library for creating simple GUIs. It is designed to allow new learners to quickly and easily create GUIs for their programs." (https://lawsie.github.io/guizero/about/) A library is a group of smaller programs called functions and modules that have been written and packaged together. (Functions are covered in more detail in section 1.3.3)

There are other versions of GUI programs that work with Python, and you can read more about these in the appendix D.

**What is a function?**

A function is a reusable block of code that can be called and used anywhere within a program. Functions save time and make the program more efficient.

This means that you do not have to write the code from scratch; libraries save you a lot of time and reduce your effort in writing programs, For example, a library function can play a sound or display an image. At the time of writing this book there are over 137,000 Python libraries. Let’s get started and install guizero.

## Installing guizero

There are several methods to install guizero and it has even been designed so that you can download the files and make GUIs without installation. This is useful if you are not permitted to download and install the software on the computer. It means that you can just get started with the projects with no need to install anything. Check out Appendix C which covers the guizero easy install method. This section of the chapter covers how to install guizero on a Windows computer. It assumes that you have already installed Python, if you have not, then check out appendix A and follow the steps, then come back here. If you are using an Apple, Linux or other computer, then head over to Appendix B and follow the steps to install guizero on your computer.

To install guizero, we are going to use the command prompt and write a short instruction to pull down the required program files. Before you start the installation, ensure that you are connected to the Internet as you are going to download the installation files.

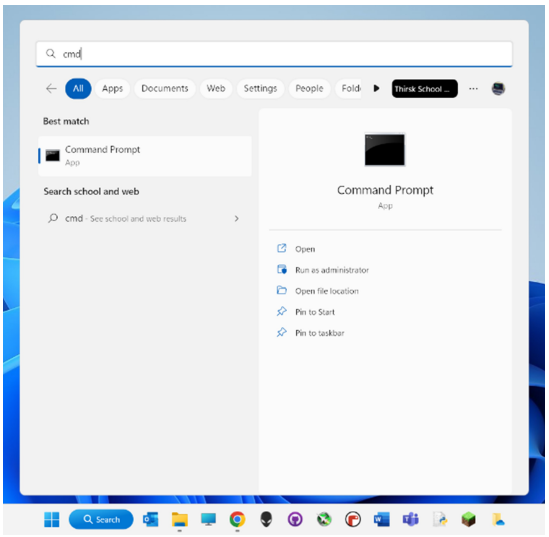
#### Command Prompt

Command Prompt is a command line interpreter application available in most Windows operating systems. It's used to execute entered commands. Most of those commands automate tasks via scripts and batch files, perform advanced administrative functions, and troubleshoot or solve certain kinds of Windows issues.

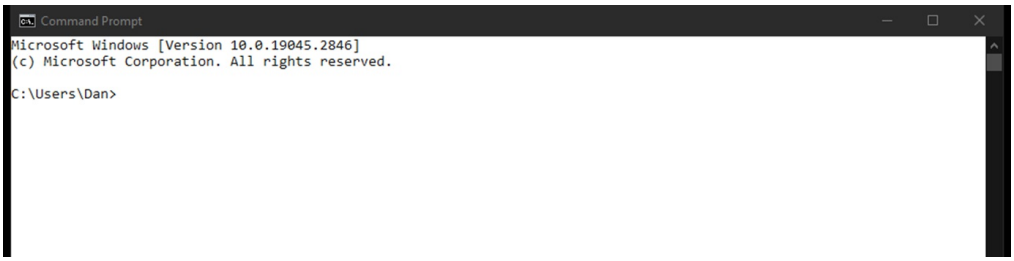
Command Prompt is officially called Windows Command Processor, but it's also sometimes referred to as the command shell or cmd prompt, or even by its filename, cmd.exe.

To open the command prompt,

1. Select and open the Windows start menu.
2. Type the letters cmd.
3. Let Windows search for the command prompt app
4. Select the app and open it.




The command prompt app will load, and you will be presented with the following window.




Now that you have the command window open, let's install guizero. Type in the following command, (a command can be thought of as an instruction, you are writing an instruction for your computer to perform)

```
pip3 install guizero
```

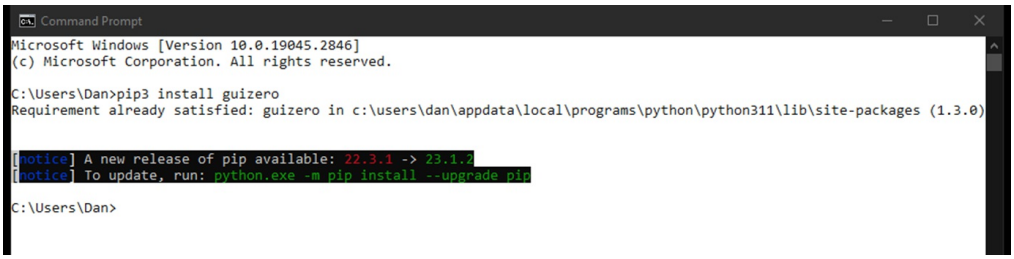

This instruction tells your computer to use a program called pip3 to install the

guizero program. Press the **Enter** key on your keyboard to begin the installation.

#### Pip3

Pip3 is a program sometimes called a package manager that manages the installation of Python libraries, modules and packages.

Guizero will now install, taking a little time. Once complete, close the command prompt.




Remember that this installation method only works if you have already installed Python using the method in appendix A. You can also check out the installation instructions on the main guizero webpage https://lawsie.github.io/guizero/.

## Guizero elements and widgets

For ease of understanding guizero, you can split guizero into two categories. Category one contains the main elements of the software, these elements enable you to build a GUI. For example, Layout is an element. You can use the Layout to adjust the size of the GUI and its buttons and pictures. You can customize your GUI's layout as you wish. The Event elements are used to cause or trigger a reaction (called triggering an action) when something happens; for example, when a person double-clicks the left mouse button on a picture, then some text is displayed.

The second category is widgets, these are the main components of the GUI and there are fifteen widgets that you can use in your GUI. For example, the PushButton widget creates a button with a text label that can be clicked. The ButtonGroup widget creates an option button where you can select from

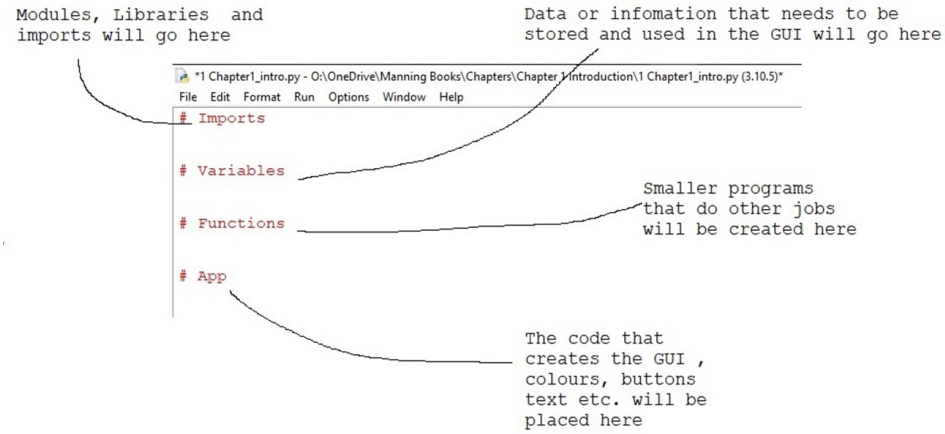
several choices. You will use the widgets throughout the various projects, and you can refer to appendix D for more details.

## Writing a program with Python and guizero

To code a program, you need to use an Integrated Development Environment (IDE). This is an interface that enables you to write programs, test them and run them on your computer. The Python installation includes its own IDE, called IDLE, which is pronounced, ‘idol.’ There are other IDEs available; an overview of the alternatives is covered in appendix A.

When writing each program, the code will always have four main sections (figure 1.3). These sections break up the lines of code into organized and manageable chunks. This organization will make it easier to check your program for errors and will also make it easier when you wish to make edits or changes. When you organize your code, good organization makes it easier to find the code that you need.

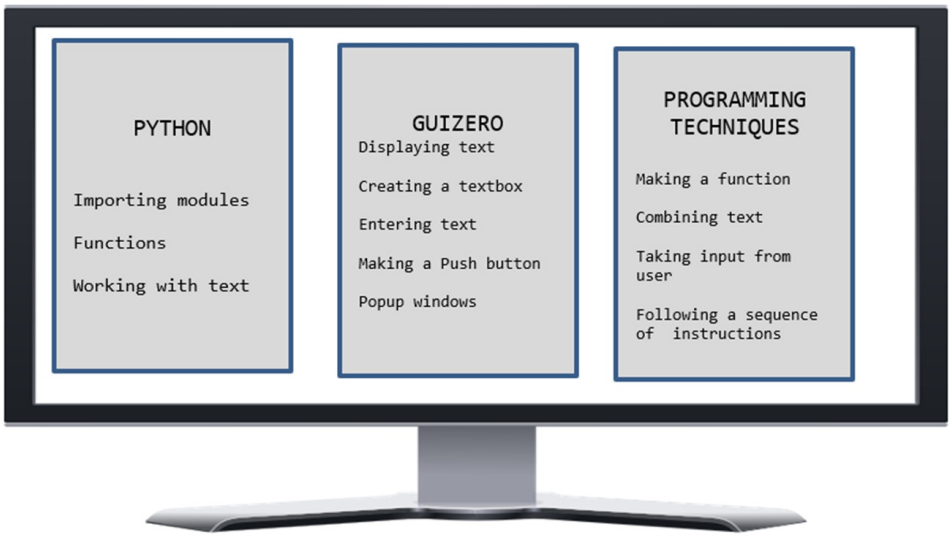
**Figure 1.3 How we will structure the GUI program.**




# Hello GUI: Coding your first GUI

Now for the fun to begin! This section of the chapter walks you through creating your very first GUI program so that you learn all the good stuff.

**Figure 1.4 Project uses the following programming skills, are we still including this in the chapters?**




Earlier in this chapter I referred to the Hello World program, which is considered the beginner's program in any computer programming language. Traditionally, the program simply prints out the words hello world. It takes no inputs and has no outputs.

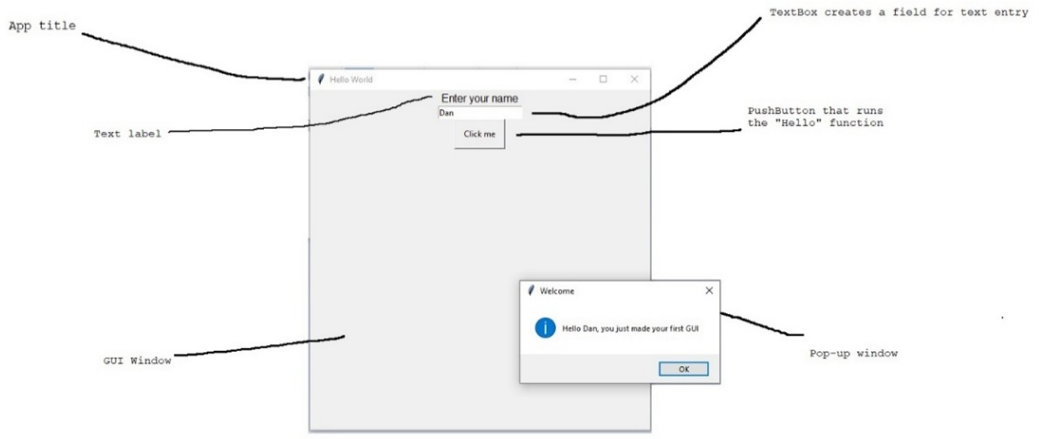
#### Inputs and Outputs

An input is defined as one of two things. Firstly, an input can be a device that is used to control the computer; for example, a mouse, a keyboard, and even a gamepad is an input you can use to control your computer. The second definition of an input is data or information that is put into program. One example is when you log onto your computer, and it requires you to enter your username and your password. You input your username and your password into the software. Output refers to a device that is used to display data or information. The most common output is a screen or monitor which displays what a program is creating that is supposed to be visible; that's called a program's output.

In this version of the hello world program, we are going to add an input in the form of a text box that will enable you to enter your name or any name that you want to. The GUI also features a Button that, when pressed, tells the

program to display a pop-up window that will say “hello” followed by your name, as shown in figure 1.5.

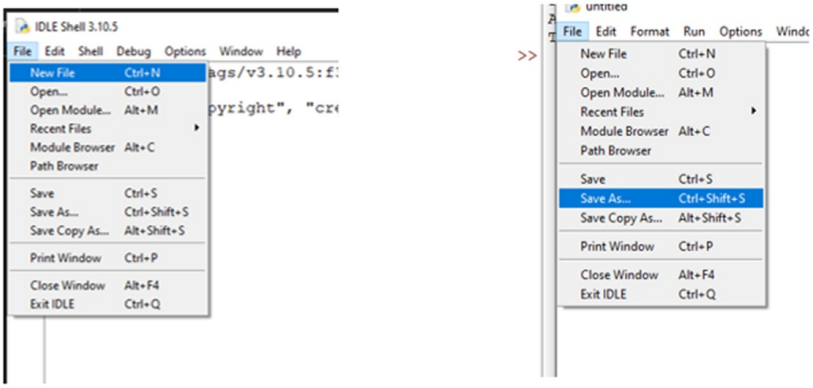
**Figure 1.5 Your first GUI**




To begin writing your program, first load Python.

1. Assuming that you followed the appendix A installation of Python, then go to the Taskbar and select the Python icon, or open the windows menu and search for Python, typing in the word, ‘Python’. Click to load Python.
2. Create a new program by choosing the **File** > **New File** option. You should now save your program; do so. Saving your files right away is good practice, just in case the computer crashes and you lose your file and have to start all over again.
3. If you have installed guizero using the installation instructions in this chapter, then you can simply save the Python file. (However, if you have downloaded the guizero files for the quick start appendix B, then you must save the file **HelloGUI.py** into the same folder as you download, see figure 1.6.)
4. Save the file by choosing the option **File** > **Save As** from the menu.
5. The Python editor will prompt you to save the file.
6. Name the program **HelloGUI.py** and choose **Save**.

**Figure 1.6 Save your Python file**




## Importing the modules

Each GUI program that you write first requires you to import the required modules. You can think of modules as smaller programs that allow you to add and use different features in the program. Both guizero and Python contain modules, so you will be using modules from both. For example, in this project we will be adding a pressable button to your GUI. (Figure 1.5). To do this, you import the `PushButton` module, which contains all the lines of program code required to instruct Python how to create a button. You do this at the start of your program by simply using the code, `from guizero import PushButton`

#### Guizero modules

Guizero stores the modules in a folder called **guizero**. If you are interested, you can open this web link https://github.com/lawsie/guizero/tree/master/guizero and see the modules and code for yourself. You will notice that the name of the file is the same name of the module that you use when importing them. For example, the PushButton module is named PushButton.py.

The module enables you to create the widgets (appendix D) that we can then use in the program. Modules save us time and complexity. For example, the PushButton module contains prewritten code to program features of the buttons such as, the size of the button, the color of the button, what the button does when pressed, the amount of padding around the button and more.

Lucky for us we can simply use the `PushButton()`function which is a single word. Functions are so useful and efficient. You can find a list of all the Elements and Widgets and what they do in appendix D.

In your Python program file, type the code shown in listing 1.1.

**Listing 1.1 Import the modules**

```
# Imports

from guizero import App, info, PushButton, TextBox, Text
```

The program begins with the hashtag symbol `#` used on `# Imports` is a comment. Comments are useful for telling the reader what your program code does. It is useful for organizing and managing the sections of your program.

**Comments**

A comment is a reminder or notice that tells you something about the code. A little bit like a keep off the grass sign, it tells you what to do and which grass to keep off! Comments start with the # symbol indicating something that the program code will ignore or not execute.

The next line of the program begins with the code,

```
from guizero import
```

which tells your Python program to go to the guizero library and then import the modules listed in the same line. The modules used in the HelloWorld GUI program and each of their purposes are,

- `App`, used to create the main window that holds all the widgets
- `info`, this displays a pop-up message window
- `PushButton`, this creates a button that can be clicked
- `TextBox`, is used to enable the user to enter text which can be used by the program
- `Text`, displays the title of the GUI

## Creating the variables

The next stage in writing your program is to create the variables. A variable is a location in the computer's memory that stores data or information. The data is only stored whilst the program is using it. When you close the program, the data is forgotten. The content or value stored in the variable can change; that is, it can vary.

Programs need to be able to find data quickly, so each variable you create must be given a unique label. Think about how food in your house may be in a jar or a box which has a label (figure 1.7). The label helps you to identify what food is stored in the jar.

**Figure 1.7 Labels on Jars are like variables; they tell you what is stored in each Jar**


In a similar way, labeling a variable means the program can find the memory location quickly and go straight there and collect the data.

You can label a variable almost anything that you want; however, it's good practice to use a name that represents what data is being stored in the variable. For example, if the variable was storing the name of a person, then you might label it `persons_name`. When creating the label for the variable, it has no spaces between the words. Should you use spaces, then the variable won't work, and you'll get an error message.

**Constants**

The opposite of a variable is a constant. Guess what, constants stay the same and do *not* change. For example, Christmas Day is always on 25 December. That makes it a constant. The present or gift that you get each year, is a variable as it changes each year, unless you are over 50 years old, and then you always get given socks!

In this program we are using some variables to create the widgets, the text, the textbox, and the button. The code for these widgets is written in the #App section of the program, so it makes sense to store these variables in the #App section of the program. Although it will remain empty for this program, add the `#Variables` comment to the program, as we will be using variables in the chapter 3 and it is good practice.

## Creating a function to display the pop-up message

The next section of the program requires you to create a function named Hello, that you will assign to a button in the GUI. When the button is pressed, it will call the function named Hello, then run the code contained in the function, to combine your name with a short message and display the name and message in a pop-up window. This action happens every time that you press the button.

A function is a reusable block of code that can be called and used at any point in your program. Imagine you are playing your favorite computer game and you are down to your last life; you can die or lose a life at any point in the game. So potentially, there are millions of points during the game where it's game over and the game would end. If the game programmers didn't use a function, they would have to program the "game over" code millions of times, once for every scenario where you lose your last life. This would be very tedious and make the program's code massive. Instead, developers use a flag, which is triggered when the game is over. The flag informs the program to run the game over function.

Using a function, listing 1.2 means the developers can write the "game over" code once and then when the game ends, the program calls the function, and it runs. You only need to write the block of code once, then you can use it as many times as you like, which saves processing power and makes the

program more efficient.

**Listing 1.2 Function code**

```
# Functions

def Hello():
    info("Welcome", "Hello " + textbox.value + ", you just made y
```

The line of code in listing 1.2 looks long and has a lot of elements, so let's break it down. To create a function in Python, you use the keyword `def`, which indicates to the Python program that it should expect a function to follow. Then you give the function a name and type it in. In our program, the function is called `Hello()`. The name `Hello` is used to call the function, calling means that the program runs the function when a particular action is performed by the user. In this program the function is called when a button is pressed.

Do you remember that when we were labeling a variable, we used a suitable name so that it helped us to identify what the variable held? Well, the same naming applies here; the function's name should be related to the function's purpose; that is, the name should indicate what the function does.

Using suitable function names will also help other programmers that look at your code, understand how the program works. It also makes error checking easier. For example, if the error is in the function, Python will inform you of the error and you can easily find the relevant area of your code. You can find the function in your code without having to check every line for the error. Python will inform you where it found the error; it will display a message something along the lines of

```
SyntaxError: invalid syntax. Perhaps you forgot a comma in the fu
```

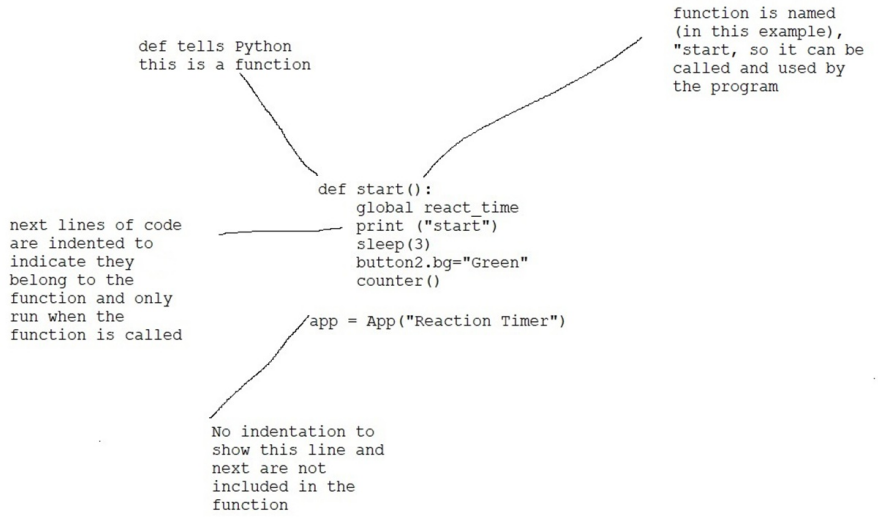
A syntax error indicates that the way the code has been written does not follow the required Python structure rules. In some ways, syntax is like grammar. For example, Thor from the Avengers owns a hammer, which only he and several other characters have ever been worthy enough to lift. (Do you know who they are?) The correct grammar or syntax for referring to that hammer is "Thor's Hammer." That is, you use an apostrophe to indicate that

the hammer belongs to Thor. The incorrect syntax is Thors Hammer, which implies that there is more than one Thor. You can read more about types of errors in Python code in appendix E.

A function ends with `():` which is the syntax that Python uses to indicate that anything in the next section of the code belongs to the function. You may have noticed that the line of code under the function name is slightly indented to the right. To get the amount of indentation correct you can press the TAB key once or the spacebar four times. Ensure that you always use the same amount of indentation throughout your program.

By indenting the lines of code (see figure 1.8), you tell the program which lines to include within the function, and to only include these lines. So how does the function know what the last line of code in the function is?

**Figure 1.8 Parts of the Function**

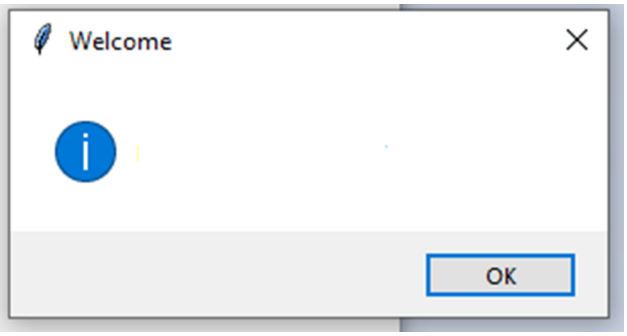



The last line of code and the end of the function is indicated when the next fully **left-aligned** line of code appears in the program. Now that you have created the function and you know what it does and why we are using it, you can add the function's contents. The first part

```
info()
```

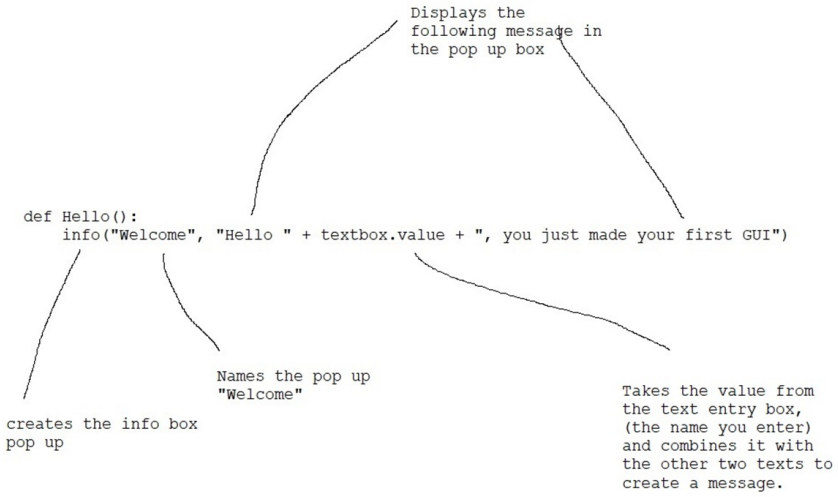
tells Python to create and display an information message box (figure 1.9).

**Figure 1.9 An information pop-up**




The information box pops up when you press the “click me” button (which we will code in the next section) and combines the name that you enter into the text box and the string, “Hello” to create a final message (figure 1.10).

**Figure 1.10 The parts of a Function**




A string is a series of characters. A character can be a letter, number, or symbol. For example, “this sentence is a string”, “Hello World” is also a string. In Python, strings start and end with the `"` symbol, like this: `"This is a string."` Strings are used to hold content that is going to be displayed or printed out in the program. For example, consider the web banner on Amazon’s homepage. All the options headings will be saved as strings (figure 1.11). Most commonly, strings are printed or saved in a variable, I’ll

talk more about this in appendix E.

**Figure 1.11 Strings used on a web banner**


Digits and numbers can also be a string, for example, “C3P0” is a string. Now you might say that this is the name of the droid that you are looking for, therefore it uses letters and numbers, similar to a car registration plate or your zip / post code. A car license plate or a postal code are both created with letters and numbers. However, a list of digits or a number can be a string; for example, this number “221” is a string because the numbers are enclosed within quotes. However, the number has no value. It is not two hundred and twenty-one, it is simply the symbols 2, 2, and 1.

If you look again at code listing 1.2, you will see the strings being added together,

```
"Hello " + textbox.value + ", you just made your first GUI")
```

We know that it is impossible to add text in the mathematical sense, so this line of code uses the + symbol to combine the two strings and the name that you input, into a longer string, a full sentence that will be displayed.

## Creating the GUI window and outputting the message

The last part of the program is to add the lines of code that will build (create) the GUI and the interaction between the parts: the textbox, pushbutton, and the pop-up window. You may remember we introduced variables and noted that the program uses several variables as discussed in the previous section. As a reminder, a variable is a location in the computer’s memory that stores data. In this Hello World program, there are four variables.

- The first variable labelled, `app`, creates the main window of your GUI.
- The second, `text`, holds the instruction that tells you what to do (the

instruction tells you to enter your name).

- The third variable, `textbox`, stores the data to create the text box and also stores the name that you enter into the text box.
- The final variable, `button`, holds the information about the button.

Add the code in listing 1.3 to your program.

**Listing 1.3 Creating the main GUI window**

```
# App

app = App("Hello World")
text = Text(app, text="Enter your name")
textbox = TextBox(app, width=20)
button = PushButton(app, command=Hello, text="Click me")
app.display()
```

The first line of the code creates the GUI window and positions the name "Hello World" at the top left of the window. You can name the GUI window differently if you wish; just ensure that you enclose the name in quotes because, yes, you have guessed it, this is a string. The `App` is the main window which contains all the other widgets. You will notice in line two that the word `app` also appears:

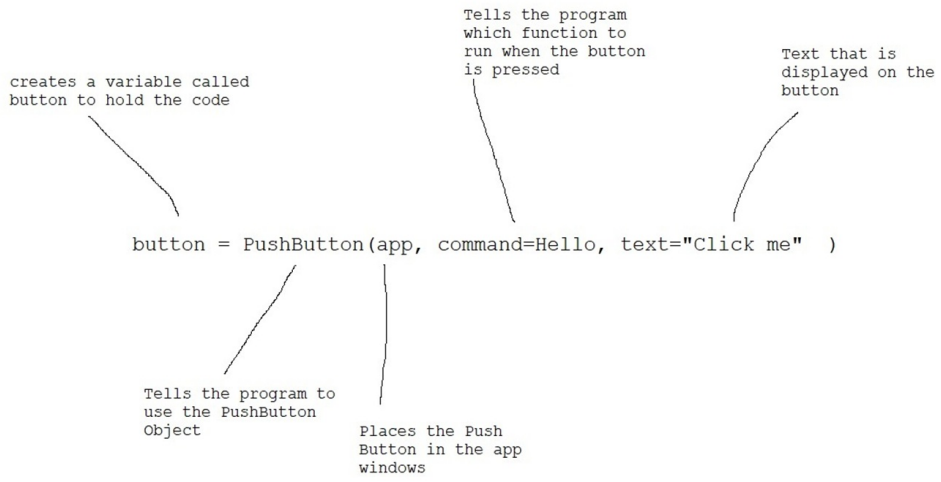
```
Text(app, text="Enter your name")
```

This is because we use the `app` object to edit and add content to the main GUI window. (There is also a `Box` object that works in a similar way to the `App`; you can add and edit content. A box is usually held within an App. You will come across boxes in chapter 3 when you make a FART soundboard!) An object can be thought of as a way to define the properties and features, so the app object enables you to set the properties, say the size and color of the app window.

The second line creates and displays a line of text with the instruction to "Enter your name". Notice that it uses `app` after the `Text()`. This tells the program where to display the text; in this case, to display the text in the App window. The third line is responsible for creating the text box where you enter your name.

Next, on line 4, add the code to create the button. You need to tell guizero which function you want the button call, (which function to run) when it is clicked (figure 1.12). Notice here that, confusingly, the function is called a command. When you create a function like you did in section 1.2.3, it is called a function. When you write code in guizero to call or run this function then you use the code, `command =` . Finally, you need to tell the program to display all the elements that you have just coded. To do so, simply add the line `app.display()`to the end of the program.

**Figure 1.12 The PushButton code**



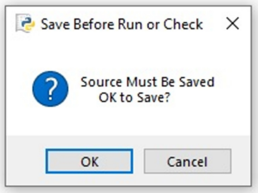

## Running and testing your first GUI program

You just built your first GUI. Well done! Now to test that it works correctly. Do not worry if the program fails the first time. If you are new to programming or even experienced, then it is likely that you made an error. Such errors are usually syntax errors, as explained under listing 1.2. A syntax error is like a grammar error; Python expects you to use certain syntax for the program to work correctly.

So, let's try to run the program. Press **F5** on the keyboard. (If your keyboard does not have function keys then, at the top of the IDLE window click, **Run** and select **Run Module** from the drop down menu. You may be prompted to

save the program file (figure 1.13). Choose **OK** and the program will save, and then the program will run. Fingers crossed!

**Figure 1.13 The Python save window**




If your program fails or does not run as expected check for these common errors:

1. Have you installed guizero correctly?
2. Did you save the program into the GUI folder?
3. Do all the comment headings start with a # and appear in red font, `# Imports, # Variables, # Functions, # Apps?`
4. Are all the required modules in the import section?
5. Have you used the correct case? Some syntax calls for capital letters in the middle of a word, for example, `TextBox`, `PushButton`.
6. Have you used `""` for each string?
7. Did you spell all the words correctly?
8. Check out the final code listing and compare it with your program code.

**Listing 1.4 Final code listing**

```
# Imports

from guizero import App, info, PushButton, TextBox, Text

# Variables


# Functions

def Hello():
    [CA] info("Welcome", "Hello " + textbox.value + ", you just m

# App
```

```
app = App("Hello World")
text = Text(app, text="Enter your name")
textbox = TextBox(app, width=20)
button = PushButton(app, command=Hello, text="Click me")
app.display()
```

# Three other things to try

This section of the chapter introduces three extra things or edits for you to add to your program. These ‘extra things’ can be used to personalize your GUI, and can make your GUI look and feel quite different. You will come across different changes throughout the chapters and you can try them out with any of your projects.

## Changing the background color

The first change to make to the program is to change the color of the background of the GUI window. Color can add a different feel to your projects. The line of code that adds color is,

```
bg = "red"
```

where `bg` is short for background. The line of code is saying “make the background color equal to red.” In the App section of the program, locate the first line where you named the GUI, "Hello World". Add the additional code to the line, `bg = "red"`.

```
app = App("Hello World", bg ="red")
```

Press F5 on the keyboard to save and run the program. The background will now be red. There are many colors that you can choose from. Check out this website, https://wiki.tcl-lang.org/page/Color+Names%2C+running%2C+all+screens, which lists the names of each of the colors. Try some out.

## Resizing the text box

Secondly you can change the width of the textbox. Changing the width is

useful if you have a longer name and need more space, or if you have a shorter name and need less space. To change the width of the text box, add the code `width=40` to the TextBox line of code. This is the line before the PushButton line. Try experimenting with the width until you find a suitable size.

```
textbox = TextBox(app, width=40)
```

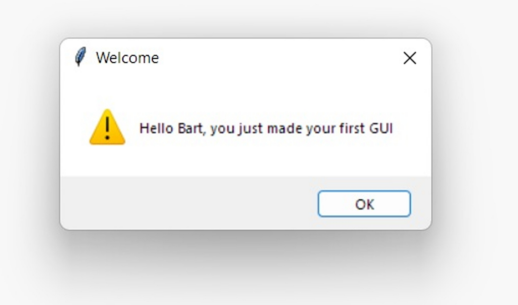
Press F5 on the keyboard to save and run the program.

## Using a warn window instead of the info window

In the last change to the program, edit the function so that a warn window is displayed instead of the info window (figure 1.14). The warn pop-up window has an image of a yellow warning sign. It is often used in GUIs to grab the user's attention and warn them that an event is about to happen. To make this edit, change the word `info` to `warn` on the line of code held in the Function,

```
def Hello():
    app.warn("Welcome", "Hello " + textbox.value + ", you just ma
```

**Figure 1.14 The warn window pop-up**




Press F5 on the keyboard to save and run the program.

**Listing 1.5 Final code listing after three things to try**

```
# Imports

from guizero import App, info, PushButton, TextBox, Text

# Variables
```

```
# Functions

def Hello():
    [CA] app.warn("Welcome", "Hello " + textbox.value + ", you ju

# App

app = App("Hello World", bg = "red")
text = Text(app, text="Enter your name")
textbox = TextBox(app, width=40)
button = PushButton(app, command=Hello, text="Click me", )
app.display()
```

# Conclusions

In this chapter you have created your first GUI. This project is important as it has introduced you to some of guizero's basic features. In this chapter you were introduced to the programming language Python. This language is a popular language amongst developers and can be used to create websites, games, perform data analysis and much more. You can download the chapter code here:

https://github.com/TeCoEd/Twisted_Python_Projects/tree/main/Project_Code

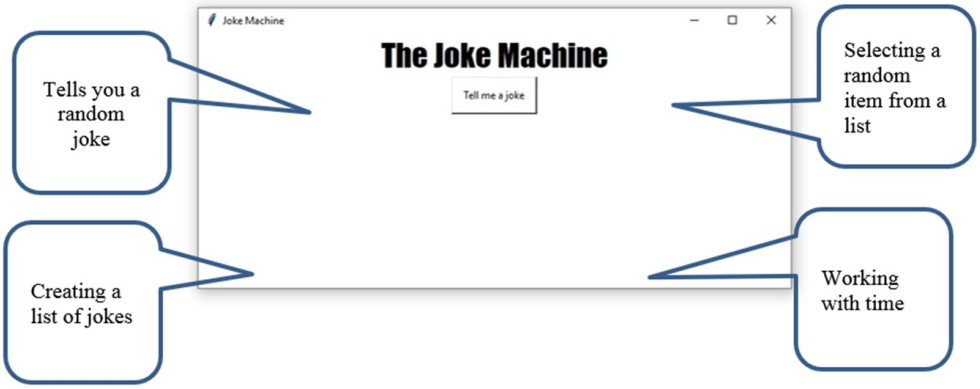
If you are new to Python, you can read more about the features of the language in appendix E before you start your next project. Or just jump straight in. Have fun!

# Statements of fact

- Guizero is a package that is installed and enables programmers to easily create GUIs using the Python programming language.
- Widgets are the guizero name for the elements or items that appear on the GUI. Widgets enable you to interact with the GUI.
- Functions are reusable blocks of code that can be called and used anywhere within a program. Functions save time and make the program more efficient.

- A module is a library of code that contains a set of pre-built functions. Modules are imported at the start of a program.
- A variable is a location in the computer’s memory which stores information and data.
- A string consists of either text, numbers or characters enclosed within quotation marks. Strings can be stored and combined together to create a longer string. Numbers that are strings are valueless.
- Popups are message boxes that appear on your screen. Programmers use pop-ups to display information.
- The App windows holds all the elements and widgets of the GUI.
- A PushButton is a pressable button. Programmers assign functions to a button; when the button is pressed, the program calls and runs the function.

# 2 Are you funnier than a hyena?




## This chapter covers

- Creating a joke machine
- Creating a list of jokes for the joke machine
- Creating a GUI for the joke machine
- Using a button to display a random joke from the list

There is a well-known saying that “Laughter is the best medicine.” The original quote is not quite as catchy: “A merry heart doeth good like a medicine.” The point of the quote is still the same; laughing makes you feel good. It can make you feel happy and make you feel better. If a good story or joke made you laugh, you are likely to share the joke with others and spread the good medicine!

### What you will build in this chapter

In this chapter, since laughter is the best medicine, you will build a Joke Machine (figure 2.1) to spread laughter all around. The Joke Machine consists of a simple button which, when pressed, will display a random joke. Don’t like the joke? Not a problem. Simply press the button again and a new joke will be displayed. As people tell you or you hear more jokes, you can collect them and add them to your Joke Machine so that it always makes

people laugh. You will need four jokes to start with for this project, you can use the ones in this chapter and if you hear any good ones, write them down so that you can use them in this project.

**Figure 2.1 The Joke Machine GUI**


While building the project you will learn the skills shown in table 2.1, including the random function, which is used to select a random item from a list. Choosing random items is a common feature of games such as Minecraft where the game world is randomly generated. A staple method of most user interaction with a device, from mobile phones, smart TVs even washing machines is via buttons. This project covers how to set up and add interaction to a push button, this is a visual button in the GUI that you interact with to make something happen. You press the button, and the joke is displayed on the screen. Text is used everywhere, and this project covers how to display and edit text.

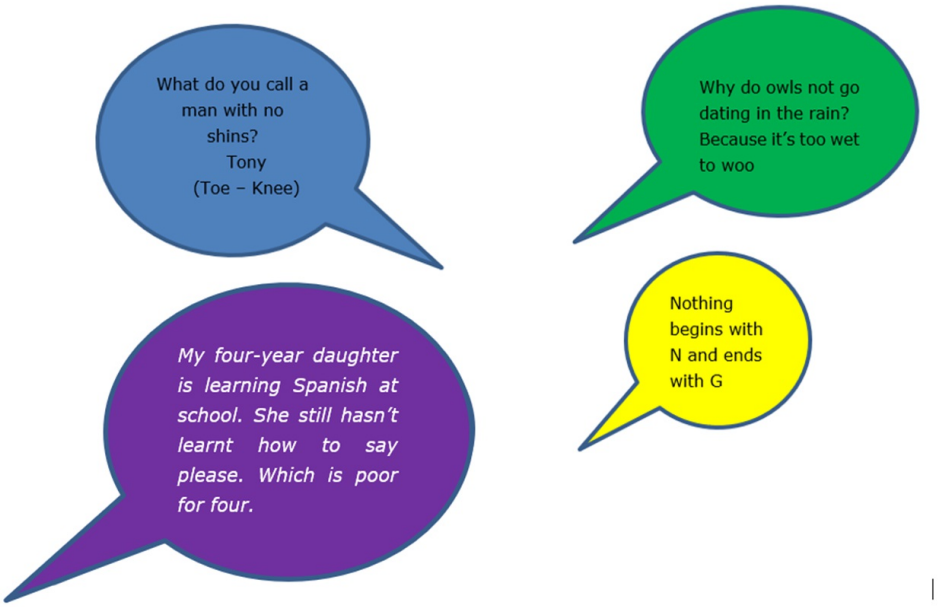
**Table 2.1 Programming skills taught in this chapter.**

| Python | guizero | Programming techniques |
|---|---|---|
| Random module<br>Variables<br>Lists<br>Strings<br>Functions | Displaying text<br>Push button<br>Resizing the window<br>Looping a program | Storing data<br>Displaying data<br>Searching for data<br>Iteration |

# Creating the project

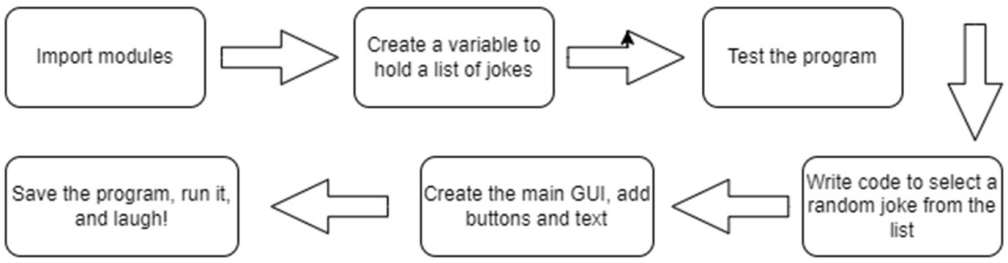
Before you start the project, you may want to gather a number of jokes that you can use in your Joke Machine. Or you can use the example jokes in the code and then replace them with yours later.

Here are some jokes to get you started.




Let's begin to build the Joke Machine. The project consists of several steps, which are shown in figure 2.2. Follow the steps in order.

**Figure 2.2 Overview of the project**




# Importing the modules

The first step of the joke program is to import the modules that the program requires. You may remember from previous chapter that modules contain prewritten functions that perform different tasks, such as displaying text, displaying images, and much more. In this context, the modules provide all the code needed to create the various elements and widgets of your GUI. Prewritten functions save you time as you don't have to create them from scratch.

You are also going to create a list. Lists are a useful method of holding more than one item in a single variable. We will go into more detail about lists later in this section. For this program, you will use the list to store the jokes and assign them to a variable.

## Creating your new program

Open a new Python program file in your editor. Refer to chapter 1 and appendix A for a reminder about editors. From the **File** menu,

1. Choose **File** > **New File**. Your Python editor will prompt you to save the file.
2. Save the file with the name **Joke_Machine.py** and choose **Save**.

**Remember**

(as with the previous projects, if you used the guizero quick install method from Appendix C then you must ensure that you save the file into the same folder as the guizero.)

## Importing the needed modules

Begin the Joke Machine program code by importing the required modules (listing 2.1) from the guizero library. (You installed this library in chapter 1.) The modules provide all the code you need to create the various elements and widgets of your GUI.

**Listing 2.1 import the modules**

```
# Imports

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

In this program, you are importing the following guizero modules:

- App, to build the app window,
- Text, to enable the program to display the Jokes,
- PushButton, to create a button that displays a joke when pressed.

This program also uses the `random` module which is imported from the Python program. This module is used to select a random joke from the list. This is instead of always displaying the jokes in the same order each time the program runs. It would be unfunny if you always already knew the joke and its punchline.

## Creating a list of jokes

Next, you need to create a list to store your jokes. You use lists all the time. Think of a to do list, a list of friends or a shopping list (figure 2.3)—these are both examples of lists that contain all the things you must do or, in the case of a shopping list, all the items that you need to buy. You can read the list, cross items off, and even add to the list.

**Figure 2.3 A shopping list is a list of items.**

## Creating your list of jokes

Your list of jokes is stored in a variable with the label `list_of_jokes`. You may recall from chapter 1 that the label cannot have spaces in between the words, but it's hard to read words when they're crammed together without spaces. To solve this, you can join the words using underbars (_). The variable `list_of_jokes` is easier to read than `listofjokes`.

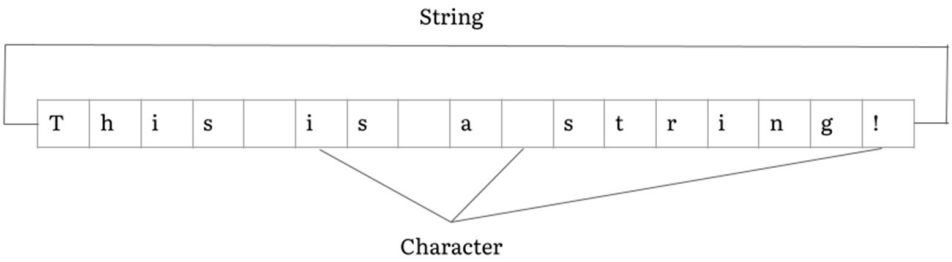
A variable is a location in the computer's memory that stores the jokes. The label or name of the variable references the location in the memory location so the information can be accessed quickly by the program. The Joke Machine program needs to be able to find the memory location where the jokes are stored, so you name or label the location so the program can quickly go straight there and collect your jokes. Write up and add the code below in listing 2.2, underneath your imports.

**Listing 2.2 Creating a list to hold the jokes**

```
# Variables

list_of_jokes = ["What do you call a man with no shins? Tony (Toe
                 "Nothing begins with N and ends with G",
                 "What kind of tree can fit in one hand? Palm tre
                “[CA]”"Why do owls not go dating in the rain? Bec

print (list_of_jokes)
```

Each joke in the list is entered as a string. A string is a series of characters. A character can be a letter, number, or symbol, figure 2.4. For example, the letters in this sentence are characters. In Python, strings start and end with the `"` symbol, like this: `"This is a string!"` Each joke is a string, and a comma separates each joke, apart from the last joke, where the comma is optional.

**Figure 2.4 Anatomy of a string**

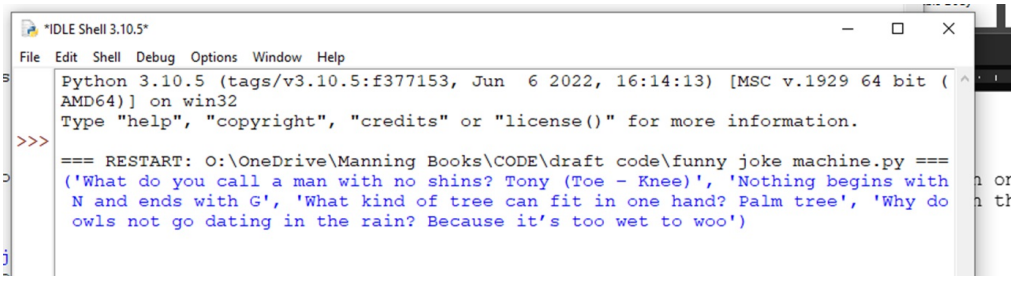



The last joke in the list ends with a bracket to close the list, `"Why do owls not go dating in the rain? Because it's too wet to woo"].` You can continue to add any number of jokes to the list, but start with a few to test that the program works correctly. For now, just add four jokes; using a small number of jokes will make finding errors and testing easier.

## Running your program to test if it works

On the last line, the code `print(list_of_jokes)` is used to test that your program is working correctly. Without the final GUI window, this is good developer practice, testing parts of your program and small sections of the code to catch any problems early on. Press **F5** on the keyboard to save and run your program; you should see all your jokes displayed in a long line in the window of the Python editor, as shown in figure 2.5. When you have tested the program, change the last line of code to `#print (list_of_jokes)`, this will stop it printing out all the jokes.

**Figure 2.5 Output of the test**




## Troubleshooting your program

Programs don’t always run perfectly the first time. Programmers are used to this. To find and fix the problems that made their code run poorly, programmers use a process called troubleshooting. The first step of troubleshooting is to check for common errors. Even the best programmers make them!

If your program fails or does not run as expected, check for these common errors:

1. Have you used quotation marks `[""]` around each string?
2. Is the closing square bracket, the `]` present at the end of the list?
3. Did you label the variable `list_of_jokes` and not `list of jokes`?
4. Did you save the program into the GUI folder as shown in chapter 1?

## Creating a function to display a random joke

Now you are ready to take the next step: writing the function that will select a random joke from your list of jokes.

#### Why create a Function?

Imagine you are playing your favorite computer game and you are down to your last life; you can lose your final life at any point in the game. So potentially, there are millions of points during the game where it’s game over. Most games contain a “game over” animation where the game character dies, faints, falls over, melts, dissolves, and even flashes. If the game developers didn’t use a function, they would have to program the ”game over” animation code millions of times, once for every possible scenario where you lose your last life. This would be very tedious and make the program code massive.

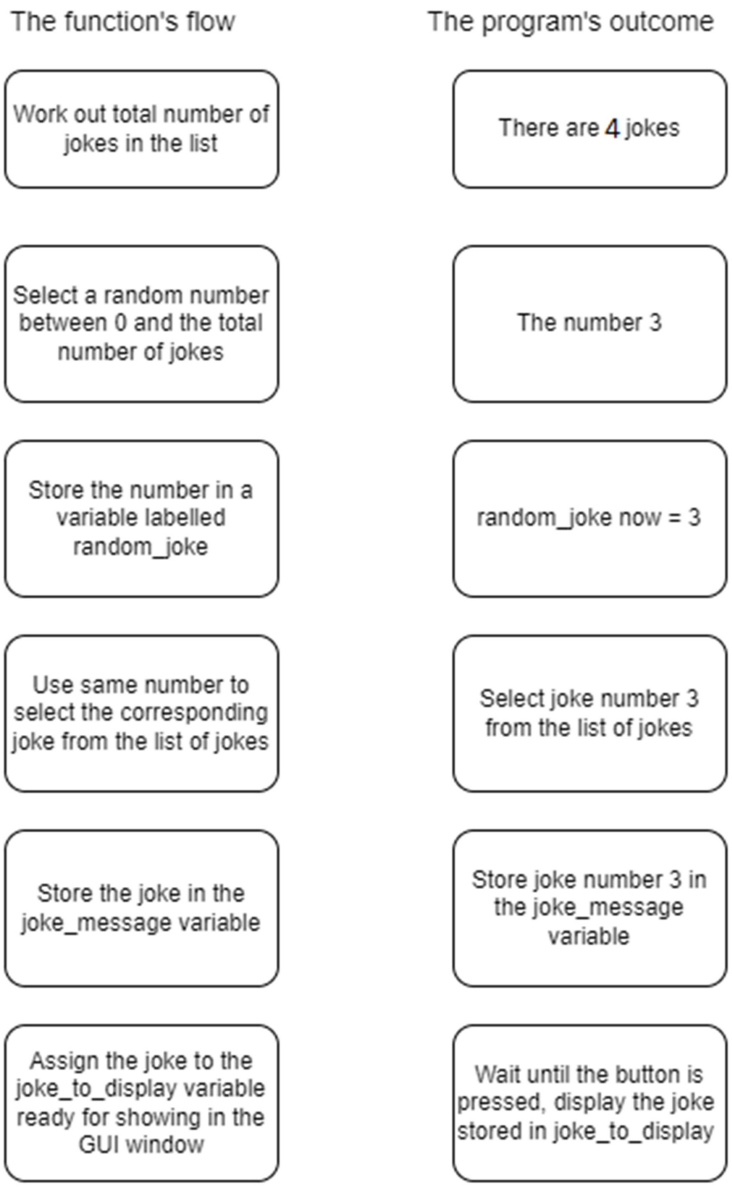
You only need to write the block of code once (figure 2.6), then you can use it as many times as you like, which saves processing power and makes the program more efficient.

**Figure 2.6 Creating a function to select a random joke from the list**


__

The function in the Joke Machine program will be assigned to a button in the GUI, which when pressed, will call the function and then run the code to select a random joke. This happens every time the button is pressed. Figure 2.7 shows the flow of the function on the left side, and on the right, what the outcome is in the program.

**Figure 2.7 Flow of the function that selects the joke**




Add the lines of code in listing 2.3 to create the function on the next line of your program.

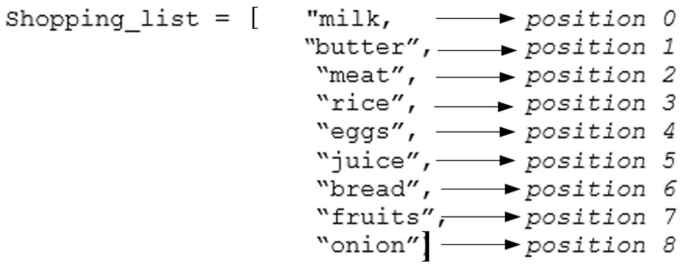
**Listing 2.3 Create a function to select a random joke**

```
def select_joke(): #A

    random_joke = random.randrange(0,len(list_of_jokes)) #B
    joke_message = list_of_jokes[random_joke]
    joke_to_display.value = joke_message #C
```

A function begins with the text `def`, which stands for define. This tells Python that you are creating or defining a new function. Then you must name the function so that you can call it by its name and reuse it. (If you have a dog, it has a name; usually when you call their name, they come up to you! Not all the time. Likewise with cats! But functions must come every time that you call them) The function in the Joke Machine program is called `select_joke():` as its purpose is to select a joke from the list. Note the use of the `():` at the end of the function, you must use this syntax, so Python knows that you are creating a function.

The next line of code selects a random item. In this program, the random item is a random joke from the list. Consider the image in figure 2.8 of a shopping list (in the style of a Python list) and number each of the items starting from zero to the end. The first item, milk, would be numbered 0, then butter numbered, 1, then meat 2, rice 3, and so on. The line of code uses `random.randrange` which means a random selection within the range, and you may know from maths that a range is from the first number to the last number in a list. You can read more about the random module on the official Python website, (https://docs.python.org/3/library/random.html)

**Figure 2.8 A shopping list as a Python list**




In the example of the shopping list, our first item on the list is milk and the last item, an onion. The position of the onion is eighth, so the range of the shopping list, is 0 to 8. Now, you need to tell the program what the range of the list of jokes is so that it knows the total number of jokes in the list and can select a position that exists in the list. (Basically positions 9 and above do not exist and the program would fail). Finding the start of the list is simple as it starts at zero, it starts at the beginning. You then set the end of the range as three since we have 4 jokes and joke 0, joke 1, joke 2 and joke 3 will give us

four positions to hold each individual joke.

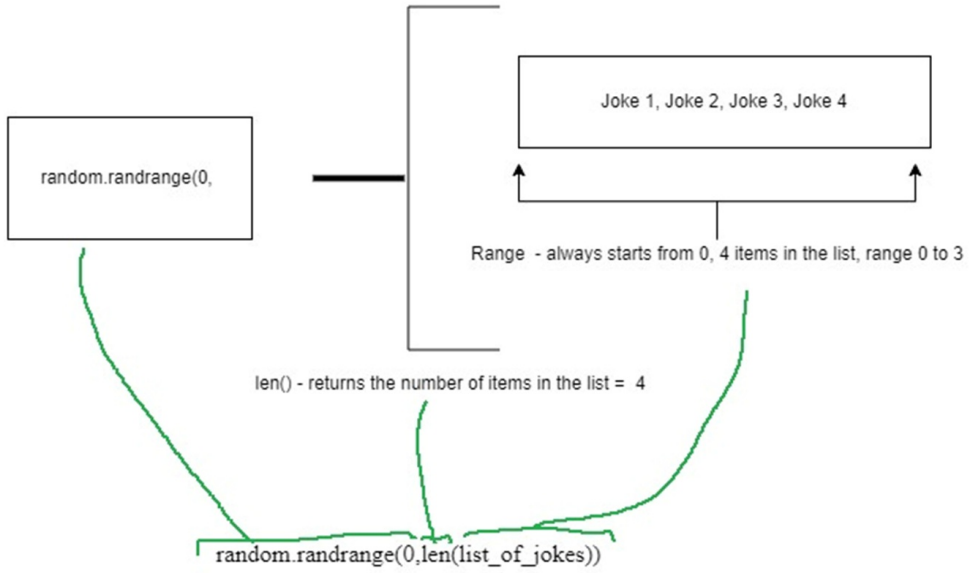
**Why start from zero?**

Counting positions start from 0 because Martin Richards, creator of the BCPL language, designed lists to start from 0 as the natural position.

However, what happens if you add more jokes? That would mean you would have to change the range. For example, if you add 8 jokes you would need to change the end of the range to 7. If you forget to do this, the program will never display the new jokes as it is unaware of their existence! (figure 2.9).

The solution is to instead, use the code `len(list_of_jokes)` which finds and returns the length of a list with any number of items stored in it. The `len()` function, short for length, enables the program to find the length or the end of the variable called list of jokes that we created in listing 2.2. So, this line of code is basically looking up how many jokes you have stored in the list and then returning the end value of the range.

**Figure 2.9 How the program selects a random number from a list of any length**




Since the program knows the length of the list, it can then use the `random.randrange` to select a random joke from your list of jokes. This is a

prewritten function from the Python random module included within Python. The function consists of code that can select a random number from a range of numbers defined by the user, in the similar way that you can roll a standard die and it will land on one of six numbers.

Note that at this stage the program is only returning the position of the item, not the actual joke. This position value is then stored in a variable named `random_joke`.

So now the variable `random_joke` holds a random position number from the list of jokes (note that this is just the number of the item and not the physical text of the joke). In order to extract or collect the joke that will be displayed, use the code

```
joke_message = list_of_jokes[random_joke]
```

which creates a new variable named `joke_message`. This variable selects a joke (the item number) from the list of jokes, returns the joke as a string, and saves it into the variable so it is ready to be used in the program, figure 2.10.

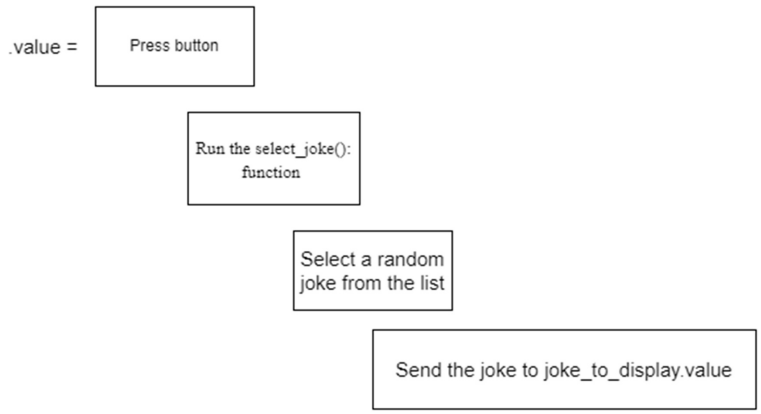
**Figure 2.10 Selecting a random joke from the list of jokes**


The last line of code assigns the joke, as a string, to a new variable `joke_to_display.value = joke_message` so that the computer can pass the joke to the GUI and display it. The term “passing” describes the process of taking the joke that is stored in the computer’s memory and making it available for the program to display it as text in the GUI window. You will see that this code uses `.value` at the end of the variable, this tells the program that the data the variable holds will be updated with a value (a joke). You will

see how this works later after listing 2.4.

The `.value` code is responsible for replacing the existing joke. What this means is that each time the program calls the `select_joke()` function, it *overwrites* the current joke with a new joke. A different joke is displayed.

**Figure 2.11 How the .value code works**




If the value was not updated, then all the previous jokes would still be displayed, and there would be a long list of jokes cluttering up the GUI window. Note that this variable, `joke_to_display.value`, currently has no value as the program has not run and the function has not been called. Therefore this variable is empty.

## Creating the GUI

Now you are ready to create the App section of your program (listing 2.4). The App section brings together all the features into a GUI.

The first line of code names your GUI. The name will appear at the top left of the GUI window. Then you can set the GUI's height and width. You may need to change these values depending on the length of your jokes. If you have chosen longer jokes, then you may need the GUI window to be wider.

Next, we set the background color of the GUI to white. You can change the color by replacing the word White with the color that you want the background to change to. Simply replace White with the word Green, to change the background to the color green.

Then display the title of the program in your GUI and set the size of the text. You can change this if you want to call your Joke Machine something else, like “Ha Joke Teller!” Simply replace the text but remember to use the " ".

Finally, set the font (what the text looks like) to Impact. You do this using this line of code:

```
title_text.font = "Impact"
```

You put the name of the font you want use between the quotation marks. The fonts that you can use depends on which Operating System you are using and, which additional fonts you have installed.

Add the code from listing 2.4 to your program.

**Listing 2.4 Set up the GUI**

```
# App


app = App("Joke Machine", width=900, height=300) #A
app.bg = "White" #B
title_text = Text(app, "The Joke Machine") #C
title_text.text_size = 28 #D
title_text.font = "Impact" #E
```

## Adding a control button

The last part of your program code adds the button, which, when pressed, will run the function from listing 2.3. When the function runs, it displays a random joke from the list.

Start by creating a variable which will hold the code instructions to create the button. In listing 2.5, the `command=` tells the program which function to run—the function that you created. The code basically says that when the button is pressed, run the function called `select_joke`.

Lastly, add the text that you want to display on the button. Something simple like “Tell me a joke” provides the user with an instruction about what the button does when it is pressed.

If you look at listing 2.3, you see the last line of code uses the variable `joke_to_display.value`. This variable holds the joke that is going to be displayed. However, the program needs to firstly be expecting a joke.

Imagine you invite six friends round to your house, and an extra friend turns up. So, there are now seven friends, not six. You were not expecting them, so you don't have a chair for them to sit on. Now as a kind human you would share or maybe give up your chair for the extra person. However, alas computers are machines and would not.

They follow exactly what the program instructions ask them to do, computers cannot adapt to deal with unexpected or additional requirements. So, in the computer world, if a seventh friend turned up and the computer was expecting six, then this would cause an error as the program was not expecting it.

In the program we can create a list of six jokes, but creating the jokes is not the same as telling the program to expect a joke. You have to program in the expectation!

To solve this issue, we create the `joke_to_display` variable, and use the `Text` command so that the program knows that it will be holding a string, because it is going to display the joke as text. Next set the text to empty/blank using two quotation marks with nothing in between them, like this "". This means that when the program runs, the variable is storing an empty string. The line of code is basically getting a chair ready for when your friend arrives. They are not seated in the chair until they arrive; however, you still get the chair ready for them. You can think of this line of code as creating a placeholder for the joke. It must be blank at the beginning of the program because the joke is only displayed when the button is pressed.

The final line of code simply tells your program to display the GUI when you run it. Copy the lines of code from listing 2.5 into your program.

**Listing 2.5 Add the control button**

```
button = PushButton(app, command=select_joke, text="Tell me a jok
joke_to_display = Text(app, text = "")
```

```
app.display()
```

Well done. You have now completed the Joke Machine. Press **F5** to save and run your program and have a giggle! If the program doesn't run, check out the troubleshooting section.

**Listing 2.6 Final Listing**

```
# Imports

from guizero import App, Text, PushButton
import random

# Variables

list_of_jokes = ["What do you call a man with no shins? Tony (Toe
                 "Nothing begins with N and ends with G",
                 "What kind of tree can fit in one hand? Palm tre
                 "Why do owls not go dating in the rain? Because

print (list_of_jokes)

# Functions

def select_joke():
    random_joke = random.randrange(0,len(list_of_jokes)) # select
    joke_message = list_of_jokes[random_joke] # selects the text
    joke_to_display.value = joke_message


# App

app = App("Joke Machine", width =700, height=300)
app.bg = "White"
title_text = Text(app, "The Joke Machine")
title_text.text_size = 28
title_text.font = "Impact" # can be changed
button = PushButton(app, command=select_joke, text="Tell me a jok
joke_to_display = Text(app, text = "")

app.display()
```

# Automating the Joke Machine

In this section, you will adapt your original program so that it automatically displays a joke after a set time. This process of making a program automatic is called automation. Automation is an essential part of computer programs, as much of today's software and hardware is required to run without human intervention. Automation the use of technology to achieve outcomes with minimal human input.

Imagine if you were listening to music and every time you wanted to change to a different song it required you to firstly stop the song you're listening to, locate the new song you want to listen to, load that song, and press play. (This is how listening vinyl records works.) It would be frustrating and time consuming.

Instead what happens is the software is automated. You simply click the songs or playlist you want to listen to, and the program automatically plays through the songs.

An automated Joke Machine is a great motivational tool. If you are working or studying (sometimes called revising in other countries) and need to take a break, you can set the program to display a joke every 15 minutes; then you know you have completed 15 minutes of work and you get to laugh! For more customizable timers, check out chapter 10.

To automate your Joke Machine, you are going to add a line of code that makes the program repeat itself. A program or a section of a code that repeats is called iteration. An iteration is used to run a part of the program repeatedly as long as a given condition is met. In the Joke Machine, the condition is that the set time between jokes has passed.

#### CONFUSING TERMS

Often the terms loop or looping and repeat are used to denote the use of iteration.

Open your original Joke Machine program (**Joke_Machine.py**). First you need to save the program with a new name. This is so that you don’t overwrite your first program.

1. From the Python editor menu, choose **File** > **Save As**.
2. Save your Python file as **Joke_Machine_Auto.py**. This will ensure that your original program is not overwritten or deleted.
3. Next, in the App section of the program, find the line of code,

```
joke_to_display = Text(app, text = "")
```

Underneath this line, enter the code in listing 2.7:

**Listing 2.7 Automating the Joke Machine**

```
joke_to_display = Text(app, text = "")
joke_to_display.repeat(1000, select_joke)

app.display()
```

This line of code is very simple and consists of a repeat function to instruct the program to repeat the function called `select_joke`. Then you tell the program how often you want it to repeat the function. In listing 2.7, the repeat time is set to 1000. This means that the repeat time is set to 1000 milliseconds, so the function runs every second. The Joke Machine automatically will display a new joke every second, this is probably too fast as it will not give you time to read the joke. The joke may also be very funny, and it makes you laugh a lot. By the time you have finished your laughing fit you will have missed maybe 20 other jokes. However, one second is a suitable time period for testing to observe that the program is working correctly.

The program uses milliseconds to store the time so you will need to be able to convert seconds into milliseconds. You need to calculate the milliseconds before you add them to your program code. Use the simple formula shown below,

Milliseconds = seconds * 1000

However, you will also have to covert minutes into seconds which uses the following formula,

Seconds = minutes * 60

Let’s look at an example, if you want your Joke Machine to display a new random joke every 5 minutes, then you would calculate the required number of milliseconds as follows,

5 minutes is 5 * 60 seconds,
this is a total of 300 seconds
300 * 1000 is 300,000

So, 5 minutes is 300,000 milliseconds.

The line the line of code in your program would therefore be

```
joke_to_display.repeat(300000, select_joke)
```

For your convenience, table 2.2 provides the time in minutes, seconds, and milliseconds of common time intervals.

**Table 2.2 Times in second and milliseconds of common time intervals.**

| Minutes | Seconds | Milliseconds |
|---|---|---|
| 1 | 60 | 60000 |
| 2 | 120 | 120000 |
| 5 | 300 | 300000 |
| 10 | 600 | 600000 |
| 20 | 120 | 120000 |
| 30 | 1800 | 1800000 |
| 45 | 2700 | 2700000 |
| 60 | 3600 | 3600000 |

Once you have added the extra line of code and decided on your time interval, press **F5** on the keyboard to save and run your program. (It’s always useful to set the time interval to 1000 at first, so that you can test the code is working correctly). Congratulations, you now have an automated Joke Telling Machine but, the real question is are your jokes funnier than a Hyena?

The jokes will be displayed automatically without you having to press the button. However, the button is still there and if you want a new joke, you can

still press the button to display different joke. The timer is independent of the button, so if you press the button, it may be that the joke changes straight away as the timer has expired.

# Three other things to try

In this chapter you created a funny Joke Machine that either displays jokes when you click a button or automatically displays a joke every X number of minutes. Remember that you can add your own jokes in order to personalize the Joke Machine, but remember that you may need to readjust the height and width so the jokes are displayed correctly. Here are some other features that you might want to add.

## Changing the font color and type

Firstly, you can change the colour of the font and the font type. This edit can be made to any of the text but would work better if you change the joke's font color. The color will grab the attention of your audience. On the line of the program, **before** the `app.display()`, add the following line of code:

```
joke_to_display = Text(app, text = "")
joke_to_display.text_color = "Green"
```

You can also change the style of font by adding this line of code after the line of code that changes the color:

```
joke_to_display.font = "fontname"
```

Replacing the name of font with the name of the font that you want to use. The fonts that are available depends on which operating system you are using. Why not try one of the following, "Arial", "Impact" or "Verdana".

## Changing the GUI's background color

Changing the background color of the GUI window will make your Joke Machine more unique and appealing. Simply change the color value that is in the code, for example,

```
app.bg = "Green".
```

## Displaying a funny picture

There are so many funny pictures that it seems a shame not to display one in your GUI. You can choose a meme if you like.

Start by saving the image file that you want to display, into the same folder where the GUI code is stored. Remember that the file needs to be either a PNG or a GIF. Also select a small picture so that it fits into the GUI window.

Next, in the `# Import` section of the code add the `Picture` import to the end of the list. This imports the code that enables the program to display images.

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

Then in the `#App` section of the program, underneath the line of code, `joke_to_display = Text(app, text = "")` and before `app.display()`insert the following line to display your picture,

```
picture = Picture(app, image="name of file.png")
```

Replace the name of file with the file name of your picture and ensure you have the correct file type, either a .gif or .png. Save your program and run it.

You may notice that despite using a small image it does not fit neatly into the GUI window, figure 2.12. This is easy to resolve.

**Figure 2.12 The image is cut off!**


We simply adjust the width, the height or both values of the GUI window so that the image fits within the display. To do this, find the first line in the `# App` section, `app = App("Joke Machine", width =700, height=300)` and adjust the values as required, for example,

```
app = App("Joke Machine", width =400, height=450)
```

You may find that you need to experiment to find a suitable set of width and height values. Figure 2.13.

**Figure 2.13 The perfect size?**


**Listing 2.8 Final code listing after three things to try**

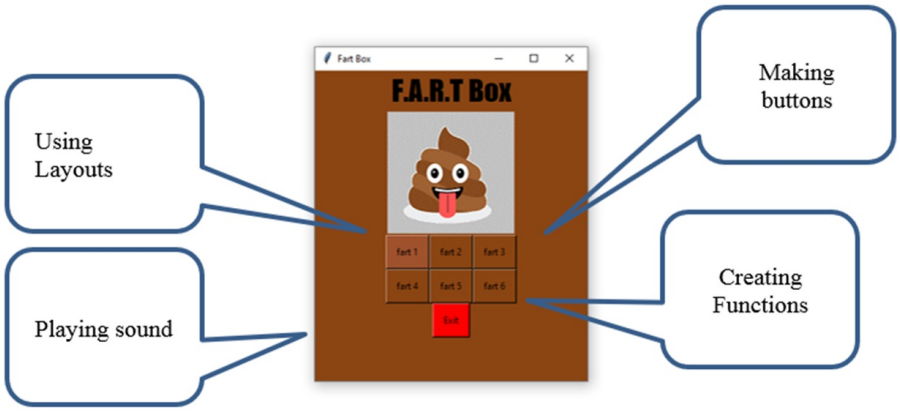
```
# Imports

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

```
# Variables

list_of_jokes = ["J1", "J2", "J3", "J4", "J5"]

# Functions

def select_joke():
    random_joke = random.randrange(0,len(list_of_jokes)) # select
    joke_message = list_of_jokes[random_joke] # selects the text
    joke_to_display.value = joke_message

# App

app = App("Joke Machine", width =400, height=450)
app.bg = "White"
title_text = Text(app, "The Joke Machine")
title_text.text_size = 28
title_text.font = "Impact"

joke_to_display = Text(app, text = "")
joke_to_display.repeat(2000, select_joke)
button = PushButton(app, command=select_joke, text="Tell me a jok
joke_to_display = Text(app, text = "")

picture = Picture(app, image="meme.png")

app.display()
```

# Statements of fact

- A variable is a location in the computer's memory which stores information and data. You create variables and use them to store data and strings.
- A list is a type of variable that can store more than one item. You create lists when you want to store more than one item in a location.
- Iteration is repeating or looping parts of the program. You use iteration to tell the program to repeat part of, or to automate parts of, the code. Iteration reduces the amount of human input required.
- When testing programs that use time, pauses and delays, it is always useful to initially set the time interval to 1000 milliseconds, so that you can test the code is working correctly.

# 3 Making a F.A.R.T. box, a disgusting soundboard




## This chapter covers

- Using layouts
- Making pushbuttons
- Assign sounds to Pushbuttons and playing audio
- Creating functions

Have you heard the term “soundboard”? A soundboard usually consists of a computer program, a physical device, or a website that has a selection of buttons. When you press a button, it plays an audio clip. The sound clip is commonly a quote from a film or a lyric from a song, and has even been sound effects. Soundboards are self-contained, requiring no outside media player; you simply press a button and sound is played.

Trumps, bottom burps, gas, pump noises are funny and make most people laugh. Farts have an amazing habit of being able to distract anyone hearing one, especially if they are playing an online game. So, in this project we will build a fun GUI that combines a selection of fart noises to create a Fluff

Audio Resonance Transmission (F.A.R.T.) sound board.

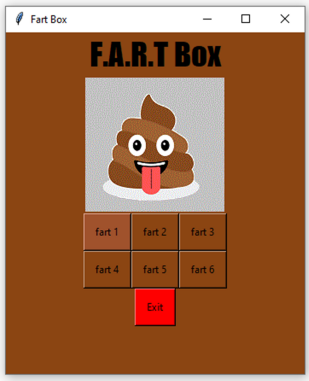
## WHAT YOU WILL BUILD

In this chapter you will build a F.A.R.T. Box soundboard (figure 3.1). This soundboard is a GUI with seven buttons. When the user presses a button, the app plays a fart sound. Press a different button and it plays a different fart sound. The last button closes the soundboard.

### Make your own

However, if you prefer, you can collect your own sound clips and make your own soundboard; for example, cat meows, famous quotes from a TV show, and so on. The program code and the process are still the same; you just replace the sound clips with your own audio.

**Figure 3.1 The Fluff Audio Resonance Transmission (F.A.R.T.) soundboard**



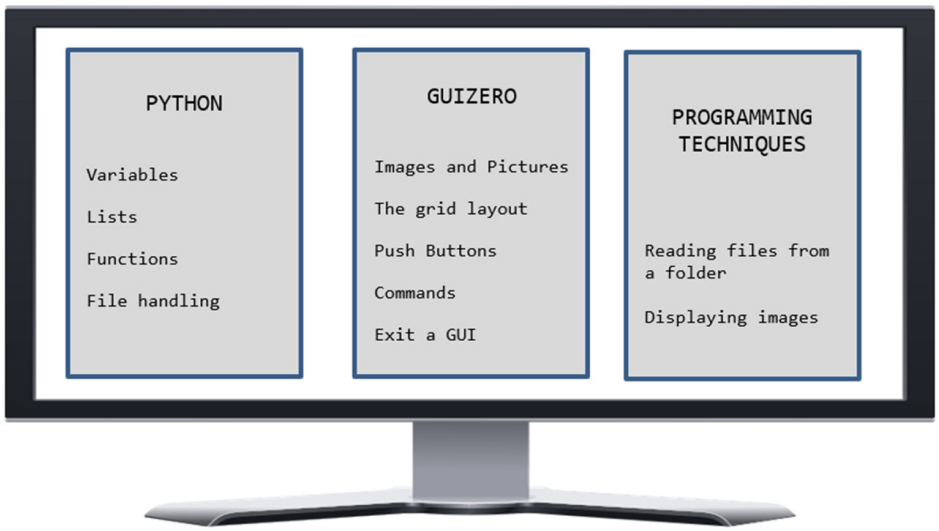



**Table 3.1 Skills that you will learn**

| Python | guizero | Programming techniques |
|---|---|---|
| Variables<br>Lists<br>Functions<br>File handling | Images and Pictures<br>The grid layout<br>Push Buttons<br>Commands<br>Exit a GUI | Reading files from a folder<br>Displaying images |

# Creating the project

Before you start the project, you will need to download the six audio clips from here:

https://github.com/TeCoEd/Twisted_Python_Projects/tree/main/Project_Code

Or you can collect your own audio files. These sound files need to be .wav (waveform) audio file format, as this file format is compatible with the Python module that we are using to play sound. More detail on this later, in the paragraph after we import the modules in the listing 3.1.

The project has six buttons. Therefore, the project requires six fart sounds.

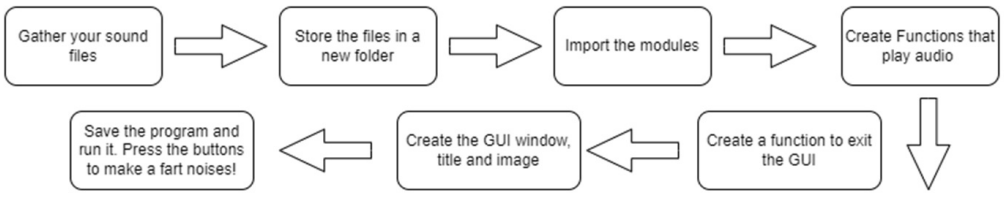
The soundboard also has an exit button which, when pressed, closes the GUI window, but not before leaving you with the sound of a loud burp! The GUI also displays a suitably related image file, of a poo, that is 160 pixels wide and 180 pixels high. The image will be displayed in the GUI window.

Guizero uses .png and .gif file formats, so make sure that your file has the correct format. If working on an Apple computer, macOS only supports the GIF file format.

Let's begin to build the F.A.R.T sound board. The project consists of several steps which are shown in figure 3.2. The main steps of the project are to

1. Download the collection of audio files of the fart noises and the image .
2. Create a new folder, then copy and store the audio files in this folder.
3. Start a new Python file and import the required modules.
4. Code six versions of a function that will each play one of the audio files when a corresponding button is pressed.
5. Create a function to exit the soundboard once you have had enough of all the fart noises.
6. Create the main GUI window to hold the title, image, and buttons.
7. Create the required buttons.
8. Save and test the soundboard
9. Turn up the volume, press a button and make a fart sound!

**Figure 3.2 Flow diagram of the building the Fluff Audio Resonance Transmission (F.A.R.T.) soundboard**



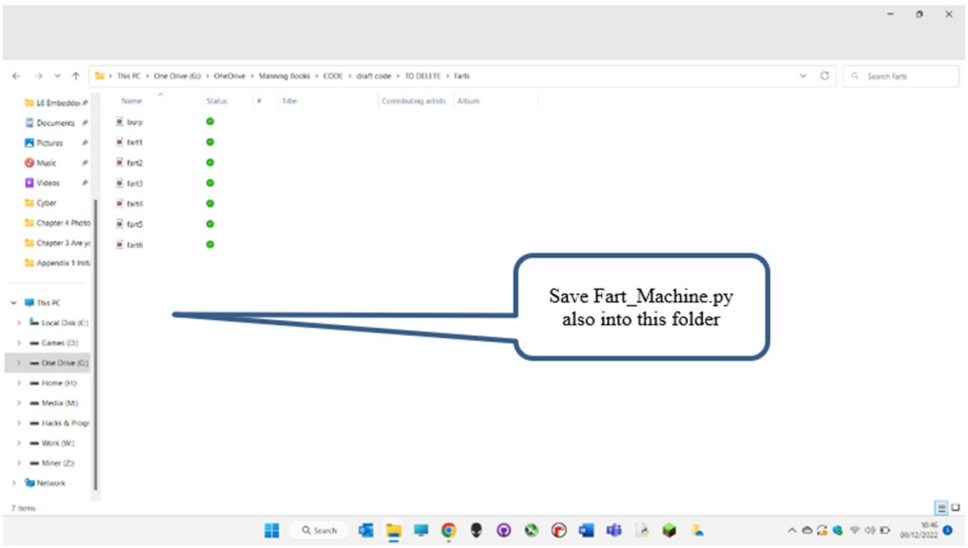

## Downloading and storing the audio files

There are several options that you can choose from to collect the audio clips for your soundboard. The simplest is to head over to the projects page and download the audio files. I recommend building this project with the supplied

audio files to ensure that the program works correctly. Once working, you can then source your own sounds and customize your soundboard. The audio sound files need to be stored in a folder that both Python and guizero can access. Begin by opening the folder where you are saving your projects:

1. Create a new folder and name it, Farts.
2. Open your web browser and enter the address, https://github.com/TeCoEd/Twisted_Python_Projects/tree/main/Project_
3. Click on the folder called **Farts** and download the folder, it contains the audio clips and image.
4. Open the folder and extract the audio files.
5. Copy and paste, drag, or save all the audio files into the Farts folder, figure 3.3.

**Figure 3.3 Fart files in the folder.**




You are now ready to write the program code and create your disgusting or not so disgusting soundboard.

1. Load your Python editor.
2. Open a new Python program file, from the **File** menu, choose **File** > **New File**.
3. A new window opens. Choose **File** > **Save**. The Python editor prompts you to save the file.

4. You must save the Python file in the same folder that also holds the audio clips— Farts folder. If you do not do this, the GUI will not be able to find the sound files. Name your file **Fart_Box.py** and choose **Save**. (Remember, as with the previous projects, if you used the guizero quick install method from chapter 1, then you must save the file into the same folder as the guizero.)

As with the Hello GUI project in chapter 1, we begin the program by importing the required modules (listing 3.1) from the guizero library. Modules contain prewritten functions that perform different tasks, such as displaying text, displaying images, and much more. In this context, the modules provide all the code needed to create the various elements and widgets of your GUI.

**Listing 3.1 Importing the modules**

```
# Imports

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

You may recognize the `PushButton` and `Text` modules from chapter 1. These modules import the prewritten code to display text and create a button that you can push. In this GUI, we also use the `Picture` module to add a picture to the GUI. The `Box` object operates as an invisible container that holds the buttons and creates an ordered layout.

Next, we import the winsound module. This module is included in the Python installation files and provides basic access to the software in Windows OS that plays sound. You can read more about the features of winsound here, https://docs.python.org/3/library/winsound.html. (If you are using a macOS or Linux, then refer to the section on the last few pages of this chapter for information about a compatible audio module.) Save your program before you move on.

This program has no discrete variables, however, there are variables that create objects that hold the related code for the title, the image and the pushbuttons, so we will write them in the `#App` section of the program. Therefore, the variable section in this project is left blank. You may recall

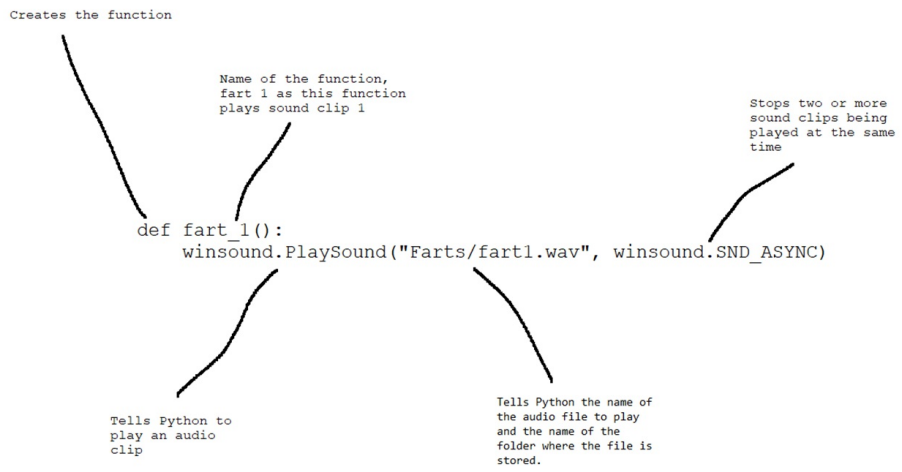
from chapter 1 that a variable is a location in the computer's memory that stores data or information.

## Creating the Farting Functions

Now let's create the farts, I mean, the functions that will play the sound files that you collected earlier and stored in the **Farts** folder. If you completed the starter GUI in chapter 1, you may recall that a function is a reusable block of code that can be called and used anywhere within a program. Functions save time and make the program more efficient.

In this program, the function is assigned to a button which, when pressed, will call the function. On the button press, a Function runs, playing the audio clip (figure 3.4).

**Figure 3.4 How the function works**




Add the functions from listing 3.2 to your program code

**Listing 3.2 Function code**

```
# Functions

def fart_1():
```

```
    winsound.PlaySound("Farts/fart1.wav", winsound.SND_ASYNC)

def fart_2():
    winsound.PlaySound("Farts/fart2.wav", winsound.SND_ASYNC)

def fart_3():
    winsound.PlaySound("Farts/fart3.wav", winsound.SND_ASYNC)

def fart_4():
    winsound.PlaySound("Farts/fart4.wav", winsound.SND_ASYNC)

def fart_5():
    winsound.PlaySound("Farts/fart5.wav", winsound.SND_ASYNC)

def fart_6():
    winsound.PlaySound("Farts/fart6.wav", winsound.SND_ASYNC)
```

Each of the six functions use the same code but with a different function name and audio file name. You just need to remember to update the number used in the name of the function and update the name of the sound file.

**REMEMBER**

- If you have a different folder name other than Farts/ to store your sound files, then you must replace the Farts/ folder name with the name of your folder.
- If you used a different file naming structure for your audio clips, then you need to use these names instead, replacing, for example, fart1.wav with your file's name.
- If you are using Linux or a MacOS look at the section on the last few pages of this chapter which covers how to code the sound.

## Closing the soundboard

Once you have added the code for the six functions that play the audio clips, you need to create a final function that will close the GUI when the Exit button is pressed. The close function uses the code `app.destroy()`, which sounds awful and destructive, but is simply a function that closes the F.A.R.T. app by closing the GUI window.

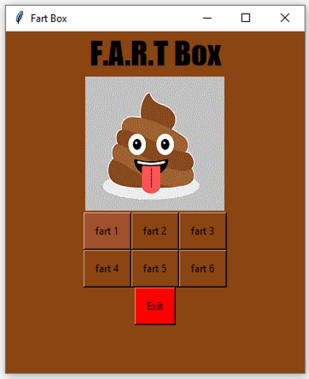
The `exit` function uses the same line of code as used in listing 3.1, except we are changing the sound to a Barney burp, and then adding the code to trigger the GUI to close. Insert the code in listing 3.3 and don't forget to save your program.

**Listing 3.3 Function code to exit the GUI**

```
def exit():
    winsound.PlaySound("Farts/burp.wav", winsound.SND_ASYNC)
    app.destroy()
```

# Creating the GUI for the soundboard

Now that the functions are ready and added to the program, the next step is to create the F.A.R.T GUI soundboard.




The GUI interface allows the user to interact with the soundboard. It consists of

- Setting the size of the GUI window
- Changing the background color of the GUI
- Adding the title: F.A.R.T. box
- Display a suitable image
- Six buttons that each individually trigger a different audio clip
- A button to close the soundboard

## Adjusting the size and color of the soundboard

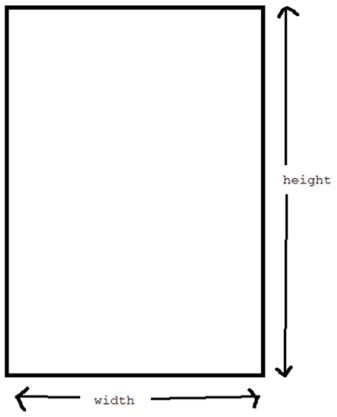
Each time we create a GUI, we use the App object. The App object is the main window of your GUI; in this program, the main App window holds all the features of our soundboard such as the text, the image, and the buttons. The App is named using the code,

```
app = App("Hello World",)
```

This code adds the text to the top left corner of the GUI window.

In this section of the program, we are going to display the name of the GUI which is ‘F.A.R.T Box’ at the top of the window. This title will grab the user’s attention and tempt them to press one of the buttons. We also need to set the size of the app window. The reason for setting the size of the app window is because the soundboard only has seven small buttons and a small image. Therefore, it does not require a large window, which would make it look bad and be a waste of screen space. Imagine drawing a small two centimetre by two centimetre picture on

**Figure 3.5 Setting the sizes of the GUI window.**




a large piece of paper, it would be a waste of the additional space on the paper! To set the size of the App window, the GUI, we provide values for the width and height (figure 3.5).

These values are measured in pixels. We are going to set the size of the soundboard GUI to 350 x 400 pixels, that is a width of 350 pixels and a height of 400 pixels. The code that sets the size is included after the App

window's title. It does not matter whether you put the width or the height value first, line 1 in listing 3.4. Add the code from listing 3.4.

#### What is a pixel?

A pixel (picture element) is a single dot on a display or screen. Each pixel consists of three colors: red, green, and blue (RGB). Displays have different numbers of pixels, and this affects the quality or sharpness of the display. A standard high-definition (HD) monitor is 1920 x 1080 pixels, which means the screen contains 2,073,600 pixels; whereas a 4K display is 2160 x 3840, which is 8,294,400 pixels.

#### Listing 3.4 Setting up the GUI

```
# App

app = App("Fart Box", width = 350, height = 400) #A
app.bg = "saddle brown"
```

On the second line of code you will notice the connection between the color of poo and the background of the F.A.R.T. soundboard. Yep it's the background color of the GUI. The last line of code in listing 3.4 uses the code `app.bg = "saddle brown"`, like in chapter 1, to change the color to saddle brown.

You have many color choices to use. The names and shades are all available on this website: https://wiki.tcl-lang.org/page/Color+Names%2C+running%2C+all+screens. Why not try out some other colors? But remember to keep them a shade of brown so that the color is related to the theme of the project.

## Adding text and an image to the soundboard

Now that you have created the main window of the GUI app, the next step is to add to the window the title of the GUI and the related image. To do this we will create two variables, one to hold the title code and the second to hold the code for the image.

You may recall that a variable is a location in the computer's RAM that stores data or information. Programs need to be able to find data quickly, so each variable you create must have a unique label. It is always a good practice to label the variable with a name that identifies the data or code being stored in the variable. For example, in this project the variable `poo_picture` stores the location of the poo image file.

When creating the label for a variable, you can use more than one word, but if you do so, ensure that it has no spaces between the words. If you add spaces, then the variable will not work, and Python will present you with an error message when the program runs. To add the title and the image, type the code from listing 3.5 into your program.

**Listing 3.5 Code for the title and image**

```
title_text = Text(app, "F.A.R.T. Box")   #A
title_text.text_size = 28   #B
title_text.font = "Impact"   #C
poo_picture = Picture(app, image="poo.gif")   #D
```

The first line of code in listing 3.5 declares the variable. Declaring is the proper name for creating and labeling a variable in a program. This variable has the label, `title_text`, which is assigned the Text widget. With this widget, we can add text to the GUI window. After the variable is created, the contents of the variable tell Python where to display the text. The word 'app' references the app window that you made in listing 3.4. So, the code is saying "display the title in the main GUI window; that is, the app window."

On the second line, the code uses `.text_size` to set the size of the text. In this program, we set the size to 28, which makes it stand out and grab the viewers' attention. If you are creating your own soundboard, you can alter the text size as required.

Next, we set the font style of the text, using the font function `title_text.font = "Impact" where` the font style is **Impact**. The font styles that you can choose depends on which operating system you are using and which fonts you like using.

The first line of code in listing 3.5 tells Python where to display the text, and

in a similar way, the last line of code tells Python where to display the picture assigning the image to the app window. We declare a variable labelled `poo_picture = Picture`, which assigns the `Picture` widget to the variable. The final part of this line of code (`image = "poo.gif"`) is the name of the image file.

The image dimensions are 160 pixels wide and 180 pixels high. If you decide to use your own image and replace the project image then ensure that it is a similar size to ensure that the picture fits within he app window. (You can alter the size of the GUI window to hold a larger image).

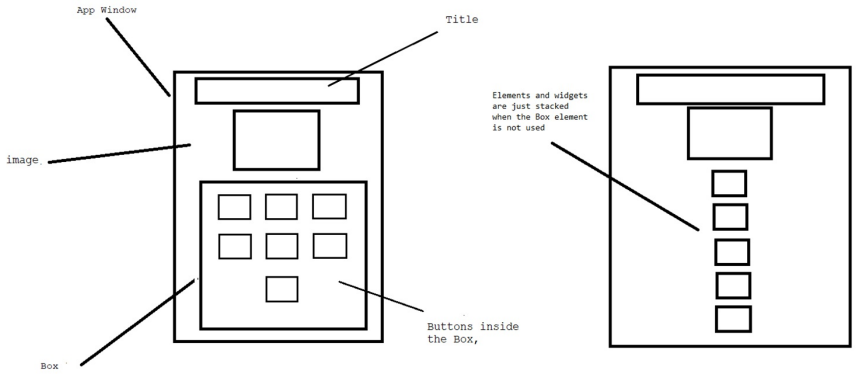
**REMEMBER**

- The image files must be in .GIF or .PNG format.
- macOS only supports .GIF image files.
- If you use your own image, you need to replace the filename after `image =` with your image's filename.
- You must save the image file into the same folder as the **Fart_Box.py** program.

## Using the Box object to lay out the buttons

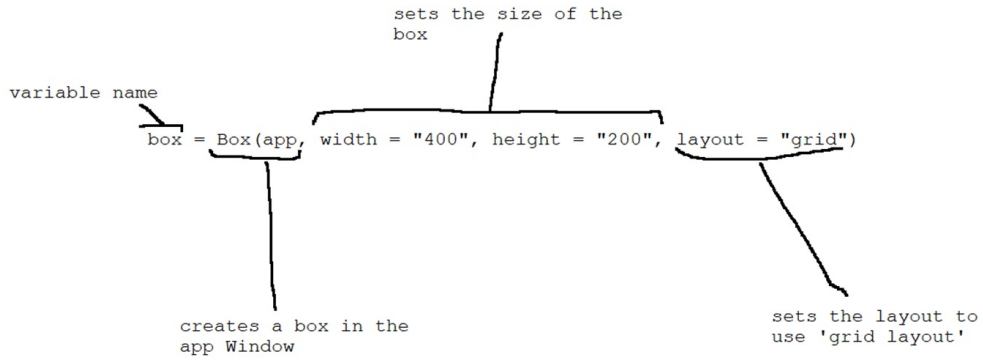
In guizero, the `Box` object is a container which can contain other widgets (figure 3.6). Unlike the App window, a box is invisible. Using the `Box` has the following benefit, it neatly lays out the widgets and elements inside it. In this project it neatly lays out the buttons, otherwise the App window just stacks the buttons vertically on top of each other.

**Figure 3.6 Layouts when using and not using the Box object.**




In this project we will create and use a `Box` to hold the six buttons that when pressed, will each trigger their corresponding functions and play a fart audio clip, figure 3.7. This means that the buttons are grouped together, and they are easy to arrange into a neat and tidy order. Add the code in listing 3.7; it starts with a variable labelled `box`. By now you will be noticing that variables are a staple part of programming.

**Figure 3.7 Code to create and configure the Box within the app window**




**Listing 3.6 Code for the title and image**

```
box = Box(app, width = "400", height = "200", layout = "grid"
```

Then we add the code to configure the box. The first word, `app`, shows that the box is part of the app window. Then we set the size of the `Box` like you previously did in listing 3.4, where we set the width and height of the GUI window. Here we are setting the width and height of the box, which you will

notice is smaller than the app window. This is so the box fits neatly inside the App window.

In chapter 1, when we added the widgets, we simply added them to the GUI and the program automatically took care of laying them out. In this project, we want to control the layout, so we use a feature called grid layout and add it to the `box` variable using the code

```
layout = "grid"
```

This layout offers more control over where the buttons are displayed. With the grid layout, you can position widgets into a virtual grid inside the box.

Imagine dividing the box into equally sized blocks, and then using an x and y value to locate one of the grids, in a similar way that you do when you play battleships or if you have ever used a cell reference in a spreadsheet to locate a specific cell. At school, math teachers used to have a phrase to remember the order of the x and y coordinates as “along the corridor and then up the stairs,” where the x value is first and horizontal (along the corridor), and the y value is second and represents to vertical values (up the stairs).

The grid layout uses an x and y value (the x and y are called a coordinate) to place the button in a specific location in the box. For example, button 1 has the following coordinate, [0, 0], where the first 0 is the x value and the second 0 is the y value. This coordinate places the first button in the first line in the box and on the left. Figure 3.8. Where it can be confusing is that [0,0] is located at the top left-hand side of the grid and not the bottom left-hand side as with graphs.

**Figure 3.8 Coordinates when using grid layout.**


| Y/X | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0,0 | 1,0 | 2,0 |
| 1 | 0,1 | 1,1 | 2,1 |
| 2 | 0,2 | 1,2 | 2,2 |

The second button uses the coordinates [1,0], which means the x value is 1 and the y value is 0. The x indicates the second block and the y indicates that you're still on the first line. Let's set up the first button.

## Coding the buttons that trigger the fart sounds

Now to add to the program the code to create the buttons that when clicked, play the fart sound (or another sound if you are building your own soundboard). The line of code in listing 3.7 assigns the Push Button widget to the GUI and identifies where to display the button in the box.

The essential feature of the code is the *command*, which is passed to the `PushButton` widget and is used to call one of the functions from listing 3.2. The called function then plays the fart audio clip. Commands make it possible for the user to interact with the GUI; for example, when you press the button, the command calls the function which makes the GUI respond creating interaction. In the chapter 1 GUI, the command made the GUI say Hello to you. In other chapters, a command will make it possible for you to draw. In this chapter, the commands are linked to

- the functions that play the sound clips and
- the function that closes the GUI.

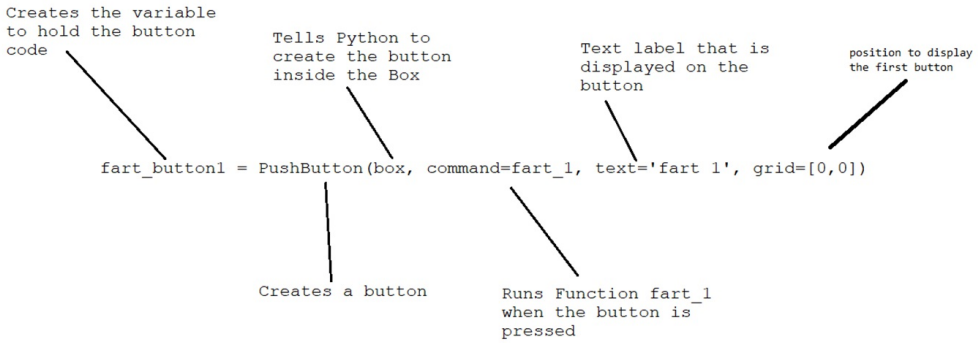
Add the code in listing 3.7 to your program.

**Listing 3.7 Code to add the first button**

```
fart_button1 = PushButton(box, command=fart_1, text='fart 1', gri
app.display()
```

In chapter 1 we gave the button a label so that the user knows what it does. Do the same thing here and label the button with fart 1. You can label the button with any text, so you could use the name of the fart sound instead? For example, you could label one of the buttons, 'squeaker' or 'loud and proud', 'silent and deadly' or how about, 'Trump'. Then use the grid code to align the button's layout (figure 3.9).

**Figure 3.9 How to create a fart button**




At this point you can test your GUI by including the line `app.display()`. This means that you can see if the program is working correctly before you add the other buttons. It will be easier and quicker to find and correct errors in a program with fewer lines of code to check.

Do not worry if the program fails the first time. With a longer program, it is likely that you will have some errors. Commonly, errors are syntax errors. A syntax error is like a grammar error, only more serious. With human language, people can generally understand you even if you made grammatical mistakes. Programming languages, such as Python, expect you to use proper syntax for the program to work correctly.

Press **F5** on the keyboard. Python prompts you to save the program file. Choose **OK**. The program will save, and then the program will run, and the F.A.R.T. GUI soundboard will be ready. The GUI should display the title, the poo related image, and only one button, which when pressed plays fart sound clip 1.

If your program fails or does not run as expected, check for these common errors:

1. Check that your computer audio is not on mute.
2. Is the volume turned up?
3. Are the audio clips that you are using .wav files?
4. Did you create a folder called **Farts**, and are all the audio files placed inside this folder?
5. Is your **Fart_Box.py** program saved in the same folder that holds the

**Farts** folder? (Not saved inside the Farts folder)

6. If you named the folder and audio clips differently, then have you used the correct names in your program code? For example, if your folder is named Burps and you first audio file is called burping1.wav, then the code in the functions should be `winsound.PlaySound("Burps/burping1.wav", winsound.SND_ASYNC)`
7. When creating the buttons, have you used the same function name in the command?
8. Have you used the correct case? Some syntax has capital letters in the middle of a word. For example, TextBox, PushButton.
9. Have you used straight quotation marks ("") for each string?
10. Did you spell all the words correctly?

Once the program is working, you can then delete the `app.display()` from the last line and then add the additional buttons from listing 3.8.

**Listing 3.8 Code for the additional buttons**

```
fart_button2 = PushButton(box, command=fart_2, text="fart 2", gri
fart_button3 = PushButton(box, command=fart_3, text='fart 3', gri
fart_button4 = PushButton(box, command=fart_4, text='fart 4', gri
fart_button5 = PushButton(box, command=fart_5, text='fart 5', gri
fart_button6 = PushButton(box, command=fart_6, text='fart 6', gri
```

## Creating a button to close the F.A.R.T. soundboard

The last stage of the program is to add the code that creates a button that when pressed closes the disgusting soundboard. This is important, as the user may have had enough of the fart sounds and want to leave! Also, it is good GUI practice to include a method to cleanly close the GUI. In chapter 1, to leave the 'Hello GUI' program, you chose the x in the top right corner of the GUI window. This used the standard close a window feature built into the Windows OS. Creating a dedicated close or exit button means that you can customize it. Add the final code from listing 3.9 to your program code.

**Listing 3.9 Adding a button to exit the GUI**

```
   close_button = PushButton(box, command=exit, text='Exit', grid
close_button.bg = "red"   #B
```

```
app.display()   #C
```

You will notice that the code for the `close_button` is the same code that we used in listing 3.7 where we created the buttons to trigger the audio clips. In listing 3.9, we use the variable `close_button` to store the information related to the button including which function to run when the button is pressed. The button appears in the box, so the element `box`, is included after the first bracket.

Next, assign the exit function that you created to the command using the line `command = exit`. This tells the button to run the function that closes the GUI when the user presses it. Then we name the button. Here we have called it Exit. Lastly, use the grid alignment to display the button in the middle of the very last line, underneath the other buttons.

The next line of code changes the color of the button to red, so that the user can see that it is not an audio clip. Also, the color red acts as a warning that if they press the button something else will happen, the program will stop running and the GUI will close. Finally, on the last line you add the code `app.display()` to run the GUI window.

## Running and testing your GUI program

That is it, you have now built a disgusting FART soundboard, well done. Now to test that it works correctly. Press **F5** on the keyboard. Again, you may be prompted to save the program file. Choose **OK** and the program will save, and then the program will run, and the F.A.R.T. GUI soundboard will be ready. Turn up the volume and press a button. Go on, I dare you. If you encounter any errors this time, then refer to the error questions listed in the paragraph just after code listing 3.7.

**Listing 3.10 Final code**

```
# Imports

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

# Variables
```

```
# Functions

def fart_1():
    winsound.PlaySound("Farts/fart1.wav", winsound.SND_ASYNC)

def fart_2():
    winsound.PlaySound("Farts/fart2.wav", winsound.SND_ASYNC)

def fart_3():
    winsound.PlaySound("Farts/fart3.wav", winsound.SND_ASYNC)

def fart_4():
    winsound.PlaySound("Farts/fart4.wav", winsound.SND_ASYNC)

def fart_5():
    winsound.PlaySound("Farts/fart5.wav", winsound.SND_ASYNC)

def fart_6():
    winsound.PlaySound("Farts/fart6.wav", winsound.SND_ASYNC)

def exit():
    winsound.PlaySound("Farts/burp.wav", winsound.SND_ASYNC)
    app.destroy()

# App

app = App("Fart Box", width = 350, height = 400)
app.bg = "saddle brown"
title_text = Text(app, "F.A.R.T Box")
title_text.text_size = 28
title_text.font = "Impact"
poo_picture = Picture(app, image="poo.gif")

# Fart buttons
box = Box(app, height = "200", width = "350", layout = "grid")

fart_button1 = PushButton(box, command=fart_1, text='fart 1', gri
fart_button2 = PushButton(box, command=fart_2, text="fart 2", gri
fart_button3 = PushButton(box, command=fart_3, text='fart 3', gri
fart_button4 = PushButton(box, command=fart_4, text='fart 4', gri
fart_button5 = PushButton(box, command=fart_5, text='fart 5', gri
fart_button6 = PushButton(box, command=fart_6, text='fart 6', gri

close_button = PushButton(box, command=exit, text='Exit', grid=[1
close_button.bg = "red"
app.display()
```

# Two other things to try

As with chapter 1, this section of the chapter introduces some additional edits for you to add to your program. They are basic; however, they can make the GUI look and feel very different. You will come across theses edits in later chapters and if you wish, you can use the edits in your own future projects.

## Changing the background color of the buttons

One easy edit to make is to change the color of the buttons. For example, you could color code the audio buttons in terms of, the browner the color the more bombastic the fart noise is! This would make it easier for the user to know which buttons would make a louder fart. For example, a lighter shade of brown could be a squeaky fart, but the dark brown is loud. Coloring the buttons is also a useful feature for future projects.

To edit the color, you simple create a new variable in the App section with the same name as the button variable, then use `.bg= color` followed by the color you want to use. For example, the first button is labeled `fart_button1`, so the code to change the color of this button is

```
fart_button1.bg = "sienna"
```

And the code for button 2 is

```
fart_button2.bg = "saddle brown"
```

I recommend that you add the line of code for the color underneath the code for the buttons. This will make it easier to keep track of.

```
fart_button1 = PushButton(box, command=fart_1, text='fart 1', gri
fart_button1.bg = "sienna"

fart_button2 = PushButton(box, command=fart_2, text="fart 2", gri
fart_button2.bg = "saddle brown"
```

## Adding more buttons and sounds

If you can’t get enough of the fart sounds, or your soundboard requires many

more buttons, then you can add more buttons. This involves three stages:

1. Collect or make the additional audio sound files that you want to use in your GUI. Ensure that these files are .wav and save them into the correct folder. (In this project the folder is named Farts.)
2. Create a new function for each of the additional sound files. This uses the same code structure used in listing 3.2. In the `#Function` section of the program, add the new Function, remember to use a different name for the Functions and change the filename of the new sound clip, shown in italics below.

```
def fart_1():

    winsound.PlaySound("Farts/fart1.wav", winsound.SND_ASYNC)
```

3. Add the code to trigger the additional button. In the `#App` section under the last button, which was `fart_button6`, add a new line and type in the code for the new button:

```
fart_button7 = PushButton(box, command=fart_7, text='fart 7', gri
```

Remember to change the name of the button and link the command to the new function. You will also need to update the grid numbers. Otherwise, when the program runs, Python will try to create two buttons in the same place. To make the new buttons fit, you may need to extend the height of the box that holds the button. Do this by adjusting the width value (shown in bold) in this line of code:

```
box = Box(app, width = "400", height = "200", layout = "grid").
```

# Statements of fact

- A package is a collection of different Python modules.
- The `Text()` object displays text that cannot be edited while the GUI is running. It is useful for titles, labels, and instructions.
- Objects are a bundling of variables and functions so they work as a single unit.
- The `Picture()` object displays images in the GUI. The default file

formats are .GIF and .PNG. macOS only supports GIF format.
- Commands add interactivity between the user and the GUI interface.
- A window is the main app window of the GUI. It holds the parts of the GUI, widgets, text, and buttons.
- A `Box` is an invisible container that, like a window, can hold widgets, text, and buttons. Editing a `Box` does not edit the main GUI window.
- A grid layout uses an X and Y coordinate to position a widget in a precise location within the GUI.

# Using Simple Audio

If you are using a Linux or Mac OS operating system, then you will need to use an alternative sound module to winsound. This is because winsound is only packaged and available with Windows OS. We can use simple audio instead.

This module is described as providing cross-platform, dependency-free audio playback capability for Python 3 on MacOS, Windows, and Linux. So, it provides an alternative if you are not using Windows OS. You can read more about simpleaudio here, https://pypi.org/project/simpleaudio/

**INSTALLATION**

Download and install of simpleaudio can be completed in one command from the Terminal window. Open you Terminal windows and enter the code.

```
pip install simpleaudio
```

Wait for the program to download and install, and you are ready to make disgusting FART sounds, well, your GUI is! The program code, as the name suggests, is simple to use, first you import the audio module using the line, `import simpleaudio as sa`. Then you create an instance of the audio file and assign it to a variable.

You can now control when the sound plays using the code, `play_obj = wave_obj.play()`. The program needs to wait for the sound to play through before moving to the next line of code, to do this use the code

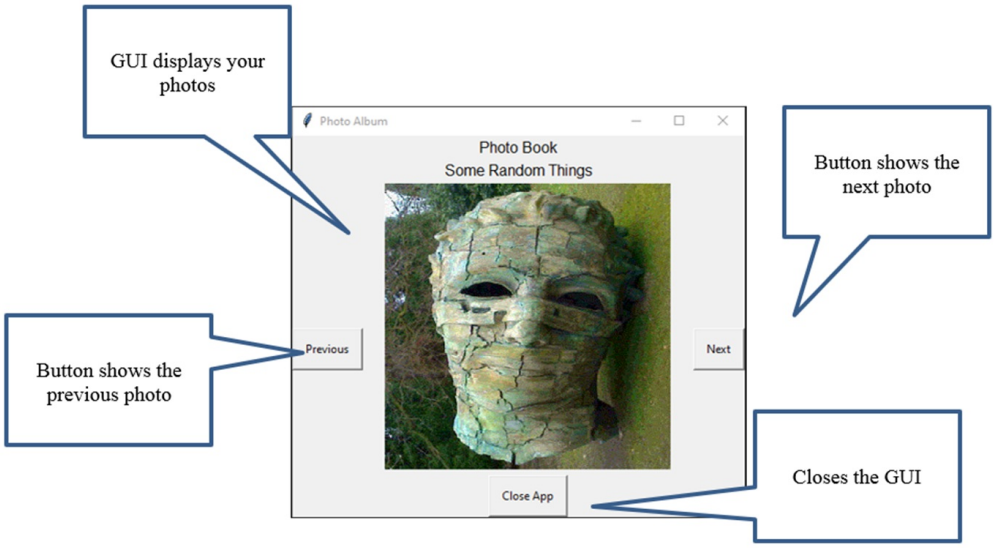
```
play_obj.wait_done()
```

**Listing 3.11 Alternative OS listing, Final code**

```
# Imports

from guizero import App, Box, Picture, PushButton, Text
import simpleaudio as sa

# Variables

# Functions

def fart_1():
    wave_obj = sa.WaveObject.from_wave_file("Farts/fart1.wav")
    play_obj = wave_obj.play()
    play_obj.wait_done()

def fart_2():
    wave_obj = sa.WaveObject.from_wave_file("Farts/fart2.wav")
    play_obj = wave_obj.play()
    play_obj.wait_done()

def fart_3():
    wave_obj = sa.WaveObject.from_wave_file("Farts/fart3.wav")
    play_obj = wave_obj.play()
    play_obj.wait_done()

def fart_4():
    wave_obj = sa.WaveObject.from_wave_file("Farts/fart4.wav")
    play_obj = wave_obj.play()
    play_obj.wait_done()

def fart_5():
    wave_obj = sa.WaveObject.from_wave_file("Farts/fart5.wav")
    play_obj = wave_obj.play()
    play_obj.wait_done()

def fart_6():
    wave_obj = sa.WaveObject.from_wave_file("Farts/fart6.wav")
    play_obj = wave_obj.play()
    play_obj.wait_done()

def exit():
    wave_obj = sa.WaveObject.from_wave_file("Farts/burp.wav")
    play_obj = wave_obj.play()
    play_obj.wait_done()
```

```
    app.destroy()

# App

app = App("Fart Box", width = 350, height = 400)
app.bg = "saddle brown"
title_text = Text(app, "F.A.R.T Box")
title_text.text_size = 28
title_text.font = "Impact" # can be any of the following p 16 SEE
poo_picture = Picture(app, image="poo.gif")

# Fart buttons
box = Box(app, height = "200", width = "350", layout = "grid")

fart_button1 = PushButton(box, command=fart_1, text='fart 1', gri
fart_button2 = PushButton(box, command=fart_2, text="fart 2", gri
fart_button3 = PushButton(box, command=fart_3, text='fart 3', gri

fart_button4 = PushButton(box, command=fart_4, text='fart 4', gri
fart_button5 = PushButton(box, command=fart_5, text='fart 5', gri
fart_button6 = PushButton(box, command=fart_6, text='fart 6', gri

close_button = PushButton(box, command=exit, text='Exit', grid=[1
close_button.bg = "red"

app.display()
```

# 4 Photo book: Show off your pics




## This chapter covers

- Pulling names of files into Python
- Creating a list
- Displaying images in a Photo book GUI
- Using buttons to change the image

The year 2016 was a significant year, do you know why? The number of photos taken in 2016 exceeded the total amount of photos ever taken in the history of taking photos! Just let that sink in for a moment. More photos were taken in 2016 than in the whole time between 1830 (when the first official photograph was taken) and 2015, a period of 185 years! (The Online Photographer)

#### Photos and fake news

Taking photos has become an everyday feature of our lives, even to the point where some people say, “if you didn’t take a picture of it, it never happened!” This has led to a rise in fake news where images are manipulated or used out

of context to make the reader believe that something happened, or never happened, or happened in a different way to how it really happened!

The data around photos is interesting. Here are some facts from Photutorial.com (https://photutorial.com/photos-statistics/)

1. In 2021 1.2 trillion photos were taken worldwide.
2. In 2022 1.72 trillion photos were taken worldwide. The number keeps growing.
3. That roughly equates to 54,541 photos taken every second, or 4.7 billion per day.
4. It is estimated that by 2030, 2.3 trillion photos will be taken every year.

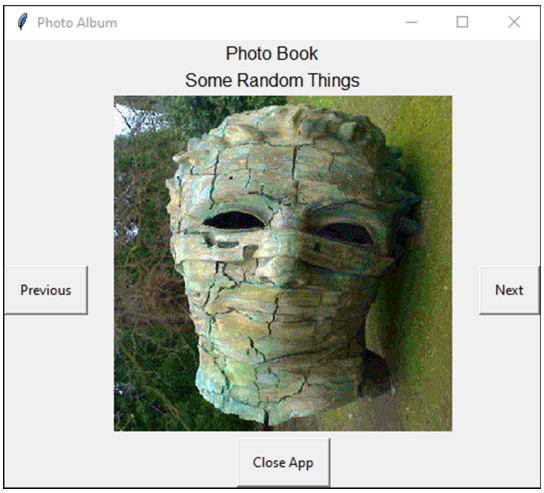
Photos are taken everywhere now, and there is no indication that the number taken is likely to slow down at any point. However, what is odd is that many people do not look at their photos. The image files simply stay housed on their smartphone or transferred to their computer, where they are stored and never looked at.

In the 1990s and even early 2000s, the use of a photo album was quite popular. You would print out your favorite photos, or memories as they were often referred to, and then stick them into a book called a photo album. Then, under each photo you added a suitable caption like “A sunny day at the beach, with lots of ice-cream.” You would flick through the book and share the photos and stories with friends and family members.

## WHAT YOU WILL BUILD

Since the average phone user has around 1,500 or more photos on their phone and never really looks at them, we are going to build our own Instagram style digital photo viewer called Photo Book (figure 4.1).

**Figure 4.1 The Photo Book GUI**




The Photo Book GUI consists of a simple interface that displays your favorite photos and allows you to select a new photo by clicking a button, the **Next** button. You can scroll forward through the photos and then press the **Previous** button to scroll backwards through your photos. You can show your friends and family members and share your memories.

The program that we will code will enable the GUI to automatically pull the photos from a folder on your computer. This means that if you want to change the photos or add more photos, you just move the new photos into the folder, without changing the code. It is that easy. You don't even have to change the file names, which saves time. Want to make a Photo Book for a party? Copy your party photos over to the folder. How about a Photo Book of your pet? All you need to do is copy over your favorite photos of your pet to the folder. Each time you run the GUI the program will automatically pull in your new images and create your new Photo Book.

**Table 4.1 Skills that you will learn**

| Python | GUI ZERO | Programming techniques |
|---|---|---|
| Variables<br>Lists<br>Functions<br>File handling | Images and Pictures<br>The grid layout<br>Push Buttons<br>Exit a GUI | Reading files from a folder<br>Displaying images |

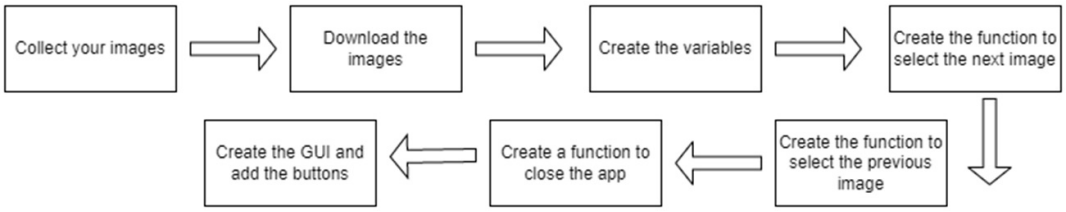
# Creating the project

Before you start the project, you will first need to download the sample photos to build and then display the photos in Photo Book. If you want to use your own images or the images taken with the camera on your smartphone or downloaded from the Internet, the chances are that each image is rather large and high quality. This means that if you were to use the images straight from your smartphone, the GUI window would be massive and unusable, potentially the size of a large sheet of paper. To use your own pictures, you will need to adjust the size of the images. This skill is covered in Appendix E.

Let's begin to build the Photo Book. The project consists of several steps which are shown in figure 4.2. The main steps of the project are to

1. Download the prepared image files.
2. Create a new folder, then copy and store the images in this folder.
3. Start a new Python file and import the required modules.
4. Declare the variables to hold the photo number.
5. Create a function to display the next image.
6. Create a function to display the previous image.
7. Create a function to close the GUI.
8. Create the main GUI window to display the title, photos, and buttons.

**Figure 4.2 The project outline**




## Downloading the images

To download the images, open your web browser and enter the address, https://github.com/TeCoEd/Twisted_Python_Projects/tree/main/Project_Code If you are using your own images then ensure that they have the file extension .gif, else the picture will not be displayed. This is because by

default guizero supports and displays gif image files on all operating systems. Windows and Linux also support .PNG files. To use images saved as JPEG file format we will need to first install some additional software. This is covered later on in the chapter, (in the three other things to try section) once we have built the Photo Book GUI.

## Storing your image files

The Photo Book code needs to be able to access the images and this is done by including the file location in the Python code. To do this we will make a dedicated folder for storing your images. Later on in the chapter we will cover how to access photos that are stored in a different folder location. Follow the steps below:

1. Move to the folder where you are saving your GUI programs.
2. Create a new folder.
3. Name the folder **Photo_Book_Images**, you can use a different name, but you must remember to change the file name in the main program code so that it matches.
4. Drag and drop or copy over your edited images / photos into the **Photo_Book_Images** folder.

## Importing the Modules

As with the previous chapters, we begin the GUI by importing the required modules. You will already be familiar with these modules if you have completed chapters 1, 2 or 3. The module you probably won't recognize is glob! More on this later in the chapter. Open a new Python program file in your editor and from the **File** menu,

1. Choose **File** > **New File**. Your Python editor will prompt you to save the file.
2. Name the file with the name Photo_Book.py and choose **Save**.

Begin the program code by importing the required modules (listing 4.1) from the guizero library.

**Listing 4.1 Importing the modules**

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

So, what is glob? It sounds kind of slimy! Well, you will be pleased to hear or maybe disappointed that glob has nothing to do with slime. The glob module is really useful for finding the photos that we want to display in the Photo Book.

When using pictures in guizero, the image file will have the file extension .gif. The file extension .gif is an example of a file pattern that we can use with glob to find the image files. When glob is looking for files, it only looks for files that end in.gif and we can use this to find and make a list of the names of the files that match the required pattern.

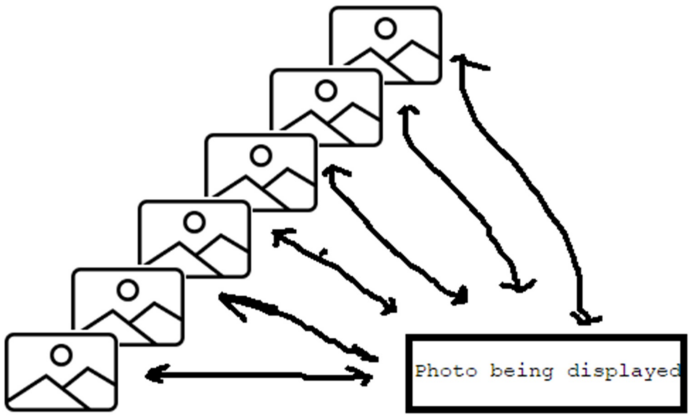
Glob is short for global, and is a module, (you may recall that a module is code or really a function that is prewritten and you can use it straight away) built into Python for finding files that match a particular file extension.

## Creating the variables

Exciting news. This is the first program that uses variables in the variable section! All GUIs use variables; however, we have previously included these in the #App section to support continuity and testing. Now, we actually get to create variables in the #Variable section of the program.

In the Photo Book GUI, we need a variable to keep track of which photo is being displayed, figure 4.3. A variable is a location in memory that holds or stores a piece of data. This data is usually a number or text, and you guessed it, when the program is running the variable can change. Therefore, you need to label the variable with a suitable name so that you, well your program, can easily find it again. This is known in computing as declaring a variable, it is a little like knighting someone. I declare you Sir Photo Number a lot! It helps the coder and also the program code to know where to find and hold data.

**Figure 4.3 The GUI can display any photo, so the variable needs to track which file is currently stored.**




What will be new to you is that there are two types of variables, a local and a global variable. In most homes you will probably find a framed photo that is either hanging on a wall or above the fireplace or placed on a shelf. The photo may even be pinned to the fridge. This photo is often a picture of a pet or embarrassing cringy family photos. This photo can only be seen when you are in the house; you cannot see the photo when you are next door, or in the car, or on holiday. This is like the local variable; it is only accessible within a specific part of a program.

Here is another example. Consider going to the cinema, once you enter the lobby, you can buy a ticket for any film. When you have bought your ticket you go and watch that specific film and only that film. You must be in screen 6 to watch the film, you cannot watch any other films. This is like the local variable; it is only accessible within a specific part of a program.

Alternatively, when you are at the cinema you could buy a weekly pass ticket, this allows you to watch any film that you want at any time, there are no restrictions, all the films are accessible. This is like the global variable which can be accessed by any part of the program.

A global variable (nothing to do with glob which is the module that finds files with certain names) is accessible by any part of the program. This is like the photos that are available on your social media feed; anyone can access them from any location in the world. The same is true of the global variable: any part of the program can access the contents of that variable.

**Listing 4.2 Declaring the variables**

```
# Variables

global photo_number
photo_number = 0
```

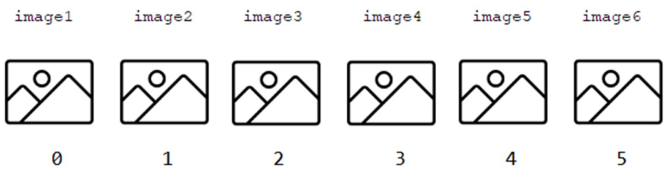
This program only has one main variable, `photo_number` that holds information about which image is currently being displayed in the app window. The variable holds the current image number from a list of numbers representing each image file. It is a similar concept to the list of jokes in Chapter 2 where the program keeps track of the joke by the position that the joke is located in, in the joke list. You will be relieved to hear that in the program we will use the glob module to automatically build the list, rather than you having to create it manually.

The information held in this variable needs to be available to other parts of the program, therefore `photo_number` is created as a global variable using the code, to do this we use the code `global photo_number`. Then, we set the variable `photo_number` to a value of 0, as 0 is the first image file in our list of files that glob will find when it runs in the #App section of the program, figure 4.4.

**Numbering items in a list**

Numbering in Python always starts from zero. The first item in the list is stored in position zero.

**Figure 4.4 The first item in a Python list is always item 0**




# Creating the functions

You may remember from the previous chapters that the next stage in building the GUI is to create the functions. These blocks of code save us having to keep writing out the same lines of code each time we want the program to perform a particular action. For the Photo Book GUI, we will create three functions,

1. The `forward()` function. When you press the button, this function is called and moves the photo onto the next one. Basically, it moves you through the images stored in the folder and displays each photo.
2. The `back()`function, moves you back through the previous photos when you press the back button.
3. The `exit()` function closes the Photo Book GUI app.

Functions 1 and 2 keep track of the current photo being displayed and also keep track of either the next or the previous image to display. Each function also must keep track of the total number of image files there are in the folder. If you have six photos, then the function needs to know when it has reached the end of the list of photos and then say, “hey I need to go back to the beginning of the list and display the first one again”. The function then displays the first photo in the list. Function 2 also needs to do this and check if it has reached the beginning of the list of photos, it tracks which number photo is being displayed and then display the previous photo when the button is pressed.

To make things easier and understandable let’s break down a function and how it works in more detail. The first four lines of the function are simply,

```
def forward():

    global photo_number
    global photo
    photo.destroy()
```

First, we define the function by naming it `forward()`:. That was nice and easy. Next, we need to bring in the variables, first, the number of the photo (`photo_number`) and then next, the variable to hold the information about the photo in the variable aptly named, `photo`. This is a new variable and will hold all the information about the photo. This variable needs to be global so that the program can update the image file number so that the correct photo is

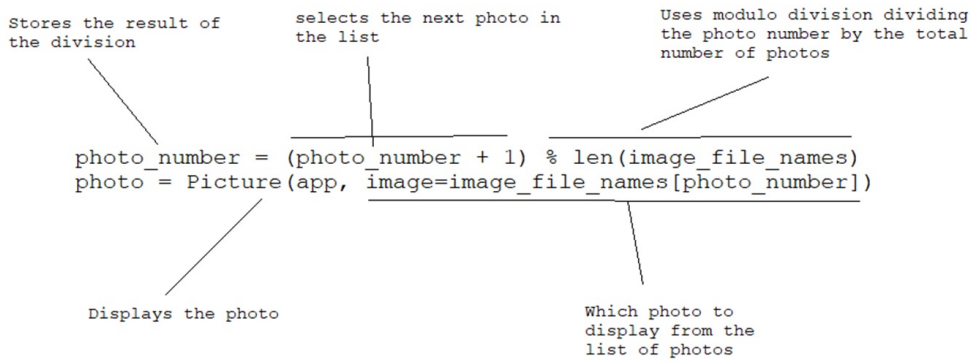
displayed when the user presses either the **Previous** or **Next** button. If the variable was a local variable only available in the `def forward():` function and not a global variable, then the number of the image file currently being displayed would not be updated correctly when the **Previous** button was pressed. This means that the photo order would be incorrect and the user would not experience 'scrolling' through the Photos.

The last line of code calls the `destroy` function to destroy the information about photo. This might seem odd and a little dramatic! Why destroy the photo? Well, the `destroy` function is not really deleting the image file from the folder, it is removing the image from the GUI window. The reason for this is because if you do not remove the current photo from the GUI, it will continue to be displayed even when a new image is loaded. This means we will soon end up with a stack of photos on top of each other!

## Displaying the next image

The remaining section of the function is the part that moves the picture forward, figure 4.5. First, we tell Python which photo we want the GUI to display. This requires us to use the `photo` variable that we just created, to hold this information and combine it with the `Picture()` object to display the image.

**Figure 4.5 How the Forward() function works**




On the first line the function adds 1 to the `photo_number`, this is because the user has pressed the **Next** button and wants to see the next image in the list. This new value is then divided by the total number of images in the list.

(Currently 6 images.) So, the line of code calculates 1 ÷ 6 which equals 0.16 however, you will have noticed that the division symbol used is **%**. In Python this symbol performs a **modulo** division which means that the calculation returns the remainder of the calculation.

For example, 6 % 6 returns 0 as there is no remainder, 6 goes into 6 once, and nothing remains. The division, 6 % 4 returns 2, because 4 goes into 6 once and there is 2 remaining.

#### Modulo division

The modulo operator, % returns the remainder after one number is divided by another. This is known as the modulus of the operation. Take two positive numbers 12 and 3, a modulo 3 (12 % 3 ) returns 0 as 3 goes into 12 exactly 4 times. However, 12 % 7 returns 5 as 7 goes into 12 once and leaves 5 remaining (12 – 7). Often in math the term is abbreviated to mod, for example 12 mod 7.

Where the modulo division appears twisted is if we divide 4 % 6. Normal division would return 0.66 but modulo returns 4. Does that make sense? How many times does 6 go into 4? Zero times, so the remainder is the original number, 4.

In the `forward()` function the modulo calculation is performed using the code, `(photo_number - 1) % len(image_file_names)`which returns a number between 0 and 5, that represents the position of the image file in the list of images and returns that photo. Table 4.2.

**Table 4.2 Modulo division results and related Photo**

| Division | Result | Photo number |
|---|---|---|
| 1%6 | 1 | 2 |
| 2%6 | 2 | 3 |
| 3%6 | 3 | 4 |
| 4%6 | 4 | 5 |
| 5%6 | 5 | 6 |
| | | |

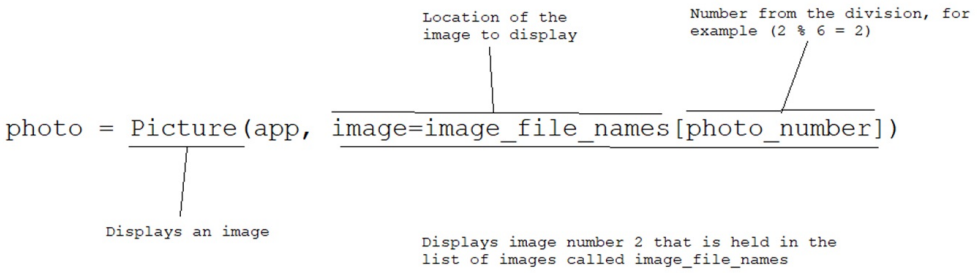
| | | |
|---|---|---|
| 6%6 | 0 | 1 |

The last line of code tells the GUI which image to display. The value stored in the `photo_number` variable that we calculated in previous line with the modulo division, is combined with the `image_file_names` list, which holds the list of images, figure 4.6, which selects a position in the list of image files. (We will create the contents of this list later in chapter, in the `# App` section of the program where we build the GUI.)

For example, `image_file_names[photo_number]`where `photo_number = 1` will display the second image in the list, (remember that lists in Python always start from zero). When `photo_number` is equal to 2, the function will display the third photo in the list.

Then we use this code and combine it with the `Picture()` object, to display the image. We also include `app,` in the `photo` variable to state where the photo will be displayed, (the image is displayed in the app window!).

**Figure 4.6 Code to display the Photo in the GUI.**




So, what happens when the user selects the last photo in the list and the function gets to the end of the list? Well, the Photo Book needs to return to the first item in the list, the image file that is stored in position zero.

To do this the function performs, 6 % 6 which returns 0, when this happens then the `photo_number` is set to 0, which triggers the function to display the first photo in the list. Then the whole process starts again from the beginning. One is added to `photo_number` and then 1 % 6 returns, 1 and image 1 (the second image) in the list is displayed.

Type in the code in listing 4.3

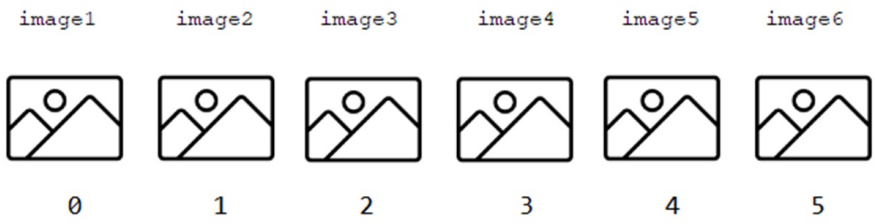
**Listing 4.3 The forward() function**

```
# Functions
def forward():
    global photo_number
    global photo
    photo.destroy() #A
    photo_number = (photo_number + 1) % len(image_file_names) #B
    photo = Picture(app, image=image_file_names[photo_number]) #C
```

Before moving on, let's walk through the stages of the function one more time.

1. The user presses the **Next** button.
2. The function pulls in global `photo_number` and takes current value stored in the variable, for example 3.
3. The global `photo` variable is pulled in.
4. The current photo is destroyed meaning the image currently displayed is removed and the `photo` variable is empty, there is no information stored about a photo.
5. The `photo_number` value is incremented by one, (becomes 4) and then divided by the number of the images in the image list (6) using modulo division, (4 % 6 = 4) which will display image 5.
6. The `image=image_file_names[photo_number]` combines the `photo_number` which locates the fourth item in the list. Remember lists start from 0 so item 4 is image number 5. Figure 4.7.
7. The previous code is combined with `Picture()` and assigned to the variable `photo` which displays the new image.

**Figure 4.7 Image 5 is in the fourth position in the list of images**




## DISPLAYING THE PREVIOUS IMAGE

Now to create the `back()` function, which does the opposite of the `forward()` function. When the **Previous** button is pressed it displays the previous photo in the list. These two functions enable the user to scroll back and forth through the images. Type up listing 4.4.

**Listing 4.4 The back() function**

```
def back():
    global photo_number
    global photo
    photo.destroy()
    photo_number = (photo_number - 1) % len(image_file_names) #A
    photo = Picture(app, image=image_file_names[photo_number]) #B
```

This function begins with the same variables as the `forward()` function. Then it uses modulo division, which hopefully makes more sense now that you understand how the `forward()` works, to select the image number. The code is the same except for this time we are subtracting 1 from the `photo_number`. In the previous function we added one to move forward, this function subtracts one, so the previous image in the list is selected and displayed.

`Let's walk through process of this function.`

1. The user presses the **Previous** button which calls the `back()`function.
2. The function pulls in global `photo_number` and takes current value stored in the variable; for example 3. (Image 4 is currently being displayed)
3. The global `photo` variable is pulled in.

4. The current photo is removed from the GUI, meaning the image on the display is cleared and the `photo` variable is empty. There is no information stored about a photo.
5. The `photo_number` value is **decreased** by one, (becomes 2) and then divided by the number of the images in the image list (6) using modulo division, (2 % 6 = 2) which will select the second item in the list.
6. The `image=image_file_names[photo_number]` combines the `photo_number` which displays the third photo in the list. Remember lists start from 0 so item 2 is image number 3. Figure 4.6.
7. The previous code is combined with `Picture()` and assigned to the variable `photo` which displays the new image.

One of the challenges when writing longer functions is ensuring that the indentation levels are correct and match. Indenting refers to the use of white space before the text, which means the text starts a little distance award from the left hand margin.

**white space**

In coding, white space is the term used to describe the space on the left hand side of your line of code that is not taken up by code or characters. It is useful for organizing your code and telling the program which lines of code belong to various sections.

**Figure 4.8 Using indentation in code**


After you create a function, the lines of code that follow underneath, are indented. This is so Python knows which lines of code are part of the

function. To use the correct amount of indentation you can press the TAB key on your keyboard once (or the spacebar four times) this moves the cursor across to the right about 1 cm, so that when you start typing the text it is indented. You can see the first line of the function ends with a colon :, when you type in a : and press the enter button, usually your Python editor will move to the new line underneath the previous and automatically indent it.

Always ensure that you always use the same amount of indentation throughout your program. Check that your indentation is the correct level before moving on (figure 4.8).

You will be pleased to know that the last function is much simpler than the previous two. It consists of just two lines of code and is assigned to a button, which when pressed closes the Photo Book GUI. Add the code in listing 4.5.

**Listing 4.5 The exit() function**

```
def exit():
    app.destroy()
```

## Building the main app

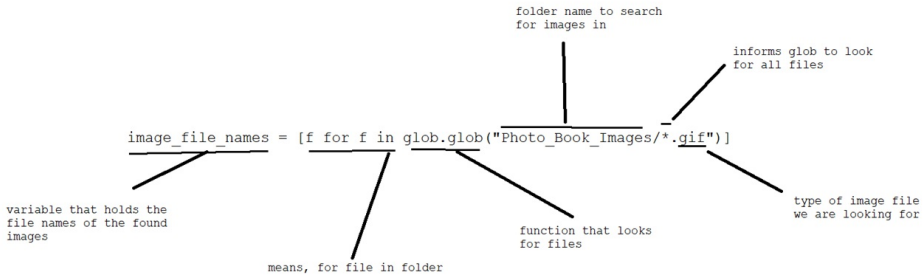
Now that we have the functions ready, we can move onto creating the main window of the GUI, the App part of the GUI. As we did in previous chapters, this is where we bring all the widgets and functions together to create the Photo Book.

This section of the program needs to find your photos and make them available for the Python program to use and display in your GUI. Look at listing 4.6, the first line of code uses the glob module to find the image files. This line consists of creating a variable to store the image files that the glob finds. Then we use the code, `f for f in glob.glob` which basically means *"for the files in the list of files that glob found"*, and then you must tell glob where to look for the files, to return a list of file names and store them in the variable.

Just like looking for a book in a library, you need to know the area to look in and the type of book, the author and usually the title of the book. When using

glob, we need to provide it with two pieces of information. Firstly, we need to tell it which folder to look in (the one we made at the start of the chapter to store the images in, Photo_Book_Images) and secondary, the type of files to look for, `*.gif`. Figure 4.9.

**Figure 4.9 The glob code to find a build a list of image files**




**The * Symbol**

The * symbol tells glob to look for any file that ends with the file extension **.gif**. For example, glob would find cat.gif and mydog.gif but it would not find mydog.jpg because the file type is .jpg, which we have not told glob to search for.

As the program is looking specifically for a file type, it means we don't have to worry about the naming convention used on the files. Glob is always watching and will find the image files. To add the glob code, type the code in listing 4.6.

**Listing 4.6 Finding the image files**

```
# App
image_file_names = [f for f in glob.glob("Photo_Book_Images/*.gif
```

As with the previous GUIs, at this stage we now configure the size of the window and add the name of the Photo Book. This follows the same steps as we have used before in previous chapters. First define the size using the width and height values. Then add the title of the app, this appears in the main GUI window. In this project we also add a catchy subtitle, to interest

the users and tell them the theme of the photos that are displayed. Photos could be of cats, friends, places, trees, food and so on. Add the code from listing 4.7.

**Listing 4.7 Creating the app**

```
app = App("Photo Book", width = 475, height = 400)
text = Text(app, text="Photo Book")
album_name = Text(app, text= "Some Random Things")
```

You may recall from previous chapters that when setting the width and height, the line of code still runs the same if you define the order, height and width. You can use an alternative name rather than Photo Book simply replace the title with your preferred text.

Now to add the images and create the navigation buttons. These buttons enable us to control the Photo Book, moving forward or backwards through the photos and, a button to close the Photo Book. As with all previous buttons, we link one of the functions to the button using the `command =` code, followed by the name of the function that we want to run when the button is clicked. For example, the `back_button` is linked to the `back()` function, which displays the previous photo in the list. Add the code from listing 4.8.

**Listing 4.8 Creating the buttons**

```
photo = Picture(app, image=image_file_names[photo_number])
back_button = PushButton(app, command=back, text="Previous", alig
next_button = PushButton(app, command=forward, text="Next", align
[CA] exit_button = PushButton(app, command=exit, text="Close App"
align="bottom")

app.display()
```

Listing 4.8 begins with the code to display the first photo when the GUI loads. We need to tell the program which photo to display because the image files in your folder will all have different names. This is because the program does not know what each reader's image files are called and therefore when the GUI runs, no image will be displayed until the first button is pressed, so the GUI window will be empty when the program loads. (All the image displaying is handled by the `forward()` and `backward()`functions).

To solve the issue, we use the code `image=image_file_names[photo_number]` where `photo_number` = 0, (from when we previously declared it in the `# Variables` section). When the program runs it pulls out the photo which is stored in position 0 in the `image_file_names[]` list and displays it.

The code for the three buttons uses the standard button code. If you have already completed other GUI projects, then you will be familiar with the code. First create a variable to hold the code for the button. Assign the button to the `app` window and use the `command=` to assign a function to the button. Give the button a label and then use the `align=` code to place the button either on the left, the right or the middle of the main GUI window. Finally add the code to display the app when the program is running. Save your program code.

## Running and testing your GUI program

We have now completed and built a Digital Photo album, the Photo Book GUI. Well done. Now to test that it works correctly. Press **F5** on the keyboard. You may be prompted to save the program file, choose **OK** and the program will save, and then the program will run. If you encounter any errors this time, then refer to the error questions listed below.

#### Button location

When you first run the program, you may wonder why the three buttons appear at the bottom of the GUI. This is due to the method that guizero uses to stack the GUI elements in the order that they appear. Once you click either the Previous or the Next button, the button placement will change to the sides.

1. Check that the filename and file path are both correct, are the image files stored in the correct place, have you used the correct filename for glob to find?
2. Are the image files the correct type? They should be .gif.
3. Indentation is important and can often lead to errors. Are the levels of indentation correct in the program? Either one TAB key or 4 spacebars.

4. Does the indentation remain the same throughout the program code?
5. When creating the buttons, have you used the same function name in the command?
6. Have you used the correct case? Some syntax has capital letters in the middle of a word. For example, App, Picture, PushButton.
7. Did you spell all the words correctly?
8. Check the final code listing below.

**Listing 4.9 Final code listing**

```
# Imports

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

# Variables

global photo_number #photo number
photo_number = 0

# Functions

def forward():
    global photo_number
    global photo
    photo.destroy()
    photo_number = (photo_number + 1) % len(image_file_names)
    photo = Picture(app, image=image_file_names[photo_number])

def back():
    global photo_number
    global photo
    photo.destroy()
    photo_number = (photo_number - 1) % len(image_file_names)
    photo = Picture(app, image=image_file_names[photo_number])


def exit():
    app.destroy()

# App
image_file_names = [f for f in glob.glob("Photo_Book_Images/*.gif

app = App("Photo Album", width=475, height=400)
text = Text(app, text="Photo Book")
```

```
album_name = Text(app, text="Some Random Things")

photo = Picture(app, image=image_file_names[photo_number])
back_button = PushButton(app, command=back, text="Previous", alig
next_button = PushButton(app, command=forward, text="Next", align
[CA] exit_button = PushButton(app, command=exit, text="Close App"

app.display()
```

# Three other things to try

As with previous chapters, this section introduces some additional edits for you to make to your program and also shows you how to print out, onto paper, one of the images. The edits can make the GUI look and feel very different.

## Adding more image file types

The simplest other thing to try is to add more images to the glob folder. The good news about the glob module is that it will find all new image files each time that you run the Photo Book program. If the GUI is already open and running, close it and open it again, hey presto your new images will be loaded. Remember, all Operating Systems support .gif image file format and Linux and Windows also support .png files.

So, you can use .png files in your Photo Book but you will need to ensure that the image file is a .png file format. You also need to change the code so that it searches for .png files, for example change the line of code from

```
image_file_names = [f for f in glob.glob("Photo_Book_Images/*.gif
```

to

```
image_file_names = [f for f in glob.glob("Photo_Book_Images/*.png
```

The Python Imaging Library (PIL) adds additional image processing powers to Python. However, this library is no longer maintained and has been transferred to a new library named Pillow. This means that we can use Pillow to display more image file types in the Photo Book. This is useful as the most

common image file type is JPEG. Most images taken with your smartphone camera are saved as JPG, many images on websites are JPEG too. Pillow supports the following image files: GIF, Animated GIF, BMP, ICO, PNG, JPG, TIF. This means that you can use the images that you have taken with your smartphone and use them in your Photo Book. These images will need editing as they are rather large, to edit them see Appendix E Editing your images, which covers using your own images section at the end which covers how to resize the images.

#### Smartphone Cameras

Most cameras and smartphones save image files in the JPEG format. These files are a compressed version of the image file that when saved, takes up less storage space than the original image. Depending on the type of compression used, the quality of the image can be reduced, usually this is negligible and cannot be detected by the human eye.

To install the Pillow library for Windows:

1. On the desktop, click the Windows symbol to open the menu.
2. Type the letters, CMD to find the command prompt.
3. Open the command prompt app.
4. Type the command **pip3 install pillow** (figure 4.10)
5. Press the enter key to install Pillow.

**Figure 4.10 Installing the Pillow software**


If you are using Linux then you can install Pillow using the same process,

first open the terminal window and enter the command, `install pip3 pillow`. Press enter to install.

One way to find out what image file types are supported by your computer is to create a Python program to check. Open a new Python file and enter the code:

```
from guizero import system_config print(system_config.supported_i
```

then save the file and run it. Python will print out a list of the supported image file types. Similar to the output below,

```
>>> from guizero import system_config
>>> print(system_config.supported_image_types)
['GIF', 'Animated GIF', 'BMP', 'ICO', 'PNG', 'JPG', 'TIF']]
```

### Upgrading guizero

Depending on when you installed guizero, it may be an older release of the program. This can create issues when you install Pillow. For example, the App no longer displays images! It is also good practice to upgrade software as new features are released and bugs and errors resolved. To resolve this issue simply upgrade the installation of guizero.

If you installed guizero using pip, then you can upgrade guizero in the terminal window using the commands in table 4.3.

**Table 4.3 Upgrading guizero**

| Operating system | Command |
|---|---|
| Windows and macOS | pip3 install guizero --upgrade |
| Linux or Raspberry Pi | sudo pip3 install guizero --upgrade |

## Changing the background color of the buttons

It would be helpful to change the color of the Next and Previous buttons to visually indicate what each button does. For example, you could change the

Next button to green and the Previous button, to Orange. The Close App button could be changed to red, to indicate or warn the user and support them in thinking about what the button does, it will close the Photo Book GUI. The code is simple to use, for example to change the Previous button to orange add the following line of code after button code:

```
back_button = PushButton(app, command=back, text="Previous", alig
back_button.bg = "orange"
```

To change the color of the other buttons, use the same line of code, just change the name of button variable and the color you would like to see displayed.

## Printing your photo

We are going to add some additional code to trigger Windows to print the current photo that is being displayed by the GUI. For example, if the image of the cat is currently displayed, then the GUI will trigger your computer to open the print options and you can send the image to your printer.

There are several stages to follow to add this feature to your program, we will need to:

1. Save a copy of the program file with a new name
2. Import the OS module
3. Create the print function
4. Find your file path to your images
5. Add the file path to the function
6. Create a print button
7. Test

### Importing the os module

Since you already have a working version of the Photo Book, we don't want to overwrite the original program and lose it. Saving the file with a new name means that you will not overwrite your original program code; whatever happens, you will still have your original Photo Book. Begin by saving the file with a new file name. With your original Photo Book program,

Photo_Book.py, open in Python,

1. Select the menu option, **File** > **Save As**
2. Rename the file to **Photo_Book_Print.py**
3. Select the **Save** option

Now we import the Operating System (OS) module. This module enables you to write code that controls features in Windows OS, which means that you can use it to trigger the print function of the OS to load. The module is part of Python which means that if you have a Linux operating system and use `import os`, then it will enable you to control the features of the Linux OS through Python code.

At the top of your program, under the #Imports section which contains the guizero and glob modules, add the following line of code,

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

Save the file before moving onto the next section, select the menu option, **File** > **Save**.

## creating the function to print the photo

Now to create the function that will print the current photo that is displayed in the GUI window. In your program find the #Function heading, then on a new line underneath the last function, the `exit()` function, add the code in listing 4.10.

**Listing 4.10 Creating a print function**

```
def print_to_paper(): #A
    global photo_number #B
    photo = image_file_names[photo_number] #C
    os.startfile(photo, "print") #D
```

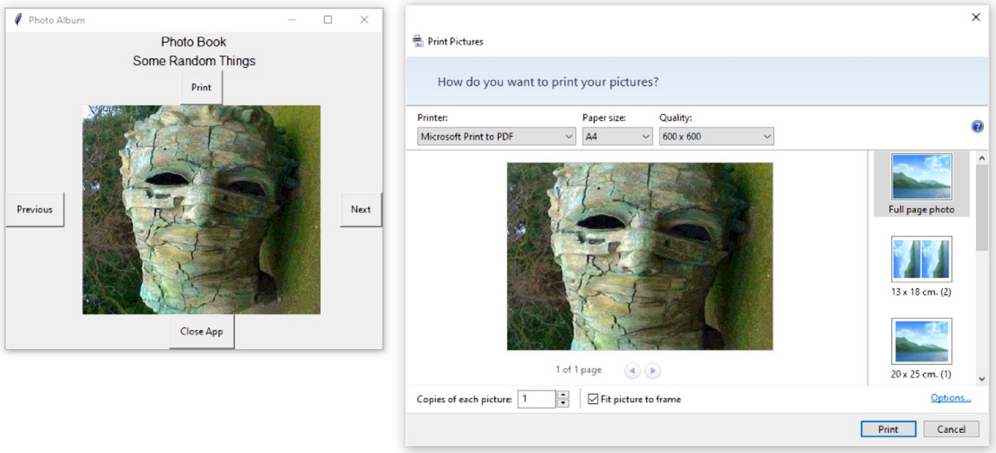
We begin the function by declaring it, basically naming it `print_to_paper()`. Although it makes sense, we cannot name the function,

`print()`. This is because `print()` is already a Python function, which is used to print strings out in a program. Therefore, our function is called `print_to_paper()`.

Next, we pull in the global variable `photo_number` into the function. Global functions are accessible by any part of the program and enables the function to know which number image is currently being displayed. Then we use the value stored in `photo_number` to locate the image file in the list of images.

Finally, we use the code, `os.startfile(photo, "print")` to call the operating system to load and start the print window with the image currently stored in `photo`. Figure 4.11.

**Figure 4.11 The Windows OS print window**




## Using an alternative file path to your images

The `print_to_paper()`function assumes that your Python code is stored in the same folder as the **Photo_Book_Images** folder. If so, then the program will locate the images. What happens if your **Photo_Book_Images** folder is in a different location, or the images are stored in folder with a different name? For example, a shared One Drive folder, (named Family_Photos) that is accessible by your family and they upload their images to it?

To access and print these files, you need to use this folder location so that it can be accessed by the OS. Basically, you are telling Python where the

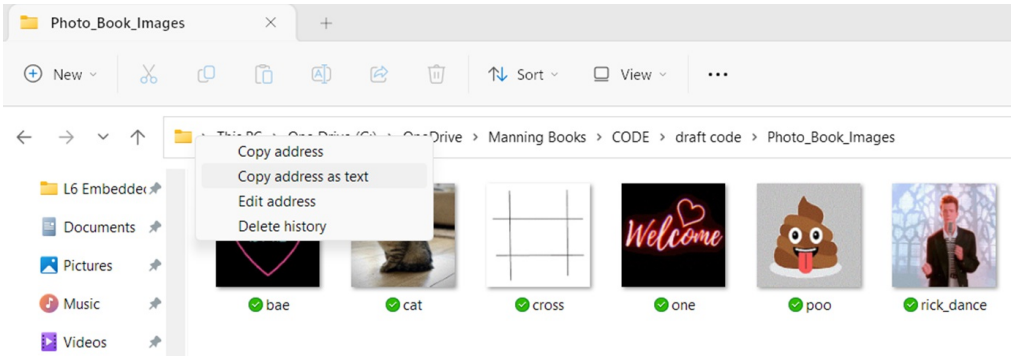
photos are stored and where to find them. This is relatively easy to do as we just need to find the full file path name of where the currently displayed image is stored.

A file path is like a house address that tells the program the exact location where a particular file is stored. Similar to an address, if you miss a line, then it is difficult to find the house, the same applies with the file paths. If you omit part of the file path, then the program cannot locate the folder or the files.

The good news is that Windows OS provides a simple method to ensure that you get the correct file path, and the full file path, figure 4.12.

1. First, open the folder that contains all of your image files, this will be, **Photo_Book_Images** or your folder may have a different name.
2. In the address bar at the top, (which displays the file path) right click anywhere on the address.
3. A menu will load, select the *Copy address as text* option from the list.

**Figure 4.12 Copying the folder address as text**




This will copy the full and correct file path for the folder that contains your images, for example,

```
O:\OneDrive\ManningBooks\CODE\draftcode\Photo_Book_Images
```

**Your file path will be different** from the one displayed above as you will have stored your image files in your own unique location, although you will recognize the `Photo_Book_Images` at the end of the text.

Head back to the Python program and find the line of code where glob searches for the gif images and builds a list of the image names,

```
image_file_names = [f for f in glob.glob("Photo_Book_Images/*.gif
```

Replace it with the following line of code,

```
[CA] image_file_names = [f for f in glob.glob("O:/OneDrive/Mannin
```

Where you replace the underlined code with your own unique file path. Note that in Python the file path is separated with **forward slashes**, so if you copy and paste the file address you will need to change these. Ensure that all the backslashes \ are changed to forward slashes /.

This new line of code is the same as we used in the previous version of the Photo_Book.py except that this time we are providing a different file address or location for glob to search for the image files in. Remember to replace the file path with your own unique file path.

**IMPORTANT File path**

Remember that the file path is unique to your computer and depends on which folders you have created and where they are saved. The O:/OneDrive/ManningBooks/CODE/draftcode is an example.

File paths or addresses can be quite tricky to work with and you may be presented with a red error message if the path is incorrect. In Python this message often appears unrelated to the address. The message usually displayed is, `IndexError: list index out of range.`

If you get this error message it means that the program cannot load an image because it hasn't built a list of them. It cannot build a list because it cannot find the images, as the file path is incorrect. Recheck the file path that you are using and compare it with the correct example code below,

```
[CA] = [f for f in glob.glob("O:/OneDrive/Manning Books/CODE/draf
```

## Adding the button to print

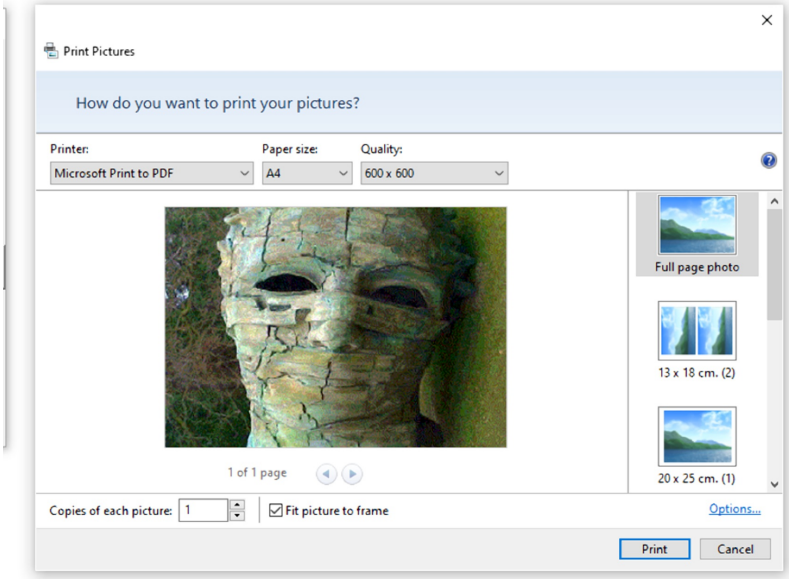
The final part of the code is to create a button that will call the `print_to_paper()` function and trigger the print option window to open. Add the code from listing 4.11.

**Listing 4.11 Adding the final button**

```
[ca] print_button = PushButton(app, command = print_to_paper, tex
```

The code for this button is the same as the previous buttons. First create a variable to hold the code for the button. Assign the button to the `app` window and use the `command` to assign a function to the button. Give the button a label and then use the `align=` code to place the button at the top of the GUI window. Save your program and then press F5 to run it. As with the previous Photo_Book GUI, the first time the program runs all the buttons will initially appear at the bottom of the window. They will rearrange on your first click. Use the forward button to move through the images, when you are on the image that you want to print, press the print button, it will trigger the print option window to open up and you can adjust the settings before printing (figure 4.13).

**Figure 4.13 The print options in Windows 10**




**FINAL CODE LISTING**

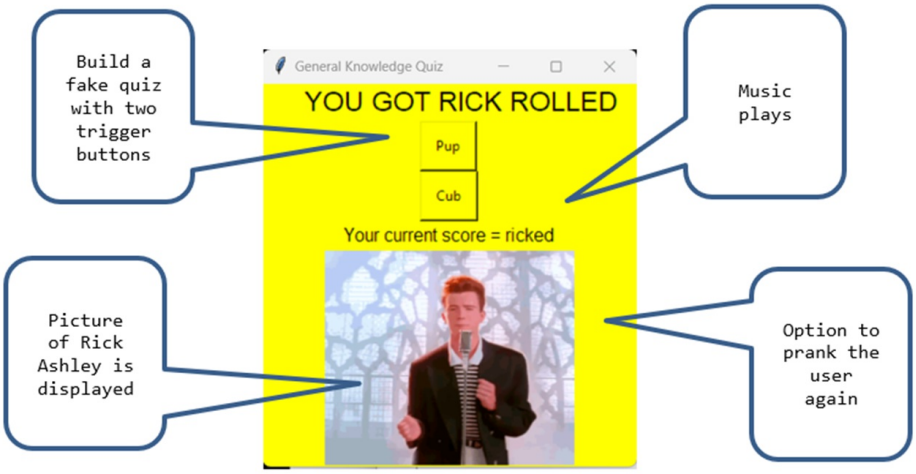
```
# Imports

from guizero import App, Picture, PushButton, Text
import glob
import os

# Variables

global photo_number #photo number
photo_number = 0

# Functions

def forward():
    global photo_number
    global photo
    photo.destroy()
    photo_number = (photo_number + 1) % len(image_file_names)
    photo = Picture(app, image=image_file_names[photo_number])

def back():
    global photo_number
    global photo
    photo.destroy()
    photo_number = (photo_number - 1) % len(image_file_names)
    photo = Picture(app, image=image_file_names[photo_number])

def exit():
    app.destroy()

def print_to_paper():
    global photo_number
    photo = image_file_names[photo_number]
    print(photo)
    os.startfile(photo, "print")

# App

[CA] image_file_names = [f for f in glob.glob("O:/OneDrive/Mannin

app = App("Photo Album", width=475, height=400)
text = Text(app, text="Photo Book")
album_name = Text(app, text="Some Random Things")

photo = Picture(app, image=image_file_names[photo_number])
back_button = PushButton(app, command=back, text="Previous", alig
next_button = PushButton(app, command=forward, text="Next", align
[CA] exit_button = PushButton(app, command=exit, text="Close App"
```

```
[CA] print_button = PushButton(app, command=print_to_paper, text=

app.display()
```

# Statements of fact

- The glob module enables Python to find and access files that are saved on your computer.
- A variable is a location in the computer’s memory which stores information and data.
- Global variables are variables that hold data which can be accessed in any part of the program.
- Modulo division uses the % operator and returns the remainder of a division calculation.
- Indentation is the number of spaces at the beginning of a line of code. Indentation in Python programs is used to indicate a block of code, for example, the code that is part of a function and code that is not.
- The OS module contains functions to enable Python code to control features of the Operating System such as managing files and folders, opening software and printing.
- The `os.startfile()` command opens a file with an associated program. Using startfile on a word document will open the file with MS Word, the associated program

# 5 It’s just a prank, bro!




## This chapter covers

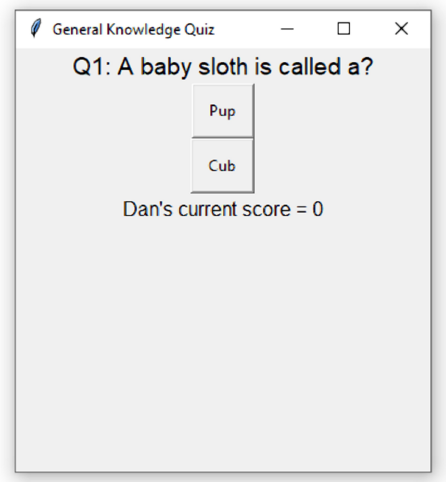
- Selecting a random item from a list
- Calling another function within a function.
- How to use pop-ups
- Responding differently to user actions.

Rick Astley is a famous English singer who has a popular song called “Never Gonna Give You Up.” In May 2007, during an online chat, a user uploaded and shared the song to troll the other users. This began a trend of people sending a file to other people pretending it was something else, when actually, it was the song, “Never Gonna Give You Up.”

For example, you receive a message along the lines of “open this photo of your brother falling over.” When you open the file it plays the song. This prank is known as being Rickrolled, or Rickrolling. Rick has said that even he has been Rickrolled by his own song. He was sent an email from someone claiming to have met him backstage at a concert when they were 12. However, the prankster posted a link to the song instead of a picture of the meeting.

If you want to learn more about the prank, click here: https://www.youtube.com/watch?v=dQw4w9WgXcQ.

**Figure 5.1 A prank quiz, looks like a simple quiz, but it Rickrolls the player.**



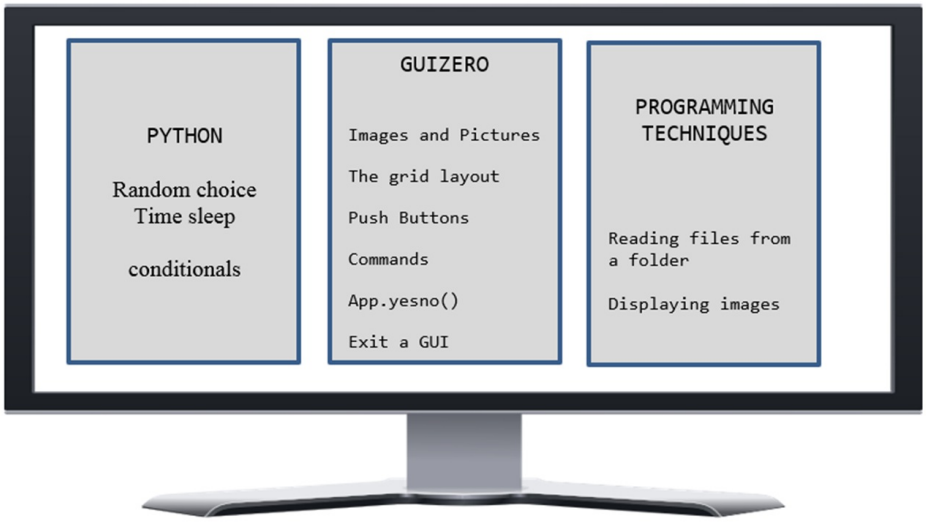

## WHAT YOU WILL BUILD

In this chapter you will build a quiz, figure 5.1. However, this quiz is a fake quiz. It only has one question and two answers. It does not matter which answer the player (I will call them a player as they think that they are playing a quiz) selects, they are both wrong and will trigger a Rickroll. Later you can edit the program and make it into your own version, such as a jump scare, the quiz button triggers a scary sound and then a picture appears, or a happy birthday prank, plays happy birthday.

The GUI will change color, display an image of Rick and most importantly, play the song, “Never Gonna Give You Up”. The following happens:

- The player presses an answer button.
- The audio file is loaded and starts playing.
- The background changes to a random color.
- There is a 2 second pause.
- An image of Rick Ashley is displayed.
- The title changes to, “YOU GOT RICK ROLLED”.
- The Score changes to, “Your current score = ricked”.
- The player is asked if they want to be pranked again.

You have pranked the player. They have been Rickrolled!

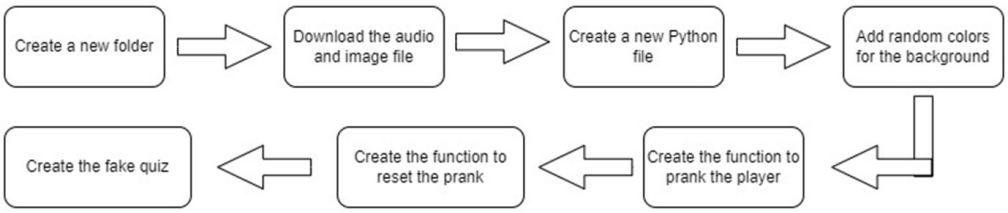



# Creating the project

Before you begin the project, you will need to download two files, but first, let's look at the steps required to build the project, shown in figure 5.2. The main steps are,

1. Create a new folder.
2. Download the two files into the new folder / copy the files into this folder.
3. Start a new Python file and import the required modules.
4. Select some colors for the background.
5. Create a function to perform the Rickroll (play music and show a picture).
6. Create a function to repeat the prank.
7. Create the main GUI window to hold the title, image, and buttons.
8. Create the fake quiz.

**Figure 5.2 Flow diagram of building the prank quiz.**




## Downloading the Audio and Image File

If you built the soundboard in Chapter 2 then you will already be familiar with using audio clips in guizero. Before you start this project, download the required audio file and the image file of Rick. The audio file has been cut to the required length to play enough of the song to perform a Rickroll. The song is purely for educational use and learning, and the full ownership of the song belongs to Rick Astley.

When using audio with guizero you need to ensure that the file is stored in the same folder that you have saved the Python program file in.

To do this, let's begin by opening the folder where you are saving your GUI projects,

1. Create a new folder and name it **Prank**.
2. Open your web browser and enter this address: https://github.com/TeCoEd/Twisted_Python_Projects/tree/main/Project_
3. Click on the folder named prank sounds and download it.
4. Open the downloaded folder and extract both the audio and the image files.
5. Copy and paste, drag, or save both files into the **Prank** folder.

You are now ready to write the program code and create the fake quiz.

1. Load your Python editor.
2. Create a new program file in your editor.
3. From the **File** menu, choose **File** > **New File**.
4. A new window opens. Choose **File** > **Save**. The Python editor prompts you to save the file.
5. You must save the Python file into the same folder that also holds the audio and image file. If you do not do this, the GUI will not be able to

find the sound files.

6. Name your Python file **Prank.py** and choose **Save**.

## Importing the modules

Like the previous projects, this one begins with the import of all the required modules. Add the code from listing 5.1 into your Python file.

**Listing 5.1 Importing the modules**

```
# Imports

from guizero import App, Box, Text, PushButton, Picture
from random import choice
from time import sleep
import winsound
```

The first set of modules on line one will be familiar, we have used them so far in all the previous projects. The modules enable you to create the GUI window, add text, buttons, and images. Line two introduces a new feature from the `random` module, `import choice`.

In chapter two we used the `random` module to select a random joke from a list of jokes. Each joke was assigned a number based on its position in the list, for example, joke0, joke1, joke2 and so on. The random module was used to select a random number which told the program the number of the joke to display.

In this project we are going to use `from random import choice`. The `choice` allows Python to select a random item from a list. This method is more efficient since each item is going to be a single word, rather than a long joke containing lots of words.

We will use the `choice` to change the background color of the GUI to a random color each time that the player is pranked (they are Rickrolled). This makes the prank colorful, eye catching and fun. We create a list of colors and then use `random choice` to select a random color from the list. This color is then set as the background color.

On the third line we import the `time` module. This is the first time that we have seen the `time` module and it allows us to control time. Well, sort of. We can use the line of code `time.sleep(2)` to add a two-second delay before the picture of Rick is displayed in the GUI. This delay is purely artistic so that the person you prank hears the intro drums, and then as the main song kicks in, is presented with Rick's happy face. (If you know the song, you will know it makes sense!) The last module that we will use is `winsound`, which is included in the Python installation files and provides basic access to the software in Windows OS that plays sound.

If you are using a Linux or Mac OS then you will need to use the alternative sound module used in Chapter 3, simpleaudio https://pypi.org/project/simpleaudio/. The full code listing can be found in listing 5.10.

## Choosing a random color

Now to decide and set up the colors that the background of the GUI will change to. When the GUI first loads the background is the default grey. The color of the background changes each time the player is pranked, and the rick roll runs. The aim is to make the interface look more exciting and original, rather than having the standard gray background.

To do this we create a new variable called `colors`, and then add the names of the colors that you want to use, into a list. For example, to add the color yellow, simply add it to the list, "yellow"; to add white, do the same thing, "white".

#### Variables

A variable is a location in the computer's memory that our Python code can access. In this memory location, we are storing the names of the colors. In order to make it easy to find the location we label the variable: `colors`, so that the program can find the location.

You can add as many colors as you want to, and this website has a list of the names of all the colors that you can use with guizero and Python:

https://wiki.tcl-lang.org/page/Color+Names%2C+running%2C+all+screens.

There are easily well over one hundred names, so you will be able to pick the ones that you like. Under the import lines of code, add the code in listing 5.2 and select your colors.

**Listing 5.2 Creating a list of colors**

```
# Variables

colors = ("yellow", "green", "pink", "white", "orange")
```

## Creating the two main functions

The next step is to create two functions that the program uses. As you may recall from previous chapters, a function is a reusable block of code that can be called and used anywhere within a program. Functions save time and make the program more efficient. In this program, the first function Rickrolls the player, and the second function repeats the prank so the player is Rickrolled again. Each function is held as a separate function. The first function is called when the player presses either one of the answer buttons. The following happens:

1. The Never Gonna Give You Up audio file is loaded and starts playing.
2. The background changes to a random color.
3. There is a 2 second pause.
4. An image of Rick Ashley is displayed.
5. The title changes to, “YOU GOT RICK ROLLED”.
6. The Score changes to, “Your current score = ricked”.
7. The second function `ricked_again()` loads and asks you if you want to be pranked again, then responds to the player’s choice.

When you create the function, you need to ensure that you use the correct indentation. The colon symbol `:` at the end of a line of code means that the next line of code will be indented. Most Python editors will automatically indent the next line of code when the previous line ends with a : symbol. One of the challenges when writing longer functions is ensuring that the indentation levels are correct and match.

**Indentation reminder**

Indenting refers to the use of white space before the code, which means the text starts a little distance away from the left-hand margin. Indentation is used so that Python knows which lines of code are part of a function, and which lines are not and belong to the rest of the program.

After you create a function name (for example, `prank_me_play_music():`, the lines of code that follow underneath are indented. This is so Python knows these lines of code are part of the function `prank_me_play_music():` To use the correct amount of indentation you can press the TAB key once, or the spacebar four times. Ensure that you always use the same amount of indentation throughout your program. Add the function from code listing 5.3 and ensure that you check you have used the same level of indentation, (either by pressing the TAB key once or, pressing the space bar four times.)

**Listing 5.3 Creating the first function**

```
# Functions

def prank_me_play_music(): #A
    winsound.PlaySound("Rick.wav", winsound.SND_ASYNC) #B
    app.bg = choice(colors)#C

    sleep(2)#D
    rick_pic = Picture(app, image="rick_dance.gif") #E

    [CA]title_text = Text(answer_grid, "  YOU GOT RICK ROLLED"
    title_text.text_size = 18
    [CA] score_text = Text(answer_grid, "  Your current score = r
    ricked_again() #H
```

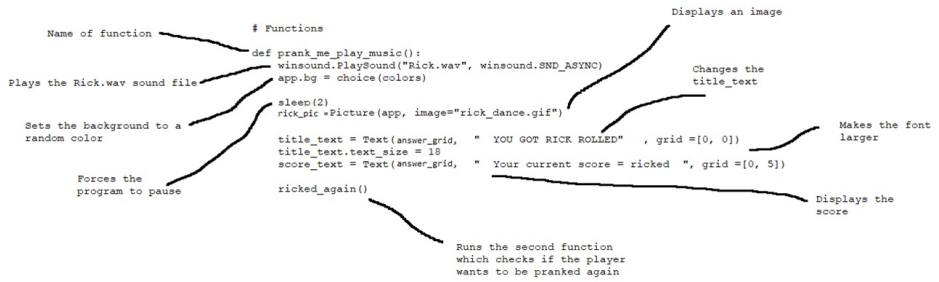
When writing any function, we start on the first line by defining the name of the function so that it can be called and used elsewhere in the program. This function is defined as `def prank_me_play_music`(): as its name reflects the fact that the function runs when you prank the player and music plays.

On the next line we use the code, `winsound.PlaySound()` followed by the name of the audio file, (Rik.wav), which tells winsound to play the audio file. This is a short clip of the intro of the song, Never Gonna Give You Up.

**What is winsound**

The winsound module allows Python to use the Windows sound functions to play audio clips.

**Figure 5.3 How the prank_me_play_music() function works.**



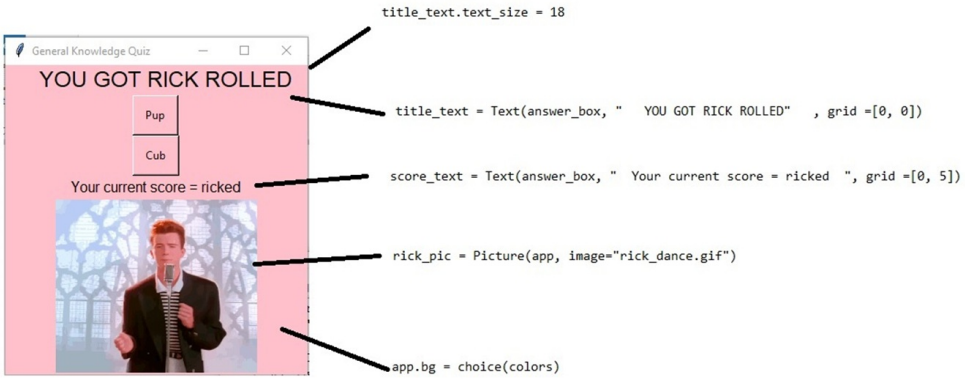

The third line of code manages the background color. The function `choice()` makes a random choice from the list of colors held in the `colors` variable from listing 5.2. The program then sets the background to that color. For example, if the random choice is yellow, it will change the background of the GUI window to yellow. The background change is simply part of the prank. Then, we program In a short pause of 2 seconds using `sleep(2)` before displaying the image of Rick using `Picture()`.

Next, we use the `Text()` widget to set a caption at the top of the GUI that displays, “YOU GOT RICK ROLLED”, and make the font bigger using the line of code, `title_text.text_size = 18`. This makes the caption stand out telling the player that they have been Rickrolled!

**answer_grid**

The answer_grid is a variable that holds the information about where the answer buttons will be displayed in the GUI window.

**Figure 5.4 Features of the Prank quiz**



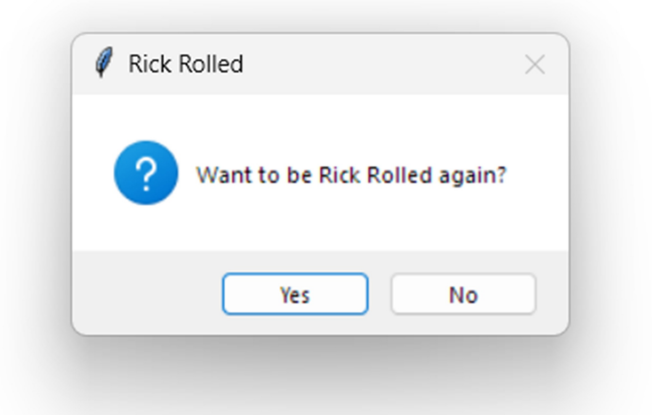

Finally, we change the quiz score from zero to ricked, which is not a score but emphasizes the Rickroll, just in case the player had not realized what was going on!

On the last line of the function, we call the second function, which we will write in the next section. The second function is called `ricked_again()` and asks the player if they want to be pranked again. If they select yes, then the `prank_me_play_music()`: is called and runs again and the user is pranked again. If they select no, then the GUI closes.

## Creating the ricked_again() function

This function introduces you to the `app.yesno()`Pop-up, figure 5.5, which is used to ask the player if they want to be pranked again. The yes/no option is so common, that guizero includes a template for you, you just need to supply a question, and guizero then does all the work for you and creates the popup window and buttons. In our program the player is presented with a Pop-up window that asks them if they want to be Rick-rolled again. The window then presents them with two options, Yes or No. Figure 5.5.

**Figure 5.5 The Yes, No Pop-up window.**



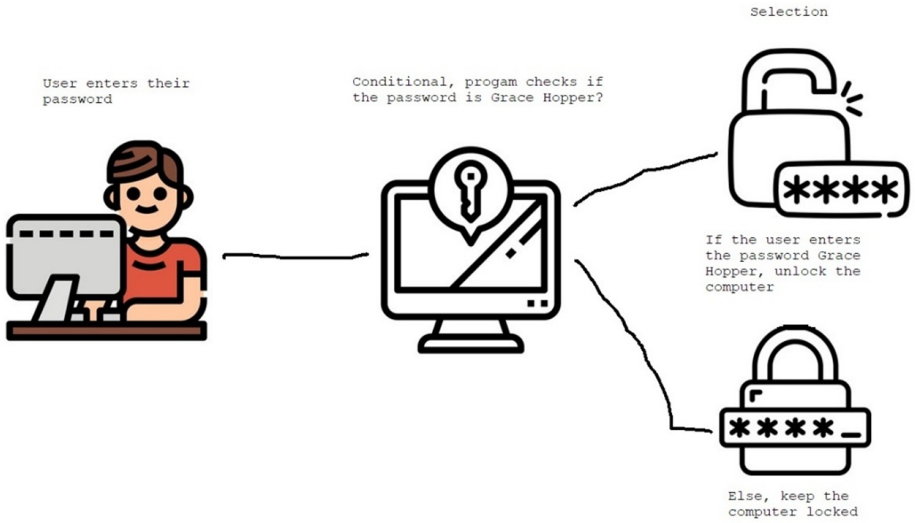

We then use a programming technique called selection, which enables the program to select how it responds when the player presses either of the buttons. Selection uses a conditional that checks if a certain condition is met, and if it is met, then the program responds with a particular response. A condition tells a program to execute an action depending on whether the condition is true or false.

For example, in this game the player is asked if they want to be Rick Rolled again, they can respond with Yes (True) or No (False). If they select Yes then the prank repeats and plays again, if they select No, then the GUI closes. You have seen and used conditionals when you enter a password to log into a computer(see figure 5.6)

**Remember**

The person writing the code must write both the condition and the responses. The program only knows how to respond because we have told it how to.

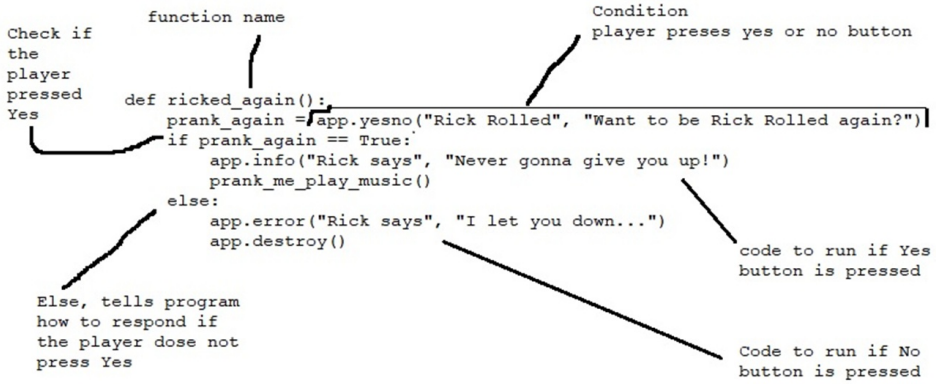
**Figure 5.6 How a condition and selection are used to check a user password.**




In this function, if the player selects Yes, (True) then the condition is met, and a the `prank_me_play_music()` function is called to run the prank again. Should the player select the No option (False), then the condition is not met, and the program responds differently using `app.destroy()`to close the GUI.

In this program, we use an `if` Statement to check if a condition has been met or not met. This statement uses the word `if` to check if a condition has been met. Then the word `else` is used to tell the program what to do if the condition has not been met. The `if` statement also includes the lines of code that tell the program what to do depending on the condition being met, or not being met as shown in figure 5.7. You will learn more about if statements and how they work in chapter 6 Are you a Mind Reader?

**Figure 5.7 The components of an if statement in our program.**




As with the previous function we begin by naming the function, this one is called `ricked_again()` as the function provides the option to be pranked again. On the next line we use `app.yesno()` which creates the popup window presenting the option to choose to run the prank again or close the program. Listing 5.4

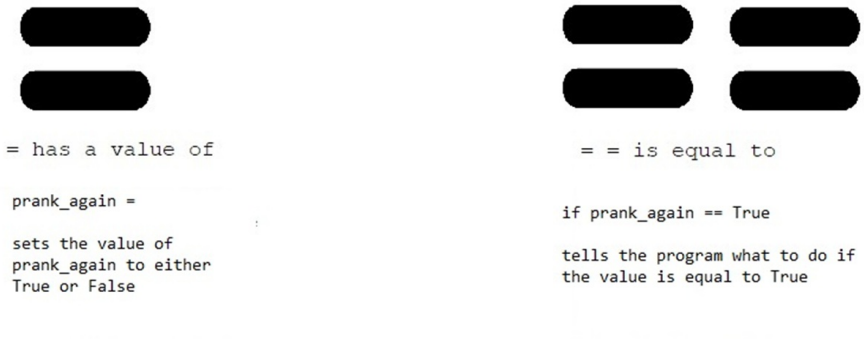
**Listing 5.4 Creating the second function**

```
def ricked_again(): #A
    prank_again = app.yesno("Rick Rolled", "Want to be Rick Rolle
    if prank_again == True: #C
        app.info("Rick says", "Never gonna give you up!") #D
        prank_me_play_music() #E
    else: #F
        app.error("Rick says", "I let you down...") #G
        app.destroy() #H
```

On the third line we code the condition using an if statement as discussed in the previous chapter, followed by `== True:` This condition checks that the player pressed Yes. The program is basically checking, is it true that the player pressed the yes button? When the Yes button is pressed, the response True is saved into the `prank_again` variable, so that the response can be checked, and the correct condition applied.

In Python, = means has a value of. It is used with variables, list and assign codes and functions (`prank_again = app.yesno….`). To check if something is equal, Python uses two equal signs ==, figure 5.8.

**Figure 5.8 The meaning of the symbols, = and ==**




Once the program has confirmed that the player pressed the Yes button, then the `app.info()`widget is used to load a popup box that displays the message that “Rick is never gonna give you up”. The program then calls the previous `prank_me_play_music()`function which runs again, playing the song intro, making the background a different random color and displaying the picture. Once the prank ends, the player is again given the option to run the prank again.

When the No button is pressed, the if statement loads a different popup box that displays a message that Rick “let you down” (another reference to the song lyrics) This popup is an error Pop-up window which uses a red banner to signify the fact that the app is closing. The player then presses the OK button, and the program runs the code `app.destroy()`and the program closes the GUI. To start the prank again you need to run the program again.

## Building the main app

This section of the chapter covers how to build the main GUI app. If you have already built some of the other projects, the structure will be familiar to you. We begin by creating the main app window, naming it, and then setting the window’s size. The width and height values are measured in pixels and are set so the image file fits neatly in the main GUI window. You can resize the width and height if you require; simply change the values. Listing 5.5.

The second line of code creates a popup box using the function `app.question()` that asks the player to enter their name. Their name is used

to personalize the quiz and display the player's current score (even though this is a fake quiz, and they never score anything, it makes the quiz appear more realistic). Remember the `app.yesno()` Pop-up that we previously created? Well, guizero does a similar thing with `app.question()` the widget automatically creates the question box and displays a question, creates a space for the player to enter in the answer, then saves their input. The pre-built components are being used to support achieve our rick-rolling dreams!

The last line of code uses the `Box` object to create a space to display two answer buttons. In guizero, the Box object is a container which can hold other widgets. Unlike the App window, a box is invisible. Using the Box neatly lays out widgets and elements contained inside it. In this project, the Box neatly lays out the buttons. If we use the App window instead of the Box, it will just stack the buttons vertically on top of each other.

Add the lines of code from listing 5.5.

**Listing 5.5 Setting up the GUI**

```
# App

app = App("General Knowledge Quiz", width=330, height=338) #A
your_name = app.question("The Quiz", "What's your name?") #B
answer_grid = Box(app, layout = "grid") #C
```

In the next part of this section of the program code, listing 5.6 we use code `Text()` to create and display the fake quiz question. The question is stored as a string; you may be familiar with strings from the other chapters, they are single character or a series of characters, numbers or symbols that are displayed or printed out in the program.

We use a string to hold and display the quiz question, "What is a baby sloth called?", which is a real question about what a baby sloth is called. We add the question inside the `Box()` which keeps the layout neat and tidy.

Next, we use `Text()` again, to create another string that will display the score. This string combines the name that the player enters (from listing 5.5), and was saved into the `your_name` variable, with the text current score = 0. Let's say for example, that your name is Amy. The combined string would be

“Amy’s current score = 0”. Remember that the score is not real and is simply there to make the quiz look more realistic.

Finally, we set the font size and the style of the font. You can choose a different font by replacing the current font name with the name of your preferred font. We have used Comic Sans as it is a funny looking font that, change the fun is just to make the GUI look cooler.

Add the code from listing 5.6.

**Listing 5.6 Adding the question and score**

```
[CA]title_text = Text(answer_grid, "Q1: A baby sloth is called a?

[CA]score_text = Text(answer_grid, your_name + "'s current score
title_text.text_size = 14
title_text.font = "Comic Sans"
```

## Creating the two answer buttons

The final part of the program is to code the two answers buttons that when either is pressed will call the prank function and trigger it to run, Rickrolling the unsuspecting quiz player.

**Listing 5.7 Creating the answer buttons**

```
[CA]answer1 = PushButton(answer_grid, command=prank_me_play_music

[CA]answer2= PushButton(answer_grid, command=prank_me_play_music,

app.display()
```

First, we start by creating an instance of the button object using the name `answer1` this holds all the code about the button, what it does, how it looks. Then create a button using the `PushButton()`code. We need to tell the program to display the button neatly inside the `answer_grid`, which makes use of the `Box()`layout, and then we use the command `command=prank_me_play_music` to tell the program to call the function that pranks the player. This is the function we created in listing 5.3 that plays the music, changes the color etc.

Next, add a label to the button that displays the answer to the question about Sloths, for example, Pup. A label is added so that the player thinks they are selecting an answer to the question. Finally, use the `grid=[]` code to align the first button. All these parts create a line of code that displays a button that when pressed triggers the Rickroll.

The second button follows the same code as the first and is used to create a second answer option for the question. (This provides the player with two answers, although we know that pressing either button will trigger the prank). However, we need to change the label to ‘Cub’ and use different values on the `grid=[]` to place this button underneath the previous.

Add the lines of code from listing 5.7.

## Testing the program

Well done, you have now built a fake quiz, which is really a prank that Rickrolls the unsuspecting player. Well done. Now to test that it works correctly. Press **F5** on the keyboard. You may be prompted to save the program file, choose **OK**, the program will save, then the program will run. If you encounter any errors this time, check the Python editor for error messages and then check for these common errors.

1. Ensure that the volume on your speakers is turned up.
2. Is the audio file saved in the correct location? The Rik.wav file needs to be saved in the same folder as the Prank.py program file.
3. Have you used the correct levels of indentation in the functions?
4. When creating the buttons, have you used the same function name in the command?

**Listing 5.8 Final Code**

```
# Imports

from guizero import App, Box, Text, PushButton, Picture
from random import choice
from time import sleep
import winsound
```

```
# Variables

colors = ("yellow", "green", "pink", "white", "orange")


# Functions

def prank_me_play_music():
    winsound.PlaySound("Rick.wav", winsound.SND_ASYNC)
    app.bg = choice(colors)

    #shows a picture of rick
    sleep(2)
    rick_pic = Picture(app, image="rick_dance.gif")
    [CA]title_text = Text(answer_grid, "   YOU GOT RICK ROLLED"
    title_text.text_size = 18
    [CA]score_text = Text(answer_grid, "  Your current score = ri

    ricked_again()

def ricked_again():
    prank_again = app.yesno("Rick Rolled", "Want to be Rick Rolle
    if prank_again == True:
        app.info("Rick says", "Never gonna give you up!")
        prank_me_play_music()
    else:
        app.error("Rick says", "I let you down...")
        app.destroy()
# App

app = App("General Knowledge Quiz", width=330, height=338)
your_name = app.question("The Quiz", "What's your name?")
answer_grid = Box(app, layout = "grid")

[CA]title_text = Text(answer_grid, "Q1: A baby sloth is called a?
[CA]score_text = Text(answer_grid, your_name + "'s current score
title_text.text_size = 14
title_text.font = "Comic Sans"

[CA]answer1 = PushButton(answer_grid, command=prank_me_play_music
[CA]answer2= PushButton(answer_grid, command=prank_me_play_music,

app.display()
```

# Other things to try.

As with the previous chapters this section introduces an additional change that you can make to your program. Since this GUI is actually a prank and not a real quiz, then the main change we will make is to display a different question and additional answer buttons.

## Changing the question and the answer buttons

Since the GUI is supposed to be a quiz, you can change the question and answers to a different one of your choice. For example, you could ask the question, which food do I not like? and then present the player with three answer buttons. When pressed, each answer button still triggers the prank to run.

When using more than two buttons, you'll need to increase the length of the GUI window. This is because each button takes up space and pushes the picture of Rick further down the GUI window.

Figure 5.9 shows the GUI with three buttons, and you can see that Rick has lost his legs, the image has been cut off as the GUI needs to be longer to display the full image. You can also see that the text, "Your current score = ricked", is now shown after the second button and not the third. The players' current score is also still shown.

**Figure 5.9 Rick is missing his legs!**

To ensure that your new GUI works as intended, I would recommend that you make the following edits in this order:

1. Save your program with a different name to avoid overwriting and losing your original GUI code.
2. Change the question.
3. Add the code for the additional buttons.
4. Change the layout of where the score is displayed.
5. Change the layout of where the prank score is displayed.
6. Run the program to see how much of the image is missing.
7. Change the length of the height of the GUI window.
8. Run the program again, adjust as required.

**Making a new copy of the program**

The first step is to save your file with a new name, this means that you still have your original working version if things go wrong or don't work as intended. To do this, follow the instructions below,

1. Open your **Prank.py** program code in your Python editor.
2. From the **File** menu, choose **File** > **Save As...**.
3. A window opens displaying your folder where the **Prank.py** file is currently saved.
4. Select and change the name of the file to **Prank2.py.**
5. Press the **Save** button.

You now have a new version of the file which you can edit.

**Changing the question and answer button code**

Find the line of code that contains the original question and replace it with the new questions. You only need to change the question (underlined in listing 5.8), not the code. You can always run your GUI program to check that the question is displayed correctly before you edit the answer buttons.

If the answer to your new question only has two options, then simply keep the same two answer buttons and just change the labels to display the new

answers to your new questions. If you want to add more answer buttons, then, in your program, then use this line of code:

```
[CA]answer1= PushButton(answer_grid, command=prank_me_play_music,
```

Add the code underneath the `answer2` button code, shown in bold in listing 5.8. Change the name of the variable to `answer3`, change the label to the answer, and then set the grid value to `grid =[0, 6]` this places your new answer button neatly underneath the previous buttons.

**Listing 5.8 Changing the question and adding an extra answer button**

```
[CA]title_text = Text(answer_grid, "Q1: Which food do I not like?

[CA]score_text = Text(answer_grid, your_name + "'s current score
title_text.text_size = 14
title_text.font = "Comic Sans"

[CA]answer1= PushButton(answer_grid, command=prank_me_play_music,
[CA]answer2= PushButton(answer_grid, command=prank_me_play_music,
[CA]answer3= PushButton(answer_grid, command=prank_me_play_music,
```

## Changing the scores

Next, we need to change the place where the GUI displays the score. The value that indicates the location is easy to calculate, as the new location is underneath the last answer button. The last answer button is stored at [0,6] which means that the new position for the score is now [0,7]. In your program code, find the line of code that contains your new question, underneath it, is the code that displays the score. Change the grid value to `grid=[0,7]` as highlighted in bold and shown in listing 5.9.

**Listing 5.9 Moving the grid location of the score**

```
[CA]title_text = Text(answer_grid, "Q1: Which food do I not like?

[CA]score_text = Text(answer_grid, your_name + "'s current score
title_text.text_size = 14
```

When the prank runs it changes the score to display the text, Your current

score = ricked. This text also needs to be moved to the same location as the original score text that you moved in the previous paragraph.

In the `prank_me_play_music()` function, find the line,

```
score_text = Text(answer_grid, "Your current score = ricked", gri
```

Change the grid layout from, `grid =[0,5]` to `grid =[0,7]`so that it matches the same values used for the score in listing 5.9. For example, if you set the value to `grid =[0,11]` then replace the grid value with the same value, `grid =[0,11]`.

## Changing the height of the gui to display all of the image

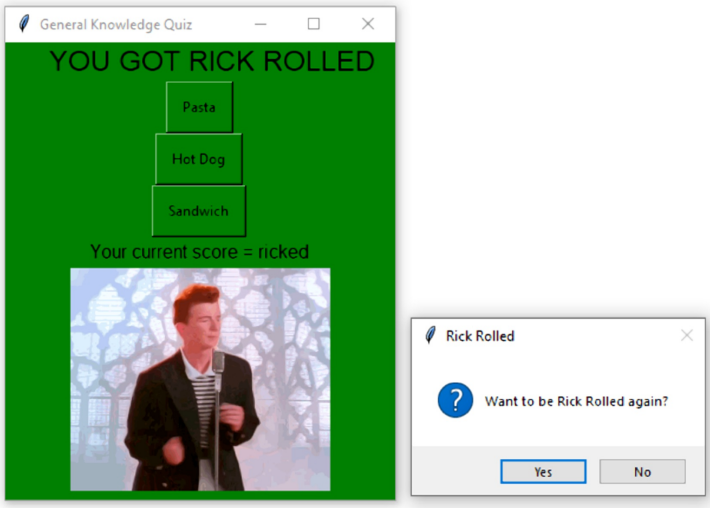
Now to check how much of the picture of Rick has been cut off or is missing from the GUI window. Run the program, (Press **F5** on your keyboard, you may be prompted to save the file first) enter your name and then select one of the answers buttons. The prank will run, look at the image displayed in the GUI and estimate how much is missing. In figure 5.9 about 15% has been cut off. Select the No option when asked if you want to be pranked again and return to your program code. Find the `#App` section and the line of code,

```
app = App("General Knowledge Quiz", width=330, height=338)
```

The code `height=338` sets the height of the GUI window, which currently is too small to display the additional answer buttons and all the image. You can use some simple math to calculate that 15% of 338 is approximately 50, so we need to add 50 to the current high value. Change the new height value to `height=388`.

If you have added more than three buttons, then the easiest method to set the new height is to try some height values until you have a good fit. Figure 5.10. Remember that you will also need to change the score grid values so that they are still at the bottom of the GUI window under the buttons.

**Figure 5.10 GUI with correct sizes and extra buttons**




Save your program and run it to check that all the elements are working correctly, and that they are all displayed where you intended. Adjust as required, and then run the program again and check. Repeat as required. Once you are happy, you are now ready to prank your friends and family with your new customized quiz.

**Listing 5.10 Updated final example code**

```
# Imports

from guizero import App, Box, Text, PushButton, Picture
from random import choice
from time import sleep
import winsound

# Variables

colors = ("yellow", "green", "pink", "white", "orange")


# Functions

def prank_me_play_music():
    winsound.PlaySound("Rick.wav", winsound.SND_ASYNC)
    app.bg = choice(colors)

    #shows a picture of rick
    sleep(2)
    rick_pic = Picture(app, image="rick_dance.gif")
```

```
    title_text = Text(answer_grid, "   YOU GOT RICK ROLLED"   , g
    title_text.text_size = 18
    score_text = Text(answer_grid, "  Your current score = ricked

    ricked_again()

def ricked_again():
    prank_again = app.yesno("Rick Rolled", "Want to be Rick Rolle
    if prank_again == True:
        app.info("Rick says", "Never gonna give you up!")
        prank_me_play_music()
    else:
        app.error("Rick says", "I let you down...")
        app.destroy()
# App

app = App("General Knowledge Quiz", width=330, height=388)
your_name = app.question("The Quiz", "What's your name?")
answer_grid = Box(app, layout = "grid")
title_text = Text(answer_grid, "Q1: Which food do I not like?", g
score_text = Text(answer_grid, your_name + "'s current score = 0"
title_text.text_size = 14
title_text.font = "Comic Sans"

answer1= PushButton(answer_grid, command=prank_me_play_music, tex
answer2= PushButton(answer_grid, command=prank_me_play_music, tex
answer3= PushButton(answer_grid, command=prank_me_play_music, tex

app.display()
```

# Statements of fact

- `choice()`is a feature of the Python `random` module that picks a random item from a list.
- A Pop-up is a window that interrupts the user by asking question or providing information.
- The `yesno()` is a premade Pop-up in guizero which displays yes and no options. The user can press Yes which returns a value of True. Pressing No, returns the value, False.
- The `info()` Pop-up is a premade popup in guizero that displays information and the Windows information icon.
- An `error()` Pop-up is a premade popup in guizero that displays a

message with the error icon.
- Indentation is the number of spaces at the beginning of a line of code or is used to indicate selection code, for example, the code that is part of the IF statement.
- A conditional statement checks if a condition has been met and then responds.
- Conditions can check values of a number, text and Yes and No (True / False) responses.
- Selection is where the program selects a particular action or outcome based on a condition. It is often referred to as, an If Statement.
- An If statement checks if a condition is met and then responds. Else, is used to respond when a condition is not met.

# Using Simple Audio

If you are using a Linux or Mac OS operating system, then you will need to use an alternative sound module to winsound. This is because winsound is only packaged and available with Windows OS. We can use simple audio instead.

This module is described as providing cross-platform, dependency-free audio playback capability for Python 3 on MacOS, Windows, and Linux. So, it provides an alternative if you are not using Windows OS. You can read more about simpleaudio here, https://pypi.org/project/simpleaudio/

**INSTALLATION**

Download and installation of simpleaudio can be completed in one command from the Terminal window. Open your Terminal windows and enter the code.

```
pip install simpleaudio
```

Wait for the program to download and install, and you are ready to make disgusting FART sounds, well, your GUI is! The program code, as the name suggests, is simple to use, first you import the audio module using the line, `import simpleaudio as sa`. Then you create an instance of the audio file and assign it to a variable.

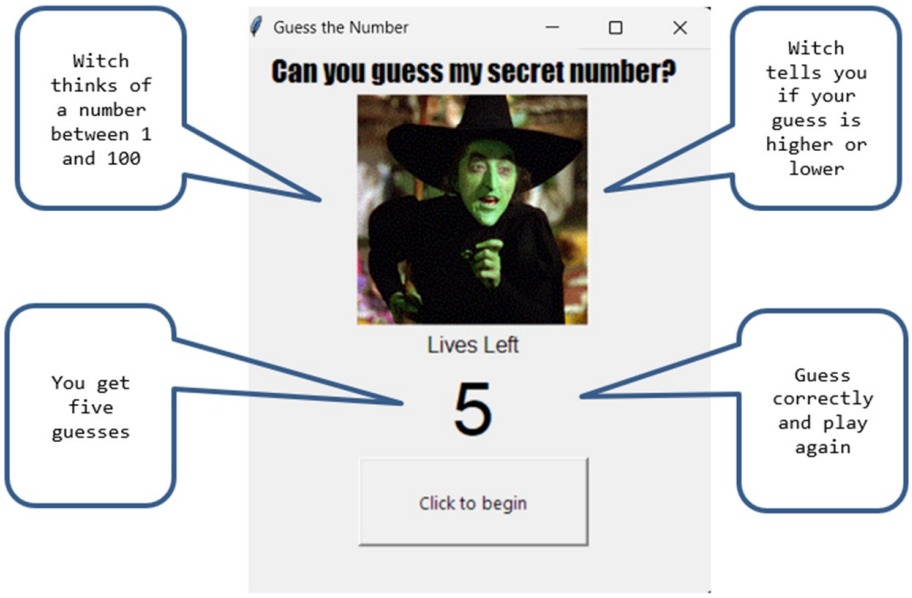
You can now control when the sound plays using the code, `play_obj = wave_obj.play()`. The program needs to wait for the sound to play through before moving to the next line of code, to do this use the code `play_obj.wait_done()`

**Listing 5.11 Alternative OS listing, Final code**

```
# Imports

from guizero import App, Box, Text, PushButton, Picture
from random import choice
from time import sleep
import simpleaudio as sa


# Variables

colors = ("yellow", "green", "pink", "white", "orange")


# Functions

def prank_me_play_music():
    wave_obj = sa.WaveObject.from_wave_file("Rick.wav")
    play_obj = wave_obj.play()
    play_obj.wait_done()

    app.bg = choice(colors)

    #shows a picture of rick
    sleep(2)
    rick_pic = Picture(app, image="rick_dance.gif")
    [CA]title_text = Text(answer_grid, "   YOU GOT RICK ROLLED"
    title_text.text_size = 18
    [CA]score_text = Text(answer_grid, "  Your current score = ri

    ricked_again()

def ricked_again():
    prank_again = app.yesno("Rick Rolled", "Want to be Rick Rolle
    if prank_again == True:
        app.info("Rick says", "Never gonna give you up!")
        prank_me_play_music()
    else:
        app.error("Rick says", "I let you down...")
        app.destroy()
```

```
# App

app = App("General Knowledge Quiz", width=330, height=338)
your_name = app.question("The Quiz", "What's your name?")
answer_grid = Box(app, layout = "grid")

[CA]title_text = Text(answer_grid, "Q1: A baby sloth is called a?
[CA]score_text = Text(answer_grid, your_name + "'s current score
title_text.text_size = 14
title_text.font = "Comic Sans"

[CA]answer1 = PushButton(answer_grid, command=prank_me_play_music
[CA]answer2= PushButton(answer_grid, command=prank_me_play_music,

app.display()
```

# 6 Are you a mind reader?




## This chapter covers

- Checking if an input matches a value
- Creating a while loop
- Conditional logic with ‘if’ an ‘else’
- Giving a different response based on a condition
- Using else if (elif) and else to respond

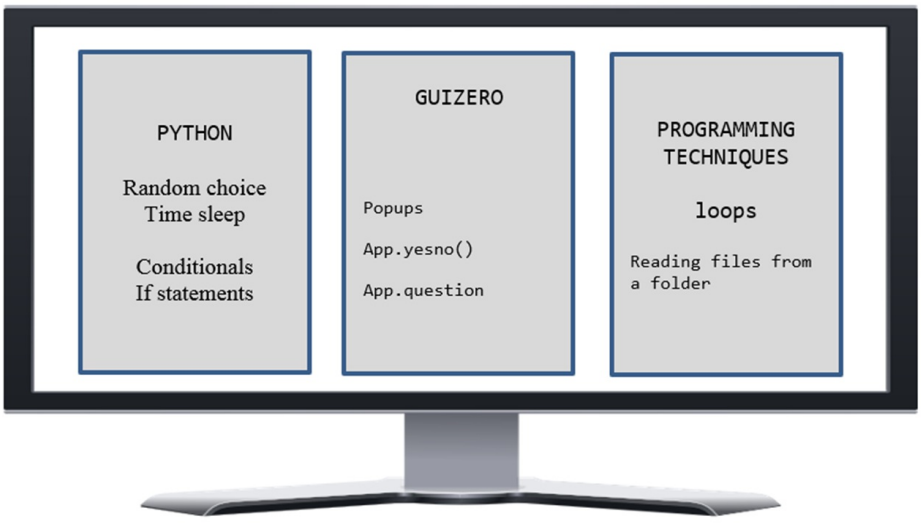
A classic binary search guessing game (higher or lower), “I am thinking of a number between 1 and 100. Can you guess what it is?” You are then given five chances to guess the number correctly. Before you begin this chapter, you may want to challenge someone to play the game as a reminder about how it works. Or we can play, I am thinking of a number between 1 and 100, what is your first guess….!

### WHAT YOU WILL BUILD

In this chapter you will build a version of the higher or lower number

guessing game where a witch or other character of your choice has chosen a number between 1 and 100. They have given you five lives or chances to get it right, if you do guess correctly, then you are free to go. You have won. Unfortunately, if you do not guess the correct number in five tries, then you lose. On the positive side, if you do guess the number correctly, the witch will invite you to try again, if you dare! Figure 6.1

**Figure 6.1 Can you guess the number and beat the witch?**

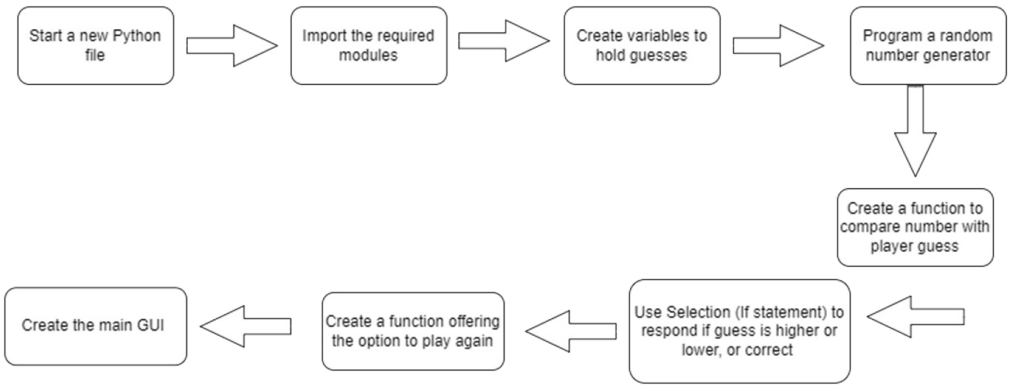



# Creating the project

For this project you will need a suitable image file to display in the GUI. This is a picture of the challenger whom you are playing the game against. You can either download the witch image (this is covered later), or source your own image. The witch (or image of a witch) is who you are playing against and trying to guess their number. They will choose a random number between 1 and 100. Before we begin, let's look at the steps required to build the project (figure 6.2). The main steps are:

1. Source and save an image for your GUI.
2. Start a new Python file and import the required modules.
3. Create variables to hold the random number and track the guesses.
4. Create a function to compare the player's guess.
5. Code an IF statement to respond to the number the player inputs.
6. Create a function that gives the player the option to play again if they win.
7. Create the main GUI window to hold the title, image, and buttons.
8. Test the project.

**Figure 6.2 Flow diagram of building the MindReader quiz.**




## Setting up the folder

If you have been working through the other chapters, you are already aware that when using guizero you need to ensure that image files are stored in the same folder that you have saved the program file in. To do this,

1. Open the folder where you are saving your GUI projects.
2. Create a new folder and name it **MindReader**.
3. Download the project image file of the witch from Chapter 6 https://github.com/TeCoEd/Twisted_Python_Projects/tree/main/Project_ or use your own image file.
4. Save or copy the image file to the **MindReader** folder.

You are now ready to write the program code and create the Mind Reader game.

1. Load your Python editor.
2. Create a new program file in your editor.
3. From the **File** menu, choose **File** > **New File**.
4. A new window opens. Choose **File** > **Save**. The Python editor prompts you to save the file.
5. Name your Python file **Mind_Reader.py** and choose **Save**.

## Importing the modules

Like the previous GUI projects, this project also begins with the import of all the required modules. Add the code from listing 6.1 into your Python file.

**Listing 6.1 Importing the modules**

```
# Imports

from guizero import App, Box, Picture, PushButton, Text
from random import randint
from time import sleep
```

The first set of modules that are imported, shown on the first line, will be familiar by now; we have used most of them in the previous projects. These modules enable you to create the GUI window, use Box to lay out the widgets, and display an image, buttons, and text.

One line two a new method from the random module, `randint()` is introduced. In chapter 2 we used the random module to select one random joke from a list of jokes. In chapter 5, you used `choice()`to choose a random background color from a list and set it as the background color of the GUI.

In this project we are going to use

```
from random import randint
```

This method from the random module allows Python to generate a random integer. What is an integer? An integer is a whole number. For example, a number such as 65 or 23 or 2. Integers have no decimal place values, so the numbers 65.2, 23.7, 2.89 are not integers.

#### A method

In Python, a method is associated with a module. In this instance, `choice()` and `randint()` are methods associated with the random module. The `sleep()` methods is associated with the time module.

When the GUI is running to make the game create the illusion that the witch or your character is really thinking of a number, we need Python to select a random number and save it without us knowing what number has been picked. This means that since we don't know the number, we can try to guess it.

However, since Python could choose any number in the world, we must set a boundary. This boundary is the lowest and the highest values the random number can be between. The code `randint(1,100)` generates a number between and 1 and 100 including the numbers 1 and 100. We will set this boundary later in the variables section when we use the `randint()`. You can change the boundaries to make the number easier or harder to guess.

On the third line we import `sleep()` from the time module. This module enables us to control timings within the program and add a short delay between the previous and the next line of code running. We will use a small pause between the GUI saying the answer is wrong and prompting the user to guess again. This gives the player enough time to read the message before they respond.

## Creating the variables and choosing a random integer

In this section of the program, we declare the variable that holds the number

of guesses that the player has. When the game starts the player is given five chances to guess the number correctly. To make the game more exciting, the GUI will display that the player has five lives.

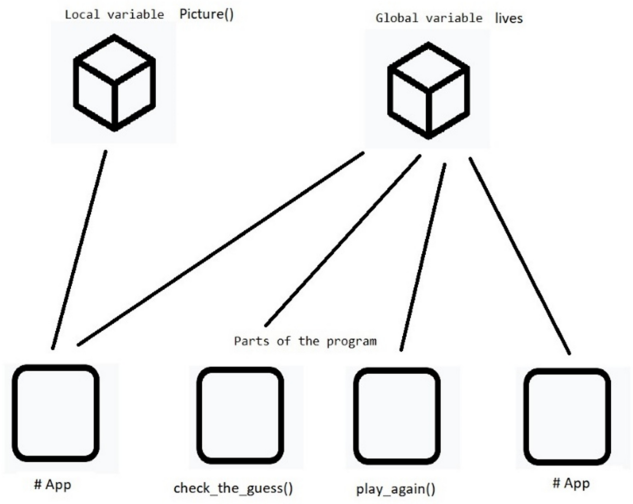
The lives, and the guesses are always the same value, but it is more motivating for a player to say that they have lost a life rather than that they have lost a guess.

Since the number of lives needs to be used by other parts of the program it is declared as a global variable. Global variables hold data which can be accessed by other sections of the program code (figure 6.3). In this program, the variable `lives` is accessed by the three separate areas of the program:

1. The `check_the_guess()` function which removes a life each time the player makes a guess.
2. The `play_again()` function which resets the lives back to 5 and starts a new game.
3. The `App()` function, which displays the number of lives in the GUI.

Local variables can only be accessed by specific parts of the program. The GUI displays an image of the witch. To do this we will create a local variable called `pic =` to hold the image details. This Variable only appears in the `App()` section (as this is the section that is responsible for displaying the image) and is inaccessible by other parts of the program.

**Figure 6.3 Data held in global variables can be accessed by any part of the program code.**




After declaring the variable `global lives`, we use `lives = 5` to set the starting number of lives to five. This variable is also used to keep track of the number of remaining guesses that the player has left and ensures that the game ends when all the player's lives have been used up.

Under the import section of the program code, add the code in listing 6.2.

**Listing 6.2 Creating the variables**

```
# Variables

global lives
lives = 5
```

The first line of code creates the global variable `lives`. This is a location in the computer's memory that stores the number of lives. As a global variable, it ensures that the information held can be accessed by other parts of the program, in particular, the `check_the_guess()` function that we will code in the section, "Running and testing the GUI," that checks whether the player's guess matches the witch's number. Then we set the `lives` variable to a value of 5. These are the five lives that the player starts the game with.

## Creating the main guessing function

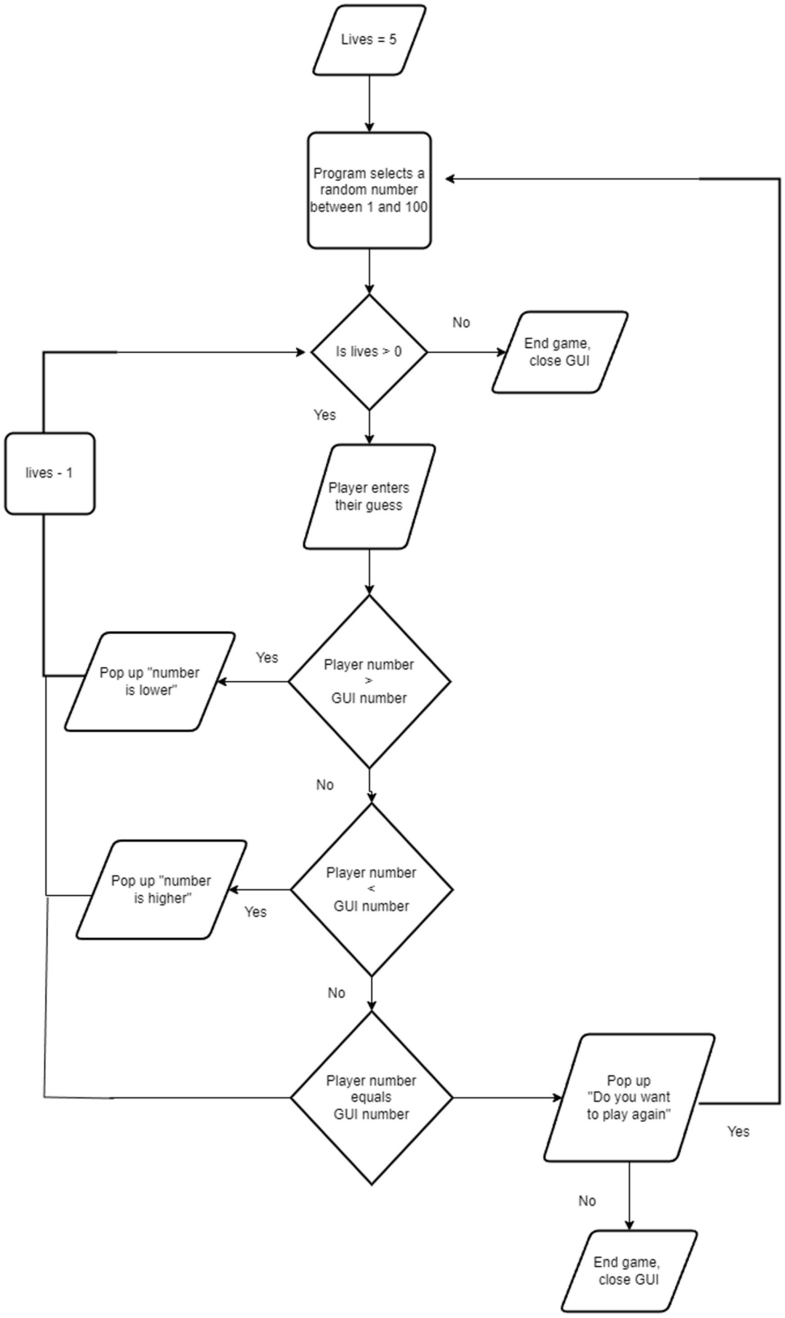
The next step is to create the two functions that the program uses. A function

is a reusable block of code that can be called and used anywhere within the program. Functions save time and make the program more efficient. The first function `check_the_guess()` manages the mechanics of the gameplay: guessing the number, comparing it with the witch's number, responding with a suitable message and so on. This function is called when the player enters a number and presses the OK button on the popup window.

The following happens (figure 6.4):

1. The global lives variable is imported.
2. Since the game has just started the number of lives is set to 5
3. The program selects a random number between 1 and 100, (including 1 and 100).
4. The function checks if the number of lives is greater than 0, (if it was not then it means that the player has used up all their guesses and has no lives left.)
5. The player types in their guess on the keyboard
6. The function compares the player's number with the random number it generated at the beginning of the program.
7. If the guess is greater than the random number, the program informs the player that their number needs to be lower.
8. One of your lives is removed.
9. If the guess is less than the random number, the program informs the player that their number needs to be higher.
10. One life is removed.
11. If the guess and the random number are equal, then the player has correctly guessed the number and a winning message is displayed.
12. The function then calls the `play_again()` function and then player is then asked if they want to play the game again.
13. If they guess incorrectly then the function returns or loops back to the beginning and the process happens again.
14. When the `lives` are not greater than 0, the function states you failed and closes the GUI, the witch does not entertain losers!

**Figure 6.4 Flow diagram of check_the_guess() function**

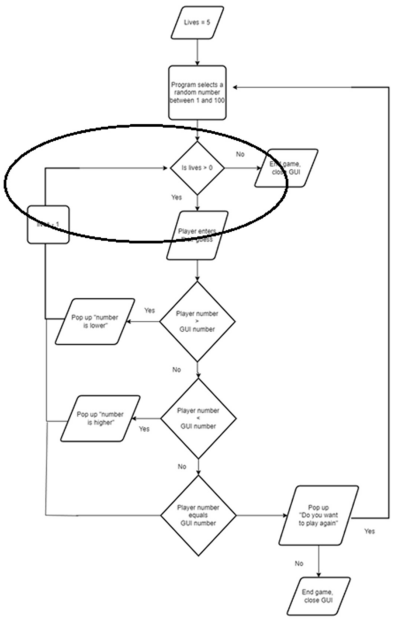



The function makes use of a loop to keep track of the number of remaining lives, figure 6.5. In programming, a loop is the term used to describe the repeating of a section of code. The program returns to the start of the code and repeats the same code again. To do this, the program needs to know when to return to the start of the loop, and when to end the loop, so it checks for a certain condition being met. The three golden rules of loops are knowing:

1. When to start
2. When to stop
3. What to repeat

A fourth rule might be to know the step size. For example, in this program we are using the default increments of 1 in the loop. (One life is lost each time you guess incorrectly) We shall look at the condition later in this section.

**Figure 6.5 The main loop**




**What is a Loop?**

A loop is a sequence of instructions that is continually repeated until a certain condition is met. In this function the sequence continually repeats until the lives = 0 and then the loop breaks.

Loops mean that, we as the programmers, can write quicker and more efficient program code. Let's consider the program we are writing. As part of the game, we want the witch to ask the player to try and guess their number, they get five chances to guess it correctly. One way to code this in the program is to ask the player five times to enter a number,

```
guess = app.question("Guess", "Mortal, enter the number I am thin
```

```
guess = app.question("Guess", "Mortal, enter the number I am thin
guess = app.question("Guess", "Mortal, enter the number I am thin
guess = app.question("Guess", "Mortal, enter the number I am thin
guess = app.question("Guess", "Mortal, enter the number I am thin
```

However, this is inefficient as you have to write more lines of code. Therefore there is an increased chance that you will make an error, the code will be incorrect, and the program will not run. Also, it will take you more time to type out and check 5 lines compared to 1 line, so it is harder for you too.

#### Reasons for using loops.

Not using a loop means that when the GUI runs, Python will have to ask the player to enter their number and then process their response. This could take 4 or more lines of code. The player has 5 chances to guess the number, which means the program could use an extra 20 lines of code. This requires additional memory and processing power from your computer. Meaning that it must work harder. This is not really an issue for 5, 10 or even 1000 lines of extra code and will make little noticeable difference. However, if we scale up the size and consider a bank program that has to check 14 digit account numbers for a million customers, then we are into more than 14,000,000 lines of additional code!

The solution is therefore, to write the lines of code once and then use a loop to repeat the line 14,000,000 times. In our program we use the loop to repeat the line of code five times.

Previously I mentioned that the loop needs to know when to start and when to stop. We control this using a conditional. A condition is like a check, the program checks if the condition has been met, if it has been met, then the program breaks the loop and moves on to the next part of the program. While the condition is not met, the loop continues to repeat.

#### Loops in school

Think about when you were in class at school. At the start of each lesson, the class register is taken. The teacher calls out the name of the student and waits

for a response, then they move onto the next student in the register and call out their name. This process loops until the last name has been called out.

In this example, the condition is “have all the names been read out?” The teacher keeps looping and calling out all the names until the last name has been called out.

When the Mind Reader program runs, the player is given five lives which we can use as part of the condition to check when to break the loop. The program runs the block of code, in our case the `check_the_guess()`function repeatedly until a given condition is satisfied. Whilst the player has lives left, the program carries on asking them what their next guess is.

The program loop is asking the question, (checking the condition) is the number of lives greater than something? Now we need to think about what the lives are greater than. Let’s consider this, when does the game end? When the player has used up all their lives.

Therefore, we can say that if the player has no lives left, they have zero lives. In Math the *more than* comparison is called ‘greater than’. Combining this altogether we can create the condition, *while lives is greater than 0*, then replace the greater than with the mathematical symbol > and we get `while lives > 0.`

Type up the code in listing 6.3.

**Listing 6.3 The check_the_guess function**

```
# Functions

def check_the_guess(): #A
    global lives #B
    number = randint(1,100) #C

    while lives > 0: #D

        [CA]guess = app.question("Guess", "Mortal, enter the numb
       lives_left_text.value = lives – 1 #F
```

One of the challenges when writing longer functions is ensuring that the

indentation levels are correct and match. This refers to the alignment of the lines of code and is used so that Python knows which lines of code are part of the function and which lines are not. Indenting refers to the use of white space before the text, which means the text starts a little distance away from the left-hand margin. Check your indentation to ensure that it is correct.

We then declare the variable `number,` which stores the random integer that the program will generate. The random integer is generated using the `randint()` function and we now set the boundaries for the numbers.

To set the lower and upper values we include them, expressed as numbers, within the brackets, like this, `randint(1, 100).` This sets the lowest integer that can be selected to 1 and the highest to 100. The player is attempting to guess any number between one and one hundred. If you want to, you can change the boundaries values to make the game easier or harder. You may also want to adjust the number of lives that the player has. This will give them at least some chance of guessing the correct number before their lives run out!

Next, we add the lines of code that we want the loop to repeat, while the player has more lives than zero. The first repeated line of code is the question asking the player to enter their guess, their number is then stored into the `guess` variable. Then, because the player has used one of their chances the program needs to remove a life and update the lives left from 5 to 4. This is done using the line:

```
lives_left_text.value = lives – 1
```

where `.value` updates the text label displaying the remaining lives and, `= lives – 1` , subtracts 1 life from the remaining lives.

After the player enters their number, then the program needs to check their guess against the witch's number, and then tell the player if their number is either too high, too low or correct.

To perform this comparison, the program uses an If statement to check if certain conditions have been met, (it is often helpful to think of conditions as questions). This is similar to the loop, where a condition is checked to know

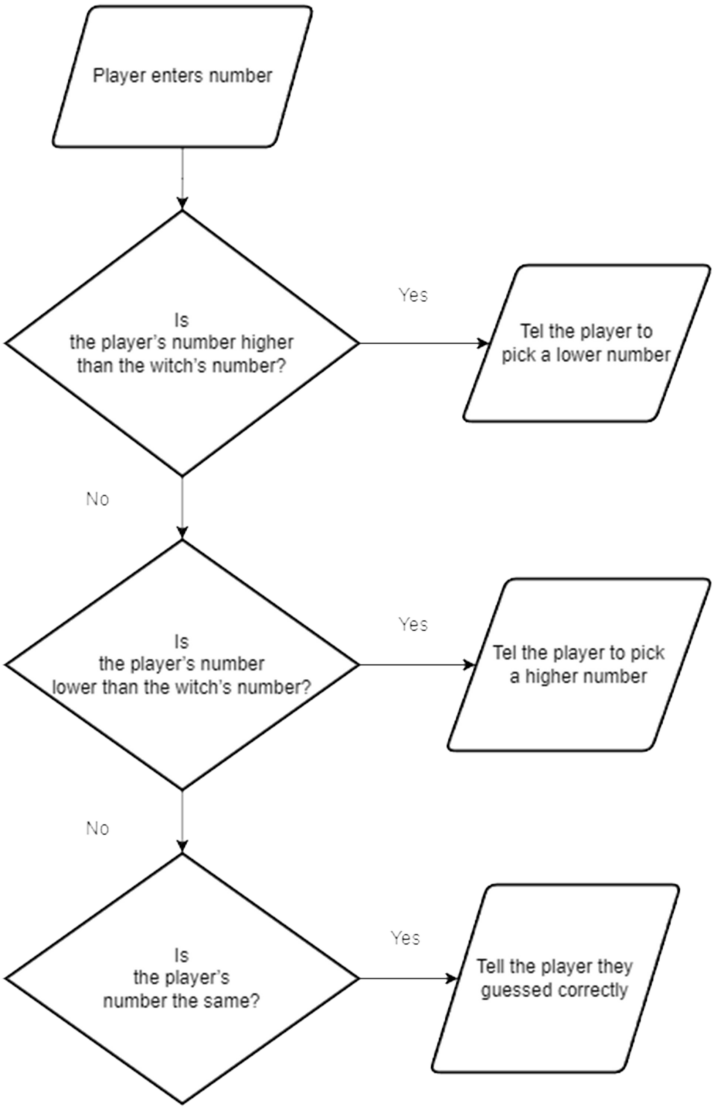
when to break the loop. Figure 6.6. Here we are using the condition to enable the program to decide which action to take after it compares the two numbers. Remember that we have to code the action for the program to take when the condition is met or not met.

In this program, we use the IF Statement to check the follow conditions, table 6.1.

**Table 6.1 Conditions and outcomes**

| Condition / Possible answer | Question to ask / Action to take |
|---|---|
| Condition 1 | Is the player's number higher than the witch's number? |
| Yes | Tell the player to pick a lower number |
| No | Check condition number 2 |
| Condition 2 | Is the player's number lower than the witch's number? |
| Yes | Tell the player to pick a higher number |
| No | Check condition number 3 |
| Condition 3 | Is the player's number the same as the witch's number? (Player has won) |
| Yes | Tell the player that they guessed correctly and won |

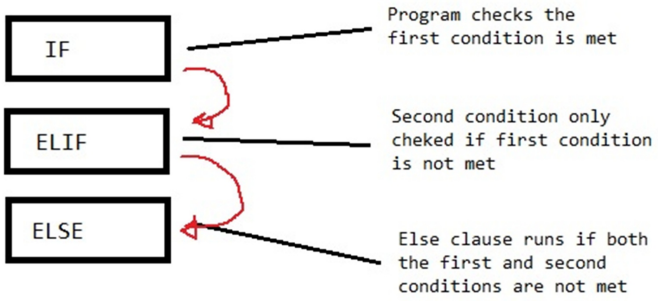
**Figure 6.6 Flowchart, outcomes after you make a guess.**




**The running order of CONDITIONS!**

Each time an if statement runs, it always starts at the beginning and checks the first condition. The second condition only runs if the first condition is false (or not met). Then, it checks the second line condition, if this is false (or not met), only then is the third condition checked and so on. Figure 6.7.

**Figure 6.7 The order the conditions are checked.**




When there is more than one condition to check, we use the word `if`, to check the first condition and then the word `elif` (`else if`) to check the next conditions. The if statement ends with `else`, (known as an else clause) which is used to tell the program what to do if none of the conditions are met. Let's revisit the three conditions, table 6.2, and replace the first word with the relevant word from the if statement. We now have:

**Table 6.2 Updated conditions**

| Condition | What to check for |
|---|---|
| 1 | if the player's number is higher than the witch's number? |
| 2 | elif the player's number is lower than the witch's number? |
| 3 | elif the player's number is the same as the witch's number? (Player has won) |

However, using logic we can know that if the player's guess is not higher **and** it is also not lower than the witch's number, then, the player's number must be the same as the witch's number. Therefore, this means that both numbers are equal, and the player has guessed correctly. So, we do not need to use `elif` for the third condition, we can replace it with an else clause. We also replace the math related terms with the equivalent symbols, and we get the final conditions of the if statement. Table 6.3.

**Table 6.3 Updated Conditions**

| Number | Condition |
|---|---|
| 1 | if the player's number > than the witch's number? |
| | |

| | |
|---|---|
| 2 | elif the player's number < than the witch's number? |
| 3 | else the player's number = the witch's number? (Player has won) |

Now we combine the conditions with the pop-ups, `app.error()` and `app.warn()` and a message to complete the if statement, which informs the player if their guess is higher or lower than the witch's number. The other outcome is that the player correctly guesses the number, so we use the `app.info()`and include a message to inform them that they have won the game.

Add the code from listing 6.4 to the `check_the_guess()` function ensuring that the indention levels are correct.

**Listing 6.4 Adding the IF statement.**

```
if int(guess) > number:    #A
            app.error("Lower", "Poor effort, the number is lower"
            lives = lives – 1 #C
        elif int(guess) < number: #D
            app.warn("Higher", "No fool, try a higher number") #E
            lives = lives – 1 #F
        else: #G
            app.info("Well done", "Mortal, you win!") #H
            play_again() #I
```

The first line of the if statement, `if int(guess) > number` is checking that the player's guess is greater than the random number. As an example refer to table 6.4, let's say that the player's guess is 34 and the witch's number is 17. Line one of the code checks if 34 is greater than 17. It is, so the next line of code runs displaying the error Pop-up and the message, *Poor effort, the number is lower*. One of your lives is deducted, you now have four left.

As a smart player, on your next guess you will want to enter a lower number than 34. You enter the number 12. This time the first line of code runs and checks if 12 is greater than 17. It is not, so the program jumps to the `eilf` and checks if the number is less than 17. It is, so the next line of code runs displaying a popup and the message, *No fool, try a higher number***.** Again, one of your lives is deducted. You now have three lives left.

**Table 6.4 An example of the guessed number and then outputs**

| Guessed number | Outcome message | Lives left |
|---|---|---|
| 34 | lower | 4 |
| 12 | higher | 3 |
| 23 | lower | 2 |
| 13 | higher | 1 |
| 16 | Game is over | 0 |

Considering your next guess, you know that the number is greater than 12 and less than 34. You decide to go for the middle value and guess 23. This time the first line of code runs and checks if 23 is greater than 17. It is, so the next line of code runs, displaying the error popup and the message, *Poor effort, the number is lower*. Again, one of your lives is deducted, you now have two left.

Your next guess is 13 and the program states, *No, fool, try a higher number***.** You now have only one life left. It is your final guess. You guess 16, the If statement checks if 16 is greater than and then less than 17, and then removes your last life, you have zero lives left.

At this point the program returns to the start of the loop, and it runs the loop again. Remember the loop only runs `while lives > 0`, and guess what, you have zero lives and your lives are no longer greater than zero, so the loop breaks, and the game is over. The if statement does not run, and the program jumps to the code under the loop code. The witch tells you that you have lost!

Complete the `check_the_guess()` function and type up the code from listing 6.5. Be sure to watch the indentation levels ensuring that all the code aligns correctly. Any issues or uncertainties then check out listing 6.6.

**Listing 6.5 Running out of lives**

```
    [CA]app.error("Lost", "You have failed my pretty, the number
    sleep(0.5)
    app.destroy()
```

In this concluding section of the function, we add the code to display a

message in a popup window that informs the player that they have lost the game. It would be really unfair if we didn't also share with the player the number that they have been trying to guess. Let's put them out of their misery! To do this, we combined the `number` variable with the message, `You have failed my pretty`, to create the complete message:

```
You have failed my pretty, the number was " + str(number)
```

Before we do this, we convert the number from an integer into a string using the code str(number) using a technique called casting. In Python, the term casting refers to the process of converting the data stored in a variable from one type of data to another. For example, changing the number 45 to text, which removes the value of the number and is simply the symbols 4 and 5. The number must be changed to a string otherwise Python will try and add `You have failed my pretty, the number was` and the 45 together. This will result in an error since you cannot add text and numbers together.

On the next line of code, we add a short pause to give the player time to read the message before the program code closes the GUI. The player has lost so if they want to play again, they have to restart the GUI. Listing 6.6 shows the complete function, ensure that your indentation is correct and matches before you move onto the next section.

**Listing 6.6 Final first function listing**

```
# Functions

def check_the_guess():
    global lives
    number = randint(1,100)


    while lives > 0:

        [CA]guess = app.question("Guess", "Mortal, enter the numb

        lives_left_text.value = lives - 1

        if int(guess) > number:
            app.error("Lower", "Poor effort, the number is lower"
            lives = lives - 1
```

```
        elif int(guess) < number:
            app.warn("Higher", "No fool, try a higher number")
            lives = lives - 1
        else:
            app.info("Well done", "Mortal, you win!")
            play_again()

    [CA]app.error("Lost", "You have failed my pretty, the number
    sleep(0.5)
    app.destroy()
```

## Creating the ‘play again’ function

The second function manages the process of playing the game again. This involves the following steps.

1. The player guesses the number correctly.
2. The witch asks the player if they want to play again.
3. If the player selects yes (True) then the `check_the_guess()` function runs, a new random number is generated. The lives are reset to 5 and the player has 5 guesses.
4. The player decides not the play the game by selecting the No (False) option, then a message from the witch is displayed and the GUI closes.

This function also uses IF statements again to respond to the choice that the player makes when asked if they want to play the game again. Add the code from listing 6.7.

**Listing 6.7 The play_again function**

```
def play_again(): #A
    global lives #B
    play_again = app.yesno("Witch", "Do you want to play again mo
    if play_again == True: #D
        lives_left_text.value = 5 #E
        lives = 5 #F
        app.info("Witch", "Excellent choice!") #G
        check_the_guess() #H
```

**About** ==

Remember that in Python, = means *has a value of*. It is used with variables and lists, and to assign data, or to declare variables and functions; for example:

```
play_again = app.yesno() or PI = 3.141592653589
```

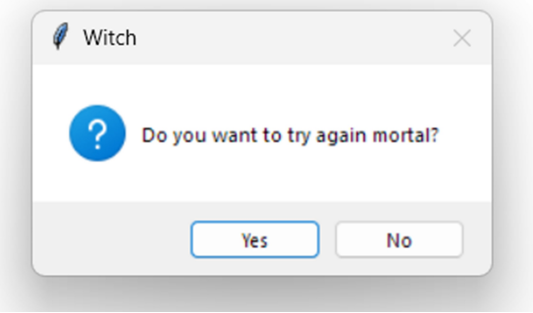
To check if something is equal, Python uses two equal signs == for example:

```
if play_again == True: 20 == 10 + 5 + 5
```

Two equals signs ==, means equivalent

Create the new the new function called `play_again()`on the first line, then on the next line import the global variable lives which means that all parts of the program, mostly the functions can access the current value of `lives`. Then we declare a new variable called `play_again` and a pop-up window `app.yesno()` that asks the player, *Do you want to try again mortal*? (figure 6.8.)

**Figure 6.8 The Yes/No pop-up window.**

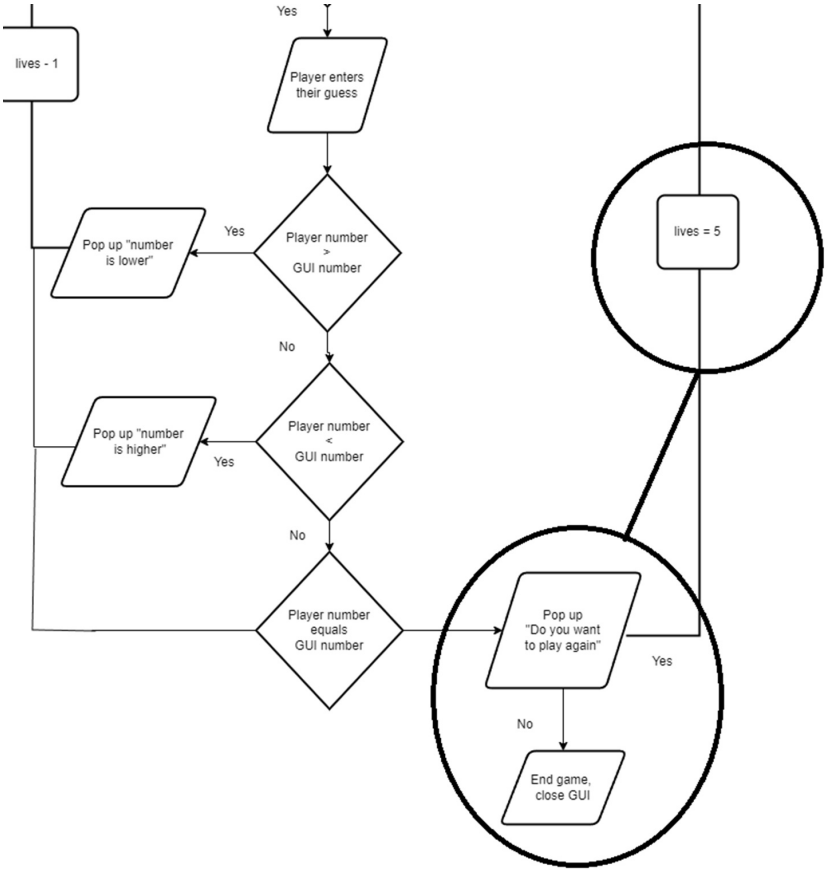



**True or False**

In the program code you will have seen that True and False are used to mean Yes and No. Computers do not understand the meaning of Yes or No, or even True or False, but they do understand numbers. This means the computer can respond to a player's choice and store their response.

Next, we create an If statement to respond to the player's choice, let's say that they select the Yes button. This sets the `play_again` variable to True then on the next line the program checks if `play_again == True`.

**Figure 6.9 Flowchart show how the program asks the player if they want to play again**




Since the condition has been met, the player clicked Yes, the next four lines of code run. The `lives_left` variable is reset to 5 so that the GUI displays that the player now has 5 lives. The code `lives = 5` resets the number of lives back to 5 to ensure that as we are starting a new game the player starts with 5 chances again. The witch displays a message via a popup and then the `check_the_guess()` function is called, and the game begins again.

Now let's consider what happens if the player decides they do not want to play again. The popup window appears, and the player selects No. The variable `play_again` stores the data 'False', in this context False means No. The if statement runs and checks if `play_again` is True, it is not so the if statement skips the indented lines of code and jumps to the elif condition, listing 6.8. The lines of code related to the elif then run, a popup is displayed with a message from the witch, there a short delay to give time for the player to read the message and then lastly, the GUI closes.

In this if statement there is also an else clause, this is not needed as there are only two options, Yes and No. However, it is good practice to end the statement with an else clause. We don't want anything to run on else and we use `pass` which does what it says, it passes and jumps to the next line of code.

Add the code from listing 6.8 to complete the function. Remember to keep an eye on accurate indentation.

**Listing 6.8 The player choses not to play again.**

```
elif play_again == False:
    app.info("Witch", "A wise choice...")
    sleep(1)
    app.destroy()

else:
    pass
```

**Remember**

The person writing the code must write both the condition and the responses. The program only knows how to respond because we have told it how to.

## Building the main app

This last section of the chapter covers how to build the main GUI app. If you have already worked through some of the other chapters, then you will be familiar with the code and what it does. We begin by naming the app and setting the size of the GUI window.

Then use `Text()` to prompt the player what to do, *"Can you guess my secret number?"* and set the font size and color of this text. Lastly use `Picture()` to display the image of the witch or witch-ever(!) character you have chosen to challenge you in the game.

The middle block of code (which begins `lives_heading = Text()`) manages the text that informs the player about the number of lives that they have left.

This consists of using the `Text()` to display the text, “Lives left” and then pulling in the data currently stored in the `lives` variable. The font size is set to 42. Since the number of lives the player has left is important, the font size used is large!

The final section of code creates the button which when pressed by the player calls the `check_the_guess()` function and begins the game. Add the final section of code from listing 6.9.

**Listing 6.9 Setting up the main app**

```
app = App("Guess the Number", height = 400, width = 350)
title_text = Text(app, "Can you guess my secret number?")
title_text.text_size = 16
title_text.font = "Impact"
pic = Picture(app, image="witch.gif")

lives_heading = Text(app, "Lives Left")
lives_left_text = Text(app, lives)
lives_left_text.text_size = 42

begin_button = PushButton(app, command=check_the_guess, width=20,
begin_button.text_size = 10

app.display()
```

## Running and Testing the GUI

Apart from the if statement in the functions, this program builds on the things we have used in previous chapters and includes many of the widgets, features and lines of code from the previous chapters. Any errors that occur will probably be syntax errors. This is caused where the code has been typed incorrectly and the program does not understand how to proceed, and so the program stops.

To run the GUI, press **F5** on the keyboard, you may be prompted to save the program file. Choose **OK**, the program will save, and then the program will run, and the Mind Reader GUI will load. Can you guess the witch’s number?

If you encounter any issues, or the program does not run as expected, check

for these common errors:

1. Ensure the files are stored in the correct folders.
2. Check that the indentation is at the correct level for all functions and if statements.
3. Remember to use the colon symbol : at the end of each conditional.
4. The lines of code that follow underneath the : must be indented.
5. Are the equality operators > and < the correct way around.
6. Change the `randint(1,100)` to `randint(1,2)`. Then you can test the program with just the values 1 and 2 and identify the error.
7. Check out the final code listing 6.10 and compare it with your program code.

**Listing 6.10 Final code**

```
# Imports

from guizero import App, Box, Picture, PushButton, Text
from random import randint
from time import sleep

# Variables

global lives
lives = 5

# Functions

def check_the_guess():
    global lives
    number = randint(1,10)

    while lives > 0:

        [CA]guess = app.question("Guess", "Mortal, enter the numb

        lives_left_text.value = lives - 1

        if int(guess) > number:
            app.error("Lower", "Poor effort, the number is lower"
            lives = lives - 1
        elif int(guess) < number:
            app.warn("Higher", "No fool, try a higher number")
            lives = lives - 1
```

```
        else:
            app.info("Well done", "Mortal, you win!")
            play_again()

    [CA]app.error("Lost", "You have failed my pretty, the number
    sleep(0.5)
    app.destroy()

def play_again():
    global lives
    play_again = app.yesno("Witch", "Do you want to try again mor
    if play_again == True:
        lives_left_text.value = 5
        lives = 5
        app.info("Witch", "Excellent choice!")
        check_the_guess()

    elif play_again == False:
        app.info("Witch", "A wise choice...")
        sleep(1)
        app.destroy()

    else:
        pass

# App

app = App("Guess the Number", height = 400, width = 350)
title_text = Text(app, "Can you guess my secret number?")
title_text.text_size = 16
title_text.font = "Impact"

pic = Picture(app, image="witch.gif")

lives_heading = Text(app, "Lives Left")
lives_left_text = Text(app, lives)
lives_left_text.text_size = 42

[CA[begin_button = PushButton(app, command=check_the_guess, width
begin_button.text_size = 10

app.display()
```

# Other things to try.

In this chapter you created a Mind Reader game where a witch, or your own

character, thinks of a random number between 1 and 100. You are then granted five lives which you trade in each one for a guess at the number. If you guess correctly, then you win, if not, then the game ends. Remember that you can change the image, the messages, to personalise your own version of the game. How about a magical cat, a talking pig, or a family member!

Here are some other features that you might want to try:

- Extend the random number range
- Changing the number of lives
- Adding sound

## Extend the random number

The first other feature to try adding is to change the lowest, highest, or both values of the boundary that the program can generate a random number from. Currently this is set to, `number = randint(1,100)`

Why not try 1 to 50? Replace the code with,

```
number = randint(1,50).
```

This will make the game easier, or you could increase the values and make it more of a challenge; for example, `number = randint(1,1000)`. You will need to remember to increase or decrease the number of lives that the player is given, as a larger range of numbers will require more guesses to correctly guess the correct number.

## Changing the number of lives

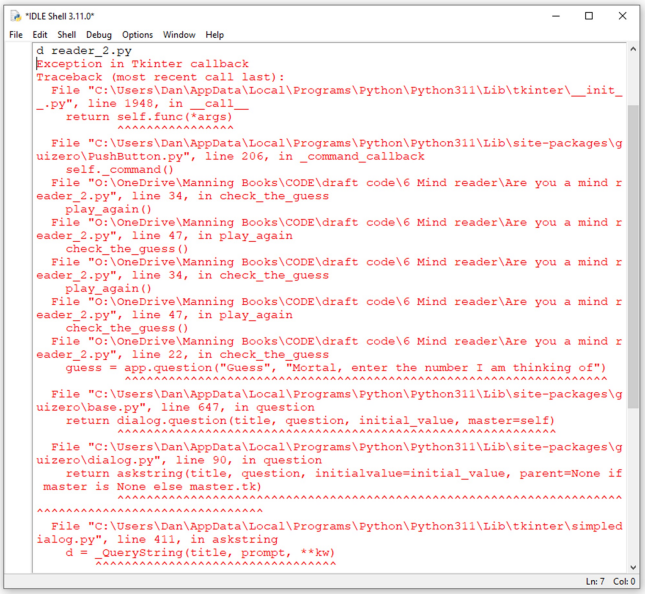
In the program we used a while loop to check if the number of lives that a player has remaining is greater than zero. Using a loop has the advantage that we can set the initial value for the number of lives to a different value, and the program still runs correctly. For example, if we set the lives to 10, then the loop does all the work and just checks ten times if the lives are greater than zero. If we set the lives to 20, then the loop will check twenty times. To change the lives, make the following two edits,

1. Locate the **lives** variable in the # Variables section of the program, `lives = 5,` then change the value. If you want to make the game difficult then reduce the lives to three by changing the code to `lives = 3.`
2. In the `play_again():` function, locate the line of code that displays the number of lives in the GUI so that the player knows how many they have left: `lives_left_text.value = 5`. Set this number to the same number as the lives, for example, `lives_left_text.value = 3.`
3. You will also need to change the `lives = 5` right below the `lives_left_text.value = 3`, to match. In this example `lives = 3`. This ensures that you always start a new game with 3 lives / guess.

## Cleanly exiting the App and program

You may have noticed that when you win the game and are asked “Do you want to play again Mortal?”, if you select the No option, then the GUI closes but an error message is displayed in the IDE. Figure 6.10.

**Figure 6.10 Python error caused when you select not the play the game again**




This is because the App is destroyed (closes) but the program is still running. To resolve this issue, we need to exit the program. This is a technique that can be applied to other GUI program where you encounter this error.

The solution uses the code, `sys.exit()` which is a built in function that enables Python to end the execution of the program. Follow the steps below and make the following edits to your program:

1. First, add systems module in the `# Imports` section of the program using the code `import sys`.
2. In the `play_again()` function, locate the last line of code, `app.destroy()`. Underneath this add the line of code `sys.exit()`. This will stop the program from running and no error will be produced.
3. Save your program (F5) and test it.

## Adding sound

One more thing to try is to add sound. How about if the witch cackles every time you guess incorrectly, or insults you if you lose the game? If you have previously completed chapters two or five, then you will be familiar with how this works. We use the winsound audio software installed with Windows and import the module into Python.

Guizero plays .wav sound files, so you can either download the file from the project page, https://github.com/TeCoEd/Twisted_Python_Projects/tree/main/Project_Code or you can use your own audio file.

Now follow these steps.

1. Copy your audio file into the **MindReader** folder where the Mind_Reader.py file is saved.
2. In the `# Imports` section of the program import winsound by adding the code, `import winsound`
3. Let's say that we want to add a witches laugh each time that the player makes an incorrect guess. In the `check_the_guess()` function, find the if statement shown below,

```
if int(guess) > number:
    app.error("Lower", "Poor effort, the number is lower"
    lives = lives - 1
elif int(guess) < number:
    app.warn("Higher", "No fool, try a higher number")
```

```
            lives = lives – 1
```

4. Insert the following line of code,

```
winsound.PlaySound("MindReader/laugh.wav", winsound.SND_ASYNC)
```

   after the line that displays the messages, shown in bold below.

```
    if int(guess) > number:
        app.error("Lower", "Poor effort, the number is lower"
        winsound.PlaySound("MindReader/laugh.wav", winsound.S
        lives = lives - 1
    elif int(guess) < number:
        app.warn("Higher", "No fool, try a higher number")
        winsound.PlaySound("MindReader/laugh.wav", winsound.S
        lives = lives – 1
```

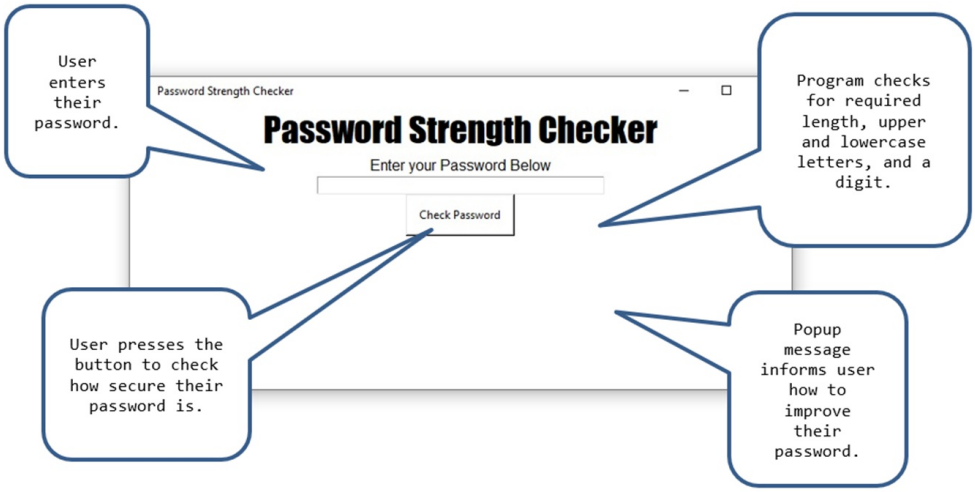
5. Save the program (press F5) and try out the new features. Don't forget to turn your speakers on and ensure the volume is up.

# Statements of fact

- A loop is a section of the program code that repeats until a condition is met.
- `While` loops repeat the section of code while the condition is being met.
- Inequalities refer to a value being greater than `>`, less than `<`, or equal to `==`.
- In Python, two equal signs `==`, means equal to, or equivalent to. A single equals sign = means assignment.
- A conditional statement checks if a condition has been met, and then code is executed in response to the condition being met.
- If statements are used to check and respond when certain conditions are met.
- Selection is where the program selects a particular action or outcome based on a condition.
- `If` is used to check the first condition.
- The `elif` statement only runs when the `if` condition is not met.
- `elif` is used to if there are one or more conditions to check.
- An else clause, `else`, is used to respond when none of the conditions are

met.

# 7 Is your password secure?




## This chapter covers

- Checking if an input matches a value
- Creating and using for loops
- Using `elif not` to check and respond to a condition
- Using `any()` to return an outcome based on any individual letter in a password meeting a condition

Did you know that it takes a supercomputer less than one second to crack a seven-character password that contains only lowercase letters! In contrast, it takes nearly 200 years to crack a 12-character password made up of lower- and uppercase letters. However, most systems limit the number of times that you can enter an incorrect password before the system locks you out. This means that even though it takes less than one second to crack a weak password, on the third attempt the system would lock out the user.

Since passwords are used everywhere for protecting data and keeping information safe and secret, it is essential that your passwords are secure.

If you want a secure password that is easy to remember but hard to guess,

then the password should meet at least the following criteria:

1. At least eight characters in length. (Special characters can be included for more security and we will explore this later on)
2. Contains at least one uppercase letter.
3. Contains at least one lowercase letter.
4. Contains at least one number (known as a digit).

For example, jelly22fi$h meets all the criteria and is secure. However, you still need to use common sense when thinking of a password. For example, consider if my password was `DanAldred1`. It meets all the criteria of a secure password, but it is easily guessed as it contains my name. Ensure that your password is random enough, you can do this by replacing the letters with numbers or symbols. So `DanAldred1` becomes `D@nA1dr3d1`, making the password a lot harder to guess.

## WHAT YOU WILL BUILD

In this chapter we build our first and only sensible program, a password checker. Passwords are so important that it would be foolish to make a prank version of a password checker.

Instead, we will build a GUI that allows a user to enter in their password, then the program checks that the password is a suitable length, and contains at least one uppercase letter, one lowercase letter, and at least one number.

| | Skill |
|---|---|
| Python | Checking strings, checking a single letter, not if statements, for loop, any()  not |
| Guizero | Enabling and disabling buttons |
| | For loop |

# Creating the project

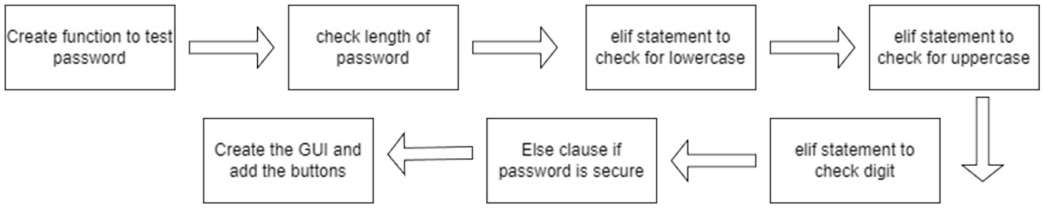
There are no files to download for this project, so you can start straight away. We'll be using two new Python features, `any()` and `not`. The `any()` function looks for any instance that meets a condition. For example, imagine a line of 20 people and we are looking for anyone with black hair. The `not` feature checks if something is not there, for example, not anyone with black hair, is checking that no one in the line has black hair.

The GUI also uses several string functions that check the length of the string and the case of the letters. The main stages of the project are:

1. Start a new Python file and import the required modules.
2. Create a function that checks the strength of the password.
3. Code an `if` statement to check if the length is suitable.
4. Use an `elif` clause to check the password contains a lowercase letter.
5. Use an `elif` clause to check the password contains an uppercase letter.
6. Use an `elif` clause to check the password contains a number.

7. Use the `else` to respond if the password is secure.
8. Create the main GUI window to hold the title, text input, and buttons.

**Figure 7.1 Flow diagram of the secure password project**




In this chapter you will come across several terms that are used in everyday life to mean the same thing (table 7.1). This is confusing! Therefore, this chapter uses the following definitions:

- character: any single element in the password (a letter, digit, or symbol)
- letter: a character from the alphabet (a b c d e f g h i j k l m n o p q r s t u v w x y z)
- digit: a single number (0 1 2 3 4 5 6 7 8 9)
- symbol: sometimes called a special character (! " £ $ % ^ & * ( ) @ ~ # < > ? / \); there are other characters than listed here.

**Table 7.1 Types of characters usually found in passwords.**

| Term | Definition | Example |
|---|---|---|
| character | any single item in a password | a letter, digit, or symbol |
| letter | a character from the alphabet | a b c d e f g h i j k l m n o p q r s t u v w x y z |
| digit | a single number | 0 1 2 3 4 5 6 7 8 9 |
| symbol | sometimes called a special character | ! “ £ $ % ^ & * ( ) @ ~ # < > ? / \ |

## Creating a new file

In the same way that we have started the previous chapter projects, we begin by creating a new Python file. To do this:

1. Open the folder where you are saving your GUI projects.
2. Create a new folder and name it **PasswordChecker**.
3. Open your Python editor and create a new program file in your editor.
4. From the **File** menu, choose **File** > **New File**. A new window opens.
5. Choose **File** > **Save**. The Python editor prompts you to save the file.
6. Name your Python file **Password_Checker.py** and choose **Save**.

## Importing the modules

Let's start the program by importing the required modules. If you have been working through the other chapters, these modules will be familiar. Add the code from listing 7.1 into your Python file.

**Listing 7.1 Importing the modules**

```
# Imports

from guizero import App, info, Text, TextBox, PushButton
```

## Creating the variables

In this program we use a variable named `password` to hold the password that the user types in, for example, `DanA1dr3d1` . The user uses their keyboard to enter this password into a `TextBox()` in the GUI. The `TextBox()` widget displays a box that the user can type into and enter their password (figure 7.2).

Like some of the previous projects, the variable that holds the instruction about the `TextBox()` is coded in the `#App section.` This is to ensure that the `TextBox()` is created and displayed when the app runs at the start of the program.

**Figure 7.2 Using the TextBox() to enter a password.**


Therefore, for this version of the program we do not need to create any variables in the `# Variables` section. You can still add the comment to keep the program structure consistent and neat. Under the import lines of code, add the code in listing 7.2.

**Listing 7.2 No variables this time**

```
# Variables
```

## Creating the password check function

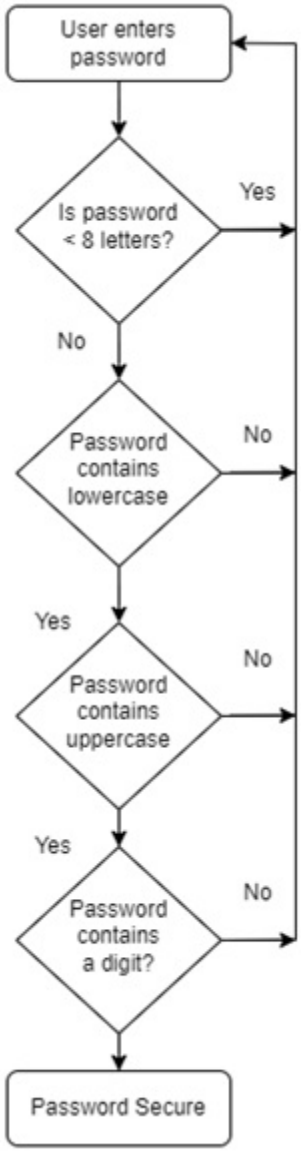
This project has one main function that uses conditionals to check that the entered password meets certain criteria. Table 7.2

**Table 7.2 The security of a password**

| Password | Length | Lowercase | Uppercase | Digit |
|---|---|---|---|---|
| dan | x | R | x | X |
| ALDRED | x | x | R | X |
| A1dr3d1 | x | R | R | R |
| DanA1dr3d1 | R | R | R | R |

The conditionals in the code are the criteria features that make a strong secure password: a certain length of characters (8 or more) a lowercase letter, an uppercase letter and a digit. If the password meets the first criteria, then the next condition is checked (figure 7.3).

**Figure 7.3 Flowchart of conditionals that are checked and the outcomes.**



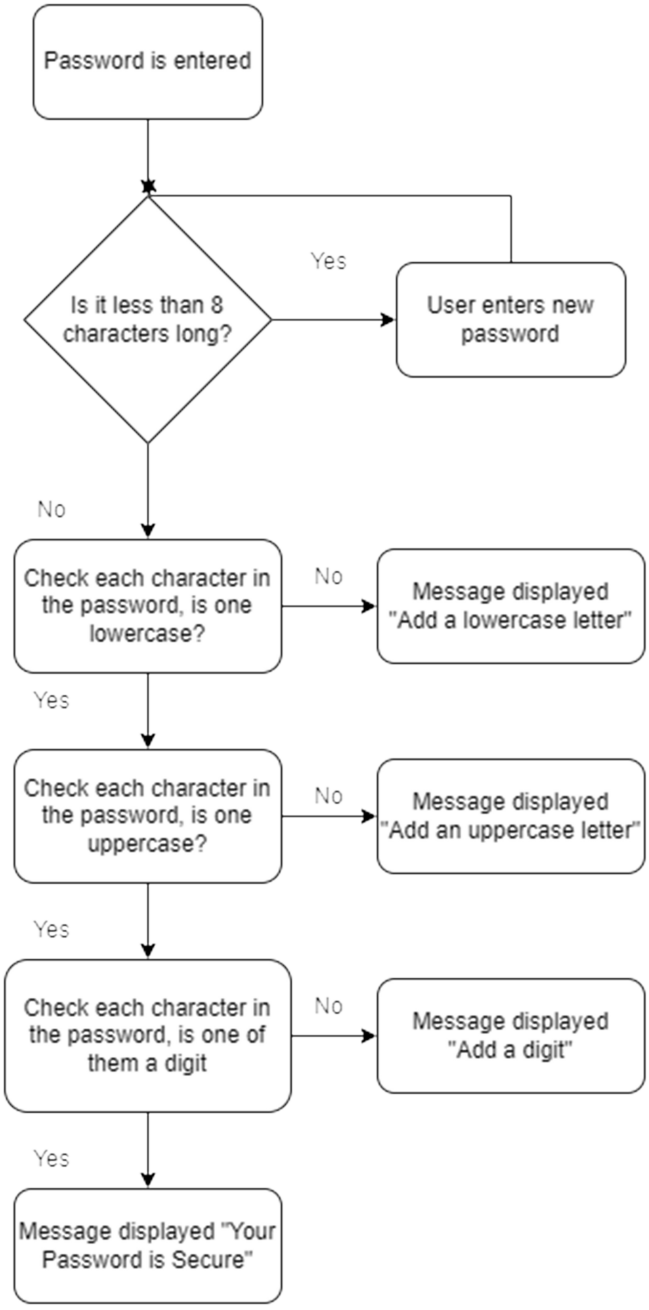

If the password does not meet the first criteria, then the user must edit their password, add the additional secure feature, and then resubmit it for checking. This process creates the illusion of the program looping. However, what in fact is happening is that each time you press the `Check Password` button (which we will create later), the conditions are checked. This happens until all the criteria are met.

The function is called each time the player presses the `Check Password` button, and the following happens:

1. The length of the password is checked, is it less than 8 characters long?

2. If it is less than 8, a message is displayed informing the user.
3. The user re-enters their amended password.
4. Each character in the password is checked for at least one lowercase letter.
5. If the password contains no lowercase letter, then a message is displayed, and the user adjusts their password.
6. Each character in the password is checked for at least one uppercase letter.
7. If no uppercase letter is found, then a message is displayed, and the user adjusts their password.
8. Each character in the password is checked for at least one digit.
9. If no digit is found, then a message is displayed, and the user adjusts their password.
10. When all the conditions are met, a message informs the user that their password is secure. Figure 7.4

**Figure 7.4 Checking each condition and the outcome.**




To create the function, add the code from listing 7.3 to your program's `# Function` section.

**Listing 7.3 Checking the length of the password**

```
# Functions

def PasswordCheck():
    password = password_textbox.value
```

```
    if (len(password)<8):
        app.error("Length ", "Your password must be at least 8 ch
```

As with all functions, we begin by declaring the function. In this program the function is named `PasswordCheck()` since it is checking the Password. Remember that functions only run when they are called and are named so the program can locate them when needed!

On the next line, underneath the function name, we add the code to instruct the program to transfer the characters that the user enters in the `password_textbox(),` into a variable labeled `password`. (We will create the `password_textbox()` in the `# App` section of the program so this is why it is the first time you have come across, `password_textbox()`)

Transferring the characters in the user's password into a new variable makes the code that checks the password simpler. There is less chance of making syntax (typing) errors if you are using a familiar word such as password.

We write

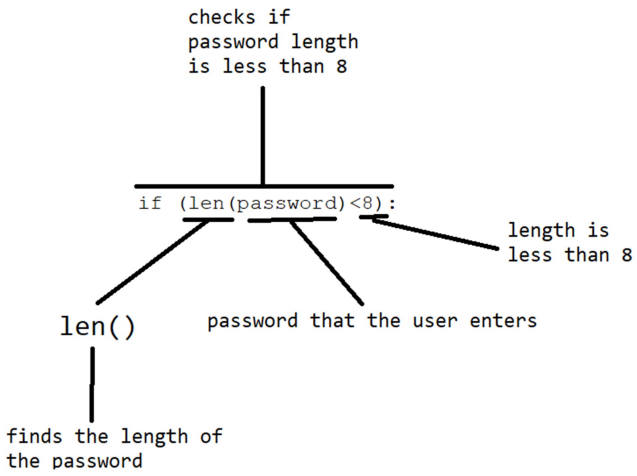
```
if (len(password)<8):
```

rather than

```
if (len(password_textbox.value)<8):
```

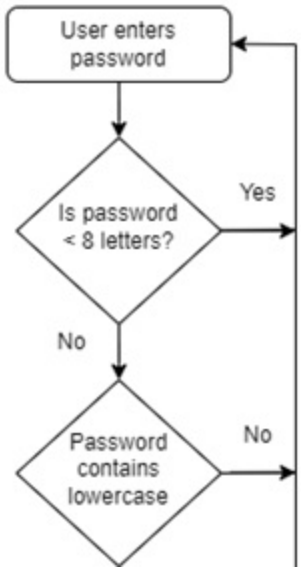
The program has no idea how long the user's password is and therefore it uses the `len()` function to measure and return the length of the password, before combining it with the `< 8` to check that "the password's length is less than 8 characters."

**Figure 7.5 Code to find if the length of the password is less than 8 characters.**




We now have a conditional that checks that the length of the password is at least eight characters or more. Figure 7.5. To do this, the program is actually checking if the password is less than 8 characters long. This uses the math equality < symbol followed by the number 8 (which means “the item on the left is less than the item on the right”). In this case, is the length of the password on the left, less than the number 8 on the right?

**Figure 7.6 Function moves to the next criteria once password is correct length.**




When the function checks the first condition, if the password is less than 8

characters, if it finds that the answer is Yes (True), (the password is less than 8 characters long) then the error pop window displays a suitable error message. Figure 7.6. If the condition is No (False), then this means that the password is 8 or more characters long and meets the first secure password check. The if statement then checks the next set of conditions which are discussed in the section below.

## Checking strings

In this section we look at the next set of conditions that check the password is secure. These conditions are shown with their Python function in table 7.3 and focus on checking each individual character to confirm that at least one character in the password is a lower case letter, an upper case letter, and a digit.

**Table 7.3 Password checks and the Python method used**

| Check | Python method |
|---|---|
| The password contains a lowercase letter | islower() |
| The password contains an uppercase letter | isupper() |
| The password contains a digit | isdigit() |

The program needs to check every individual character and not just the password as a whole; otherwise, it may miss finding a condition.

### Checking for a lowercase letter

Let's consider an example. To check the password contains a lowercase letter, we can use the code, `password.islower()`.

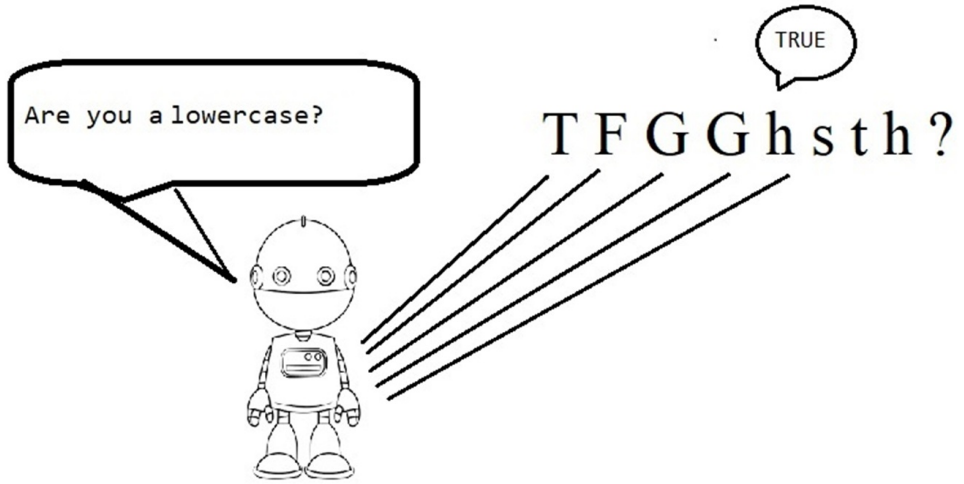
Let's assume that the password is TFGGHSTH and is tested with the `password.islower()` code. The GUI would return `False`, since there are no lowercase letters. If the password was tfgghsth, then the condition would return `True` because the password is lowercase.

What happens if the password is TFGGhsth where the lowercase letters are at the end? The code would not identify the lowercase letters because the

function is checking the password as a whole.

So, the solution to this problem is to check each individual character in the password and return True or False (Yes or No) based on an individual character and not the complete password. Imagine that the program is asking each character if it is lower case, figure 7.7. It does not stop asking until it finds the first lowercase letter.

**Figure 7.7 Condition checks each individual character in the password.**




However, the difficulty with checking every letter is that the password could be any combination and length of characters in the world, and most likely made up of a random combination of upper and lowercase letters. So, we cannot tell the program to check for 10 characters as there may be 14 characters in the password, or 6 characters, or 25, we don't know.

All is not lost. Since we can use `len()` to return the length of the password, then we can use a loop to check each character `islower()` and loop over all the individual characters in the password, the number of times that `len()` says.

For example, if `len()` returns that the password is 17 characters long, then the loop will run 17 times and check if each individual letter is lowercase.

In other chapters we have used a while loop. In chapter 6 we used it to check the number of lives a player had left and continue to repeat the section of

code until all the lives were lost, when the `lives = 0`.

You may recall that using a loop means that we can write more efficient program code. In this program we will use a `for` loop. The `for` loop is used for repeating a section of code over the contents of a string, which is perfect because the user's password is stored as a string!

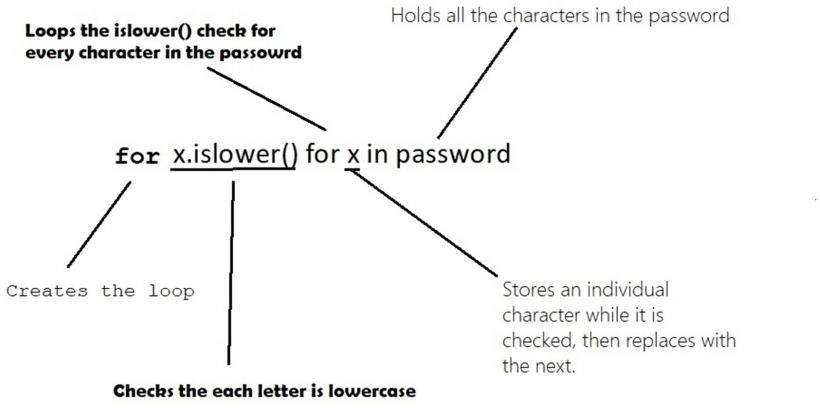
**What is a loop?**

A loop is a sequence of instructions that is continually repeated until a certain condition is met. In this loop, the condition is the length of the password, the check is applied for each of the characters in the password.

## Creating the Loop

To create the loop, we use the code `for` and then tell the program the string to loop over, which is `password.` So, the combined final line of code is `for x in password` where `x` refers to each individual character in the password. Figure 7.8.

**Figure 7.8 Code that creates the loop that checks each character in the password as used in listing 7.4**




`x` is a variable that changes each time the loop repeats by replacing each character from the password whilst it is checked. In programming, the proper term is iteration, the program iterates over each letter in the password and

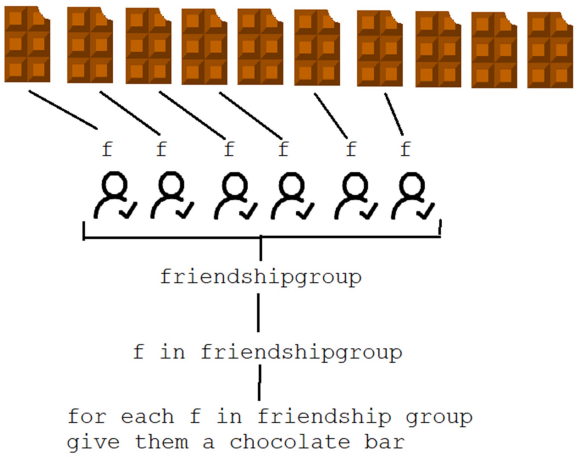
apply the check.

#### x and i in Python

In Python programming when iterating over a word using a line of code such as, `for x in password` the standard is to use a lower case `x` to represent each letter. When the variable contains numbers then the letter `i` is used, `for i in numbers`.

Consider this everyday example: Imagine you have ten chocolate bars and six friends. You will give each friend (f) a chocolate bar, so for each `f` in your friendship group, give them a bar of chocolate. In code this would look like this: `for f in friendshipgroup.` The value of `f` must change each time the loop repeats; otherwise, you would be giving a chocolate bar to the same friend six times. Figure 7.9.

**Figure 7.9 Using a loop to give your friends a chocolate bar each.**




In our program we have nothing as delicious as a bar of chocolate, although the comparison is the same. The same process happens with `for x in password.` The `x` needs to change to check each letter in the password, not just the first letter. We combine the,

```
for x in password
```

with the lowercase letter check, and we have the following line of code:

```
x.islower() for x in password
```

Then, we combine this line of code into the elif statement, and we can check whether each letter in the password is lowercase, regardless of what or how long the password is. Add the code from listing 7.4.

**Listing 7.4 Checking for a lowercase letter**

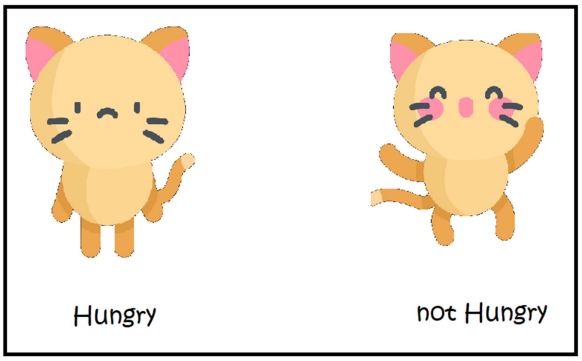
```
elif not any(x.islower() for x in password):
        app.error("Lower Case ", "Your password must include at l
```

You may have noticed that the first line of code uses an `elif` statement and is then followed by the word `not` and the function `any()`.

## Using Not in programs

Let's first discuss `not`. The `not` is known as a *not operator*. It checks if a condition is not met the same way you might say "it is not raining" or "I am not hungry." The not operator uses reverse logic to check if there are no lowercase letters in the password. Figure 7.10.

**Figure 7.10 Using not to find lowercase letters and if you are hungry!**





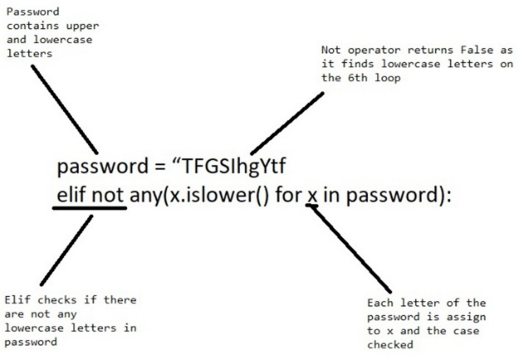
You can think of it as a question: *Is it true that there are not any lowercase*

*letters in the password*? Then the program returns either True or False. If there are no lowercase letters, the program returns True, the for loop stops, and the `app.error()`message is displayed. The user must then adjust and re-enter their password and run the check again.

## The any() function

The `any()` function is a really useful function that checks each character and returns either True or False. We use `any()`to return an outcome based on any individual letter of a whole password meeting or not meeting a certain condition, in this part, any question that is not lowercase. Figure 7.11.

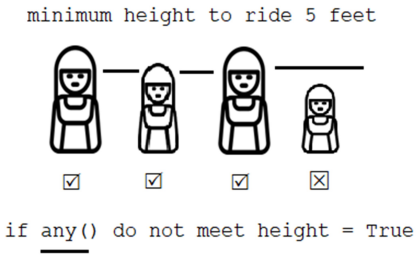
**Figure 7.11 Using elif not any to check a password does not contain lowercase letters.**




Consider an everyday example. You and three friends are at a theme park queueing (lining up) to go on a roller coaster. Each car holds four people and must be full before it leaves. Perfect, as there are four of you! To ride the roller coaster, you must be at least 1.53 meters (5 feet) tall.

The attendant checks your height and you are tall enough. They check that each friend meets the height requirements. When they check your fourth friend, they measure that they are only 1.46m and therefore under the height.

**Figure 7.12 Using any() to check if any of the people meet the height requirement.**




As the check is an `any()`check, it means that if any of the four of you do not meet the requirement, then none of you can ride the rollercoaster (figure 7.12). The attendant asks all four of you to leave the ride! (In real life this would not happen because you would have all checked your heights before you queued for the ride. You would know if one of you was not tall enough.)

Table 7.4 shows the outcomes of checking various versions of the password THRS with different combinations of upper and lower case letters. The code checks each individual letter by saving it into the x variable. After the check, the next letter is moved into x. Once one lowercase letter is found, the checks stop as the password only requires at least one lowercase letter to meet the criteria.

**Table 7.4 Outcomes of checks**

| | elif not any(x.islower() for x in password) | | | | Outcome | Output |
|---|---|---|---|---|---|---|
| Password | x | x | x | x | | |
| THRS | T | H | R | S | True | app.error() |
| THrS | T | H | r | | False | Digit check |
| tHRS | t | | | | False | Digit check |
| thrs | t | | | | False | Digit check |

The program uses the same technique to check for upper and lowercase letters, and digits using `elif` statements. Each time the if statement runs, it always starts at the beginning of the password and checks the first character followed by the next, and then the next, and so it continues until all the characters have been checked. If all the letters fail the check, then the error message is displayed.

### Checking for uppercase letters

In an if statement, the next condition only runs if the first condition is False. Then it checks the second condition, if this is false, only then is the third condition checked and so on. This means that, for example, if the user's password is missing a lowercase letter, the if statement will not check for the uppercase and digit until the user adds a lowercase letter to their password.

When the password is eight or more characters in length and contains at least one lowercase letter, one uppercase letter and a digit, then all of the conditions are met, and the statement jumps to the else clause.

The else displays a popup informing you that your password is secure. Type out the code in listing 7.5, ensuring that the indentation is correctly aligned.

**Listing 7.5 Checking for uppercase and a digit**

```
elif not any(x.islower() for x in password):
        [CA]app.error("Lower Case ", "Your password must include

    elif not any(x.isupper() for x in password): #A
        [CA]app.error("Upper Case “, "Your password must include

    elif not any(x.isdigit() for x in password): #C
        [CA]app.error("Number ", "Your password must include at l

    else: #E
        app.info("SECURE", "*** YOUR PASSWORD IS SECURE ***") #F
```

## Building the main app

This section covers how to build the main GUI app. As in the previous chapters, we use create `App()` then display the name of the GUI, (Password Strength Checker) and set the height of the GUI window. Next, we change the background color to white because this is a serious app! Then add a title that introduces the GUI, setting the font size and the font type. Add the final code from listing 7.6.

**Listing 7.6 The main GUI**

```
# App
app = App("Password Strength Checker", width=700, height=300)
app.bg = "White"
title_text = Text(app, "Password Strength Checker")
title_text.text_size = 28
title_text.font = "Impact"

#enter text
text = Text(app, text="Enter your Password Below")
password_textbox = TextBox(app, width=50)

#button
button = PushButton(app, command=PasswordCheck, text="Check Passw

app.display()
```

The middle section in the code listing creates the text box where the user types or inputs their password for checking. Figure 7.13. This uses the `TextBox()` widget set to a width measurement of 50 to create a suitably sized input spot for the password. Then we add a button which, when pressed, calls and runs the `PasswordCheck()` function. The data in the `TextBox()` is passed to the `password` variable and the checks begin:

- is the password < 8 characters?
- does it contain a lowercase letter?
- does it contain an uppercase letter?
- does it contain a digit letter?

That wraps up the final program code. In the next section, we will check for any errors before running our GUI.

**Figure 7.13 Main lines of code and GUI features**


## Running and testing the GUI

This project is a fairly straightforward. You encountered most of the widgets in previous chapters. The `if` statement is probably the main area where errors may occur. To run the GUI, press **F5** on the keyboard, you may be prompted to save the program file. Choose **OK** and the program will save, and then the program will run.

If you encounter any errors, or the program does not run as expected, check for these common errors:

1. Ensure the indentation in the `if` statement is correct.
2. Remember to use the colon symbol : at the end of each conditional.
3. The lines of code that follow underneath the : must be indented.
4. Are the comparison operators / inequality symbols > and < the correct way round?
5. Ensure you have included the `not` operator and the `any()` function.
6. Compare the final code listing with your program code.

**Listing 7.7 Final code**

```
# Imports

from guizero import App, info, Text, TextBox, PushButton

# Variables

# Functions
```

```
def PasswordCheck():
    password = password_textbox.value
    if (len(password)<8):
        [CA]app.error("Length ", "Your password must be at least

    elif not any(x.islower() for x in password):
        [CA]app.error("Lower Case ", "Your password must include

    elif not any(x.isupper() for x in password):
        [CA]app.error("Upper Case ", "Your password must include

    elif not any(x.isdigit() for x in password):
        [CA]app.error("Number ", "Your password must include at l

    else:
        app.info("SECURE", "*** YOUR PASSWORD IS SECURE ***")


# App
app = App("Password Strength Checker", width=700, height=300)
app.bg = "White"
title_text = Text(app, "Password Strength Checker")
title_text.text_size = 28
title_text.font = "Impact"

#enter text
instruction = Text(app, text="Enter your Password Below")
password_textbox = TextBox(app, width=50)

#button
button = PushButton(app, command=PasswordCheck, text="Check Passw

app.display()
```

# Other things to try

In this chapter you created a Password Checker program where you enter your password into the GUI and then the program checks that it is a secure password, containing at least one uppercase and lowercase letter, a digit, and is eight or more characters long. In this section we will improve the GUI.

The first step is to save your file with a new name. This means that you still have your original working version if things go wrong or don’t work as

intended. To do this, follow these instructions:

1. Open your **Password_Checker.py** program code in your Python editor.
2. From the **File** menu, choose **File** > **Save As...**. A window opens displaying your folder where the **Password_Checker.py** file is saved.
3. Select and change the name of the file to **Password_Checker2.py.**
4. Press the **Save** button.

You now have a new version of the file which you can edit.

## Increasing the length check of the password

Did you know that a password that contains twelve characters takes 3,000 years to crack! This surely is an extremely secure password! The first other thing to try is to increase the length check of the overall password from eight to twelve characters. In the `# Function` section of the program find the line of code

```
if (len(password)<8):
```

Change the number 8 to 12.

You will also need to change the message that informs the user that their password no needs to be 12 characters in length. Find the line of code,

```
app.error("Length ", "Your password must be at least 8 characters
```

Again, change the number 8 to 12.

Press **Save** and run the program.

## Adding a button to clear the password

Currently our GUI program has no option to clear the password, or even a button to close and exit it. The program retains the password in the `TextBox()`and continues to display it even after every check. This is done so that the user does not have to keep typing in the password every time they are required to make an improvement.

For example, if you have a 12-character password and it is missing a lowercase letter, the message pops up to inform you and you enter in the old password, which now includes a lowercase letter. However, as you had to type out all twelve characters again, you make a mistake and type the digit in the incorrect place. You have a secure password, but it is incorrect, it is not the original password that you were using.

There may be some situations where you want to clear the password in the `TextBox()` and start again. This could be because you are checking a second password or checking someone else's password and you don't want them to see yours first. No good having a password that takes 3,000 years to crack and then you accidentally show it to someone!

So, this next other thing to try is to add a button that clears the password. In the `# Function` section of the program, create a new function below the `PasswordCheck():` which closes the GUI when a button is pressed.

Add the following code:

```
    else:
        app.info("SECURE", "*** YOUR PASSWORD IS SECURE ***")

def ClearPassword():
    password_textbox.clear()
```

Next, we need to create a new button and assign the new function to the command. This means that when the button is pressed, the program calls the `ClearPassword()` function, which clears any characters in the `TextBox()`.

In `# App` section of the program find the code where we create the buttons, underneath these lines, add the new line of code to insert the button:

```
[CA]button_clear = PushButton(app, command=ClearPassword, text="C
```

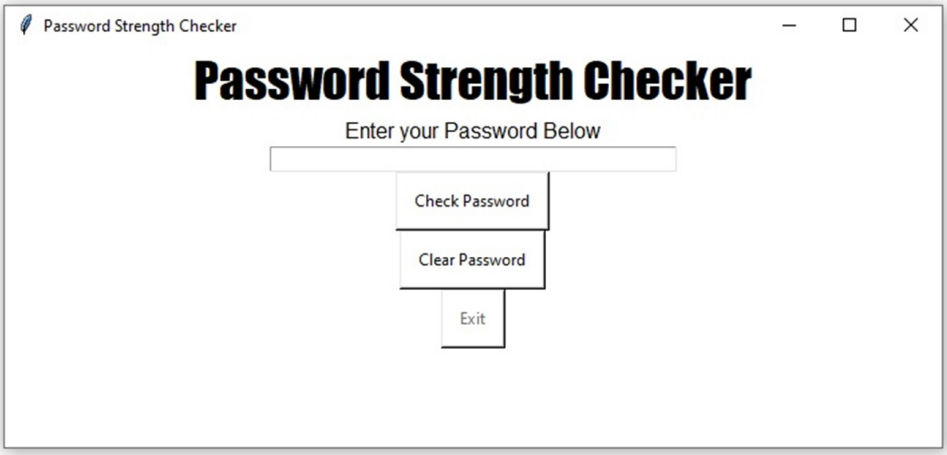
Save the program (press F5) and run and test the new GUI. Enter some random characters into the Text Box and then press the clear button.

## Closing the GUI

The third other thing to try is to add a button that closes the GUI. Currently to close the GUI you have to click the x in the top right of the GUI window. We are going to add a final button to the GUI window which appears underneath the other buttons. However, this new button will only become interactive when the current password being tested has passed all the checks. Buttons can be enabled and displayed in the program, this stops the user from accidentally clicking them or, in this case, we don't want them to be able to click the button yet.

This is an additional security feature that ensures that the user's password is secure before they can exit the program. To do this, when the program starts the exit button is greyed out and cannot be clicked. It is not until the password has passed all the checks that the button will be enabled (figure 7.14).

**Figure 7.14 Final GUI with Clear Password and Exit buttons added.**




## Enabling the button

Begin the edit by enabling the button in the `PasswordCheck():` function. The button is only clickable when all the checks are passed therefore the code is added to the else clause underneath `app.info()`. Add the line of code:

```
    else:
```

```
        app.info("SECURE", "*** YOUR PASSWORD IS SECURE ***")
        button_exit.enable()
```

## Creating the exit function

Next, we define a function to close the GUI. As with all functions, we start by creating the name and then include the line of code to display a message that reminds the user to keep all their passwords safe. The last line of the function code is `app.destroy()`, which closes the GUI. Locate the `# Functions` section and type up the code below:

```
def ClosePasswordChecker():
    app.info("Goodbye", "Remember to keep all passwords safe")
    app.destroy()
```

## Adding the exit button

To add the button to trigger the GUI to close, use the standard `PushButton()` code and include the command `command=ClosePasswordChecker` and the text label, Exit. When the button is pressed the `ClosePasswordChecker()` function is called and closes the GUI. Add the line of code below directly underneath the last button line of code in the `# app` section.

```
button = PushButton(app, command=PasswordCheck, text="Check Passw
[CA]button_clear = PushButton(app, command=ClearPassword, text="C
button_exit = PushButton(app, command=ClosePasswordChecker, text=
```

## Disabling the Exit button

Finally, we only want the exit button to be clickable once the password goes through the conditions in the if statement and is confirmed as a secure password. To do this, we disable the button when the GUI first runs. This still displays the button; however, it is greyed out and it stops the button being clickable. Under the exit button code that you just typed up, add the last line `button_exit.disable()`.

```
button_exit = PushButton(app, command=ClosePasswordChecker, text=
button_exit.disable()
```

## Checking the password for special characters

To ensure that a password really meets the highest level of security, special characters are always included as well as letters and digits. A special character is a character that is neither a letter nor a digit. It is one of the symbols usually found on the number keys on a keyboard. Common special characters are ! @ # $ £ % ^ & * ( ) -+ ? _ = , < > /

Let us add the lines of code to the program to check for special characters; this will ensure that the user has a really super secure password.

### creating a variable to hold the special characters.

First locate the variable section of the program, which is currently empty, and create a new variable called `special_characters`. Use a string to assign the symbols, ! @ # $ £ % ^ & * ( ) -+ ? _ = , < > / to the variable. You do not have to use all of them, just start with one or two to begin with and then build up the symbols in the variable.

```
# Variables

special_characters = "!@£#$%^&*()-+?_=,<>/"
```

**TIP**

Good programmers start small and build up. That makes it easier to test and debug your code. Then you can build up and add more special characters.

### Adding the elif statement to check for special characters.

Next, we need to create an extra `elif` statement to check for special characters in the password. This will make use of the `for` loop, the same as previously used, to tell the program to loop over each individual character (`x`) in the password and check to determine if it is a special character. The line of code also uses the `not` operator and the `any()`function, returning either True or False based on the password as a whole. Add the new elif statement after the digit and the `else` clause.

```
  elif not any(x.isdigit() for x in password):
        [CA]app.error("Number ", "Your password must include at l
    elif not any(x in special_characters for x in password):
        [CA]app.error("Symbol ", "Your password must include a sp
    else:
        app.info("SECURE", "*** YOUR PASSWORD IS SECURE ***")
        button_exit.enable()
```

To run the new GUI, press **F5** on the keyboard. You may be prompted to save the program file. Choose **OK** and the program will save, and then the program will run.

If you encounter any errors, or the program does not run as expected, then check the section titled “Running and testing the GUI” for support and solutions.

**Listing 7.8 Final code**

```
# Imports

from guizero import App, info, Text, TextBox, PushButton

# Variables

special_characters = "!@£#$%^&*()-+?_=,<>/"

# Functions

def PasswordCheck():
    password = password_textbox.value
    if (len(password)<12):
        [CA]app.error("Length ", "Your password must be at least

    elif not any(x.islower() for x in password):
        [CA]app.error("Lower Case ", "Your password must include

    elif not any(x.isupper() for x in password):
        [CA]app.error("Upper Case ", "Your password must include

    elif not any(x.isdigit() for x in password):
        [CA]app.error("Number ", "Your password must include at l
```

```
    elif not any(x in special_characters for x in password):
        [CA]app.error("Symbol ", "Your password must include a sp

    else:
        app.info("SECURE", "*** YOUR PASSWORD IS SECURE ***")
        button_exit.enable()

def ClearPassword():
    password_textbox.clear()

def ClosePasswordChecker():
    app.info("Goodbye", "Remember to keep all passwords safe")
    app.destroy()

# App
app = App("Password Strength Checker", width=700, height=300)
app.bg = "White"
title_text = Text(app, "Password Strength Checker")
title_text.text_size = 28
title_text.font = "Impact"

#enter text
instruction = Text(app, text="Enter your Password Below")
password_textbox = TextBox(app, width=50)

#button
button = PushButton(app, command=PasswordCheck, text="Check Passw
[CA]button_clear = PushButton(app, command=ClearPassword, text="C
button_exit = PushButton(app, command=ClosePasswordChecker, text=
button_exit.disable()

app.display()
```

# Statements of fact

- Special characters are !@£#$%^&*()-+?_=,<>/ and can be used to make a password more secure.
- A loop is a section of the program code that repeats until a condition is met. This is an efficient way to repeat sections of code by reusing the same code.
- A `for` loop repeats a section of code. The loop repeats for each item in a string.
- Typically in Python the letter `x` is used to hold each character in a longer string, in this chapter the `x` is used to hold each letter in the password as

it is checked.

- `islower()` checks for lower case letters and is useful for checking the case of a string or character.
- `isupper()` checks for upper case letters
- `isdigit()` checks for digits in a string.
- Elif is used to assess one or more conditions. The `elif` statement only runs when the if condition is not met, this means the program does not have to check every condition which saves processing time.
- The `not()` operator returns a True or False value where a condition is not met, this is useful for program responses where a condition is not met.
- `elif not` checks if a condition is *not* met. The program responds when a condition is not met.
- The `any()` function returns a value of True or False if at least one element of any does not meet the condition. This is useful as you can check the individual entries and characters in strings and lists.
- Combining `elif not` and `any()` returns a value of True if at least one element of the condition is not met and False if the condition is met.
- Buttons can be enabled and disabled. This prevents a user from accidentally clicking the button. The GUI can make a button available only when certain conditions are met.

# Appendix A. Installing Python on Microsoft Windows

This appendix walks you through downloading, installing, and testing Python on a Microsoft Windows computer. If you are using macOS or Linux, or another operating system then check out appendix B which provides the instruction to do this.

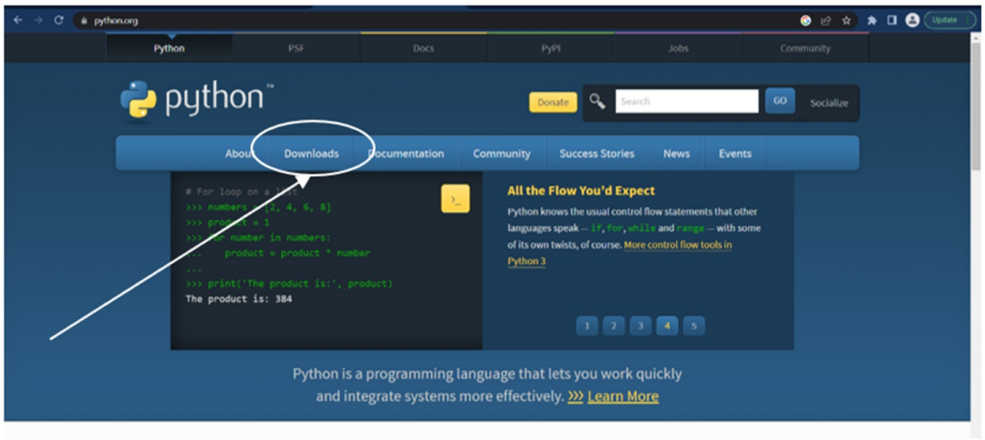
Python is a free programming language. This means that you can use Python for free to create, edit, and share your programs. Python is popular because you can use it to write programs for many different applications and purposes. However, its real strength lies in the simplicity of using the code compared to other languages. You can learn more about Python in appendix D.

To code with Python, you need an integrated development environment (IDE). An IDE is an interface that you can use to write programs, test them, and run them on your computer. The Python installation includes its own IDE, called IDLE, which is pronounced idol. There are other IDEs available; an overview of the alternatives is covered in the last section of this appendix.

## Getting Python

Let's begin. The first step is to head over to the Python website and download the Python install program so that you can install it on your computer. Open your web browser, (like Chrome, Safari, Edge etc.) and go to www.python.org. This will take you to the home page for Python. Over time newer versions of Python are released, so remember to check this site regularly if you wish to upgrade your version of Python at a later date.

**Figure A.1 Downloading Python**




# Downloading Python

On the Python home page, find the navigation menu at the top of the website and select the **Downloads** button. This takes you to the downloads page where you can select which version of Python you want to install.

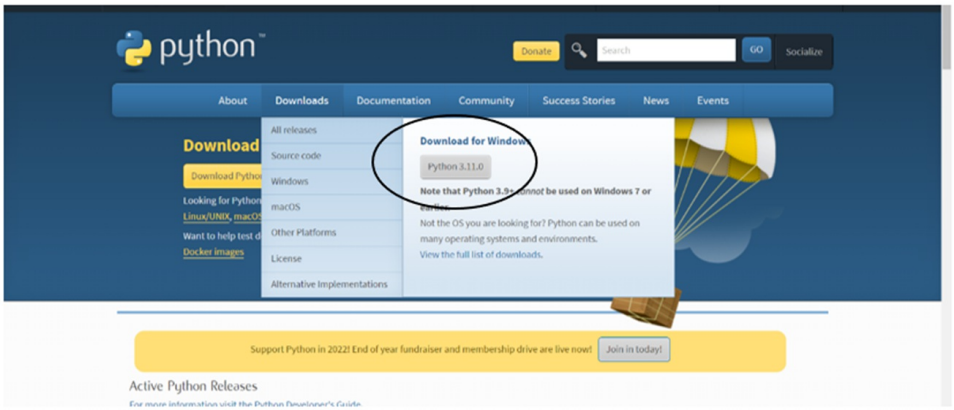
When the Downloads page opens, the site automatically displays the most up to date version of Python for the Microsoft Windows operating system. Click the button to download the latest version.

**Note**

If you are using Mac or Linux, click the link underneath the download button to navigate to the appropriate page. (Remember to check out appendix B for a detailed guide on installing on macOS and Linux.)

To begin downloading the Python installation file, click the download button. It’s okay if the version number is different from what you see here.

**Figure A.2 Select the current version of Python**

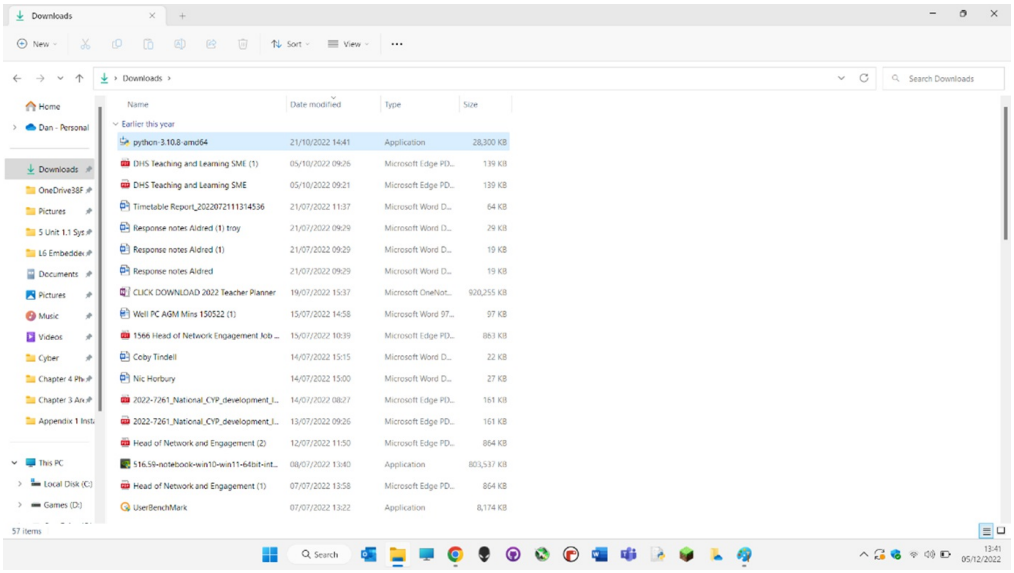



# Installing Python

Once the Python installation file downloads, on your computer, use the folder browser and head to the folder where the Python installation file has been downloaded to. In Windows, unless you changed the download location, this is probably your Downloads folder.

1. Locate the downloaded Python file,

**Figure A.3 Locating the Python installation file**



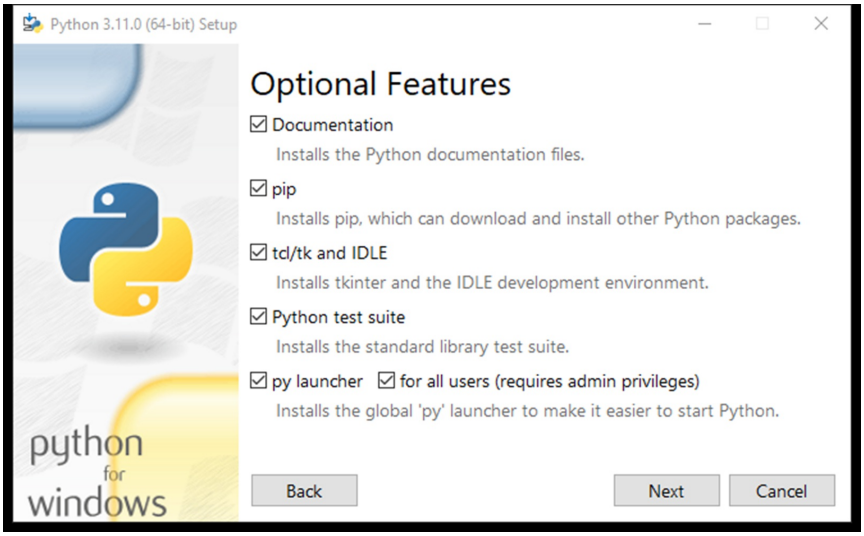

2. Once you've found the file, select it (usually double-click) to start installing Python . You will be presented with a menu window that has

several options. Select the second option: **Customize installation**.

**Figure A.4 Install Python**

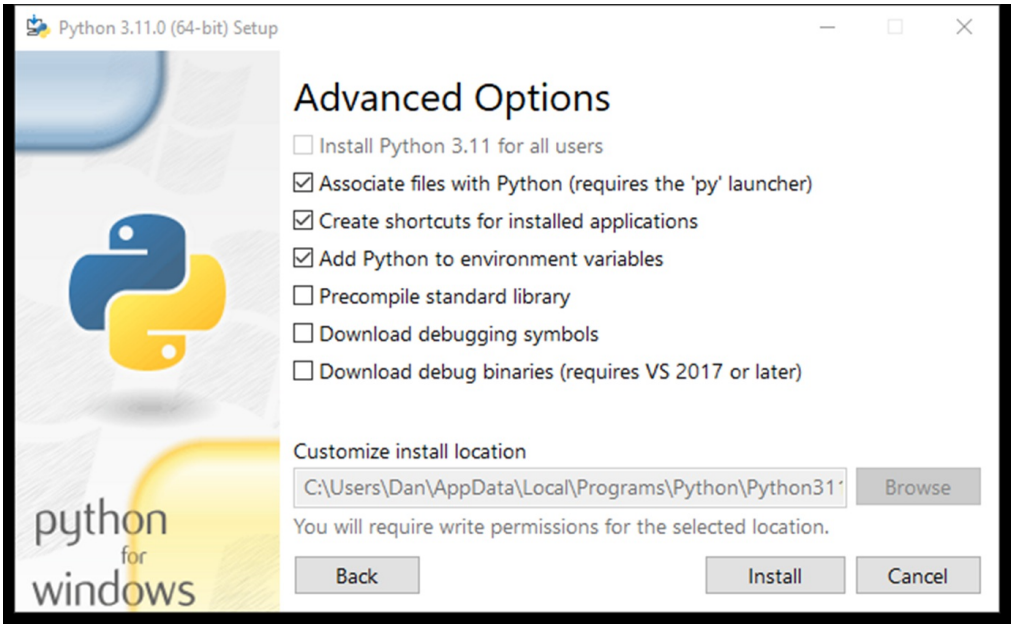

3. The Python installer loads the **Optional Features** window. All these options are ticked. This is correct.

**Figure A.5 Select the optional features**




4. Press **Next** to move to the **Advanced Options** window. This window presents you with several options, some of which may already be ticked. Leave any ticked options, do not untick them. There is an additional option that you may want to select, and one option that must be selected.

**Figure A.6 Select Advanced Options**



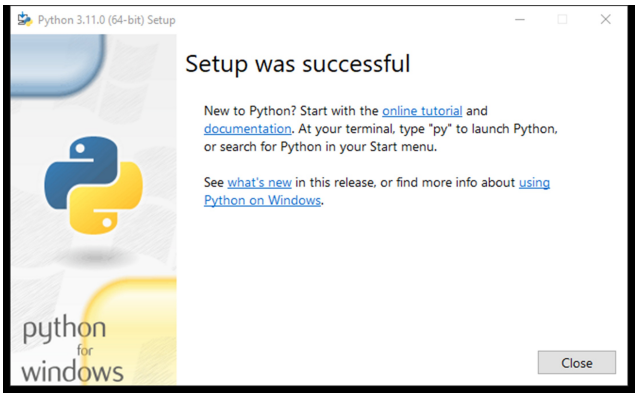

a. The first option, the first box, is to decide whether you want to **install Python for all users** of the computer. Say for example that you are using someone else's computer. They might not want Python installed. In that case, leave this option unticked. If the computer is a shared computer and used by others who may also want to build the projects in the book, then they will need Python installed, so you can tick this button.
b. The second option, the fourth box is **Add Python to environment variable**. This option is essential and must be ticked. The option ensures that your Windows operating system can locate Python and all Python's associated files. Select the **Add Python to environment variable** option.

5. After you have selected these two options you are ready to install Python. Press the **Install** button. Unless you change the location, this will install Python onto the C drive of your computer

**Figure A.7 The Install button**

6. You will be presented with the final window that asks you to confirm that you want to install Python. Select **Yes** to begin the installation.
7. On completion of the installation, the next window confirms that the installation was successful. Well done! You have installed Python. Now to test it.

**Figure A.8 Confirmation that setup was successful**

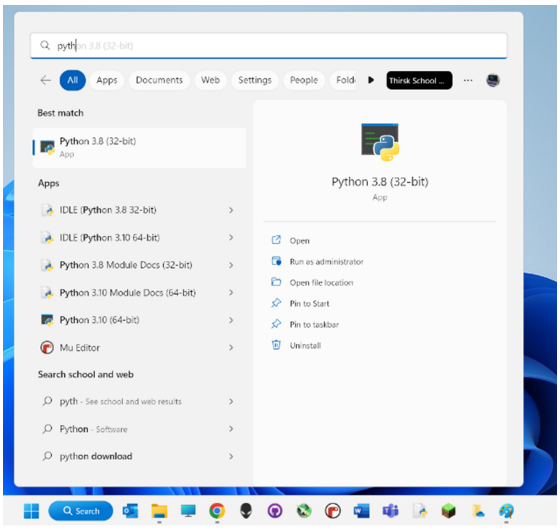



# Testing your installation of Python

Now to test that Python has installed correctly and is working.

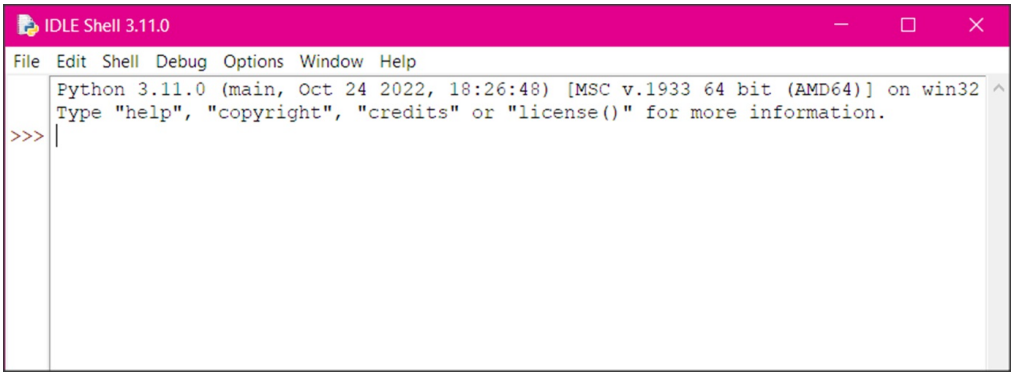
8. Open the Windows start menu. If you are using Windows 10 then you will see that Python has been added to the top of the menu under the **Recently added** section. If you are using Windows 11 then simply type Python into the search bar.

**Figure A.9 Searching for Python in Windows**



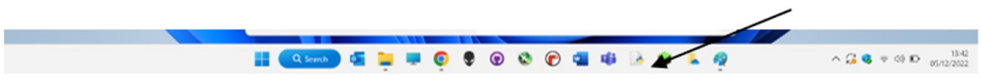

9. Double-click the IDLE program. Python loads in an **IDLE Shell** window. The shell window is an interface in which you can enter Python code one line at a time. You write the line of code, press enter, and then the code runs, displaying the output in the shell window.

**Figure A.10 The IDLE Shell**




10. Locate the Python icon on your Windows task bar at the bottom of your monitor's screen. Right click and select **Pin to Taskbar**. This makes Python always on your taskbar, which saves you having to look for the program in the future.

**Figure A.11 Pinning Python to the task bar**



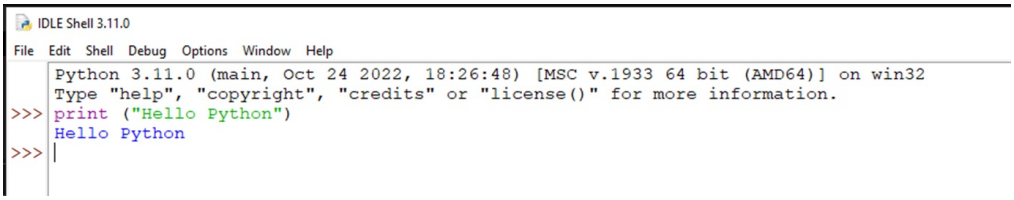
Let’s write a simple program to test that Python is working.

1. In the IDLE Shell window, click where the >>> is displayed. The >>> is called a prompt.
2. Type in this line of text exactly as it is shown here. This is programming code!

```
print("Hello Python")
```

3. Press **Enter** on your keyboard.
4. The Python Shell shows the words `Hello Python` underneath the line of code you just wrote.

**Figure A.12 Python output window**




You’ve successfully installed Python and written your first program. If you want to learn more about the Python programming language, then be sure to check out appendix D for an overview of Python’s uses and functions.

# Other IDEs you may want to try

IDLE is a free IDE provided with Python. IDLE is standard and contains many features that are included in other IDEs. Some people say IDLE’s interface is unappealing. You can try other IDEs, such as the ones in the following list, to find one that you prefer or if you want a more colorful coding experience.

- Mu. This IDE describes itself as a *“simple Python editor for beginner programmers.”* It has a nice eye-catching interface. Download here: https://codewith.mu/.
- Replit. This IDE is cloud based, which means you don’t have to download and install any software. The advantage to cloud-based

software is that you can access you projects on any device that has a live data connection or is connected to the Internet. Before using Replit, you need to create a free account. You can access Replit at https://replit.com/.

- Thonny. This IDE describes itself as a "*Python IDE for beginners.*" It provides a colorful user interface and has stripped out all the features that may distract a user. Download here: https://thonny.org/.
- Visual Studio Code. This IDE describes itself as "a lightweight but powerful source code editor which runs on your desktop and is available for Windows, macOS and Linux. It comes with built-in support for JavaScript, TypeScript and Node.js and has a rich ecosystem of extensions for other languages and runtimes." Note that to run Python code you must install an additional extension, and then you have to configure the setup.

# Appendix B. Installing Python on other operating systems

This appendix walks you through downloading, installing, and testing Python on operating systems other than Windows, such as macOS, or Linux, or Chromebook.

Python is a free programming language. This means that you can use Python for free to create, edit, and share your programs. Python is popular because you can use it to write programs for many different applications and purposes. However, its real strength lies in the simplicity of using the code compared to other languages. You can learn more about Python in appendix D.

To code with Python, you need an integrated development environment (IDE). An IDE is an interface that you can use to write programs, test them, and run them on your computer. The Python installation includes its own IDE, called IDLE, which is pronounced idol.

There are other IDEs available although these alternatives may not run on all operating systems. This book is written using the standard IDE that comes with Python, which means that if you are searching for your own instructions or videos, you should search for:

**How to install python idle on x**

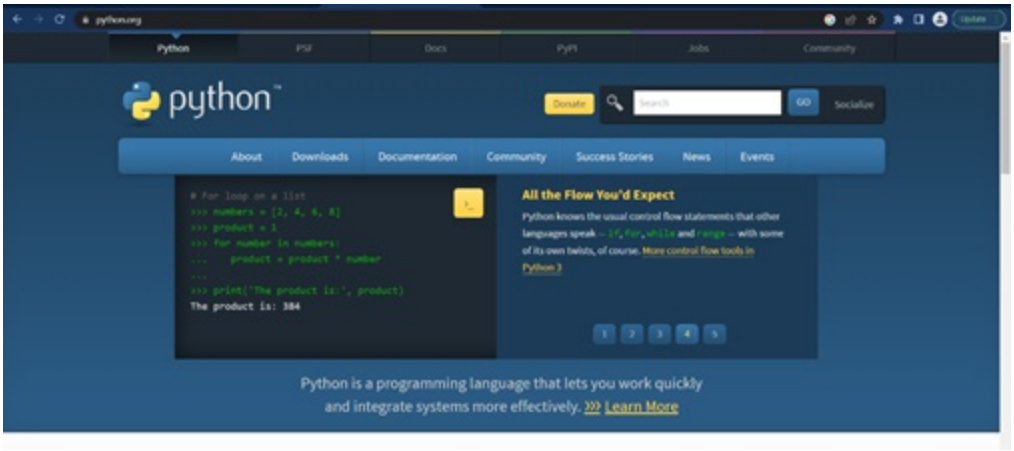
Replacing the x with your operating system, for example:

**How to install python idle on Linux Ubuntu**

This will ensure that the guizero program will work correctly and you can build the projects in each of the chapters.

## Getting Python

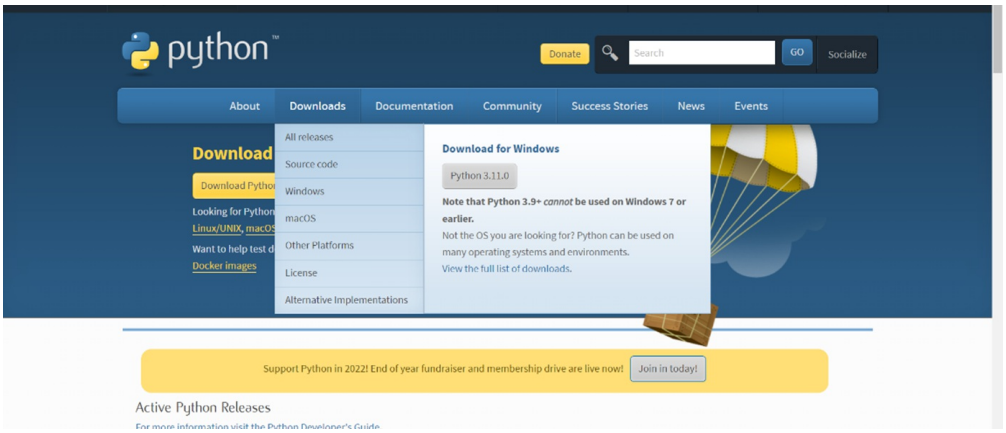
Let’s begin. The first step is to head over to the Python website and download the Python install program so that you can install it on your computer. Open your web browser, (like Chrome, Safari, Edge etc.) and go to www.python.org. This will take you to the home page for Python. Over time newer versions of Python are released, so remember to check this site regularly if you wish to upgrade your version of Python to the latest release.




# Downloading Python

On the Python home page, find the navigation menu at the top of the website and select the **Downloads** button. This takes you to the downloads page where you can select which version of Python you want to install.

When the **Downloads** page opens, the menu displays all the download options in a list. Click the tab for the OS that you are using, for example, if you want to install Python on macOS, then select this option.




To begin downloading the Python installation file, click the download button. It's okay if the version number is different from what you see here.

There are many versions of Python available on the main website, so if you are using a different OS, be sure to check out the downloads page.

# Installing Python on macOS

Select the macOS option which will take you to a page that displays all the available versions of Python. As shown below,

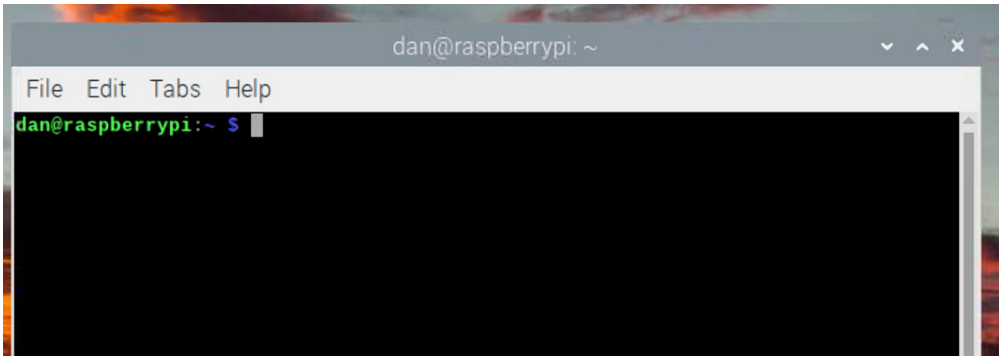

Select the version that you want, usually the latest stable release and then locate the Download and select the `macOS 64-bit universal2 installer.` This will download the Python set up program. Once the Python installation file downloads, on your computer, use the folder browser and head to the folder where the Python installation file has been downloaded to. Locate the

downloaded Python file.

When you've found the file, select it (usually double-click) to start installing Python. You will be presented with a menu window that has several steps. For further details and a video see this weblink, https://www.jcchouinard.com/install-python/#Install_Python_on_MacOS.

# Installing Python on Linux

There are many versions of Linux so covering the installation of all versions is not possible. Therefore, this section just covers installing Python using the command line. To use the command line, you have to have permission to access the Terminal window.




Many versions of Linux already have Python installed so the first step is to check if it is installed.

To do this, open a Terminal window and enter,

```
python --version
```

If the response displayed says that Python cannot be found, then it means that Python has not been installed. Linux may then provide you with a set of instructions covering how to install Python, follow these and Python will install. If not then follow these simple steps,

1. First in the terminal window type, `sudo apt install idle3`
2. Press enter to begin the installation.
3. Once installation has completed, go to your search bar and type in **idle**

4. This will find Idle, click the logo to load it.
5. You can watch a video about the install here: https://www.youtube.com/watch?v=l0g2_7csJs8

# Installing Python on Chromebook

Installation of Python on a Chromebook is similar to the Linux installation, complete the following steps,

1. Open a terminal window
2. Enter the command `apt install idle3` into the prompt.
3. Press Enter

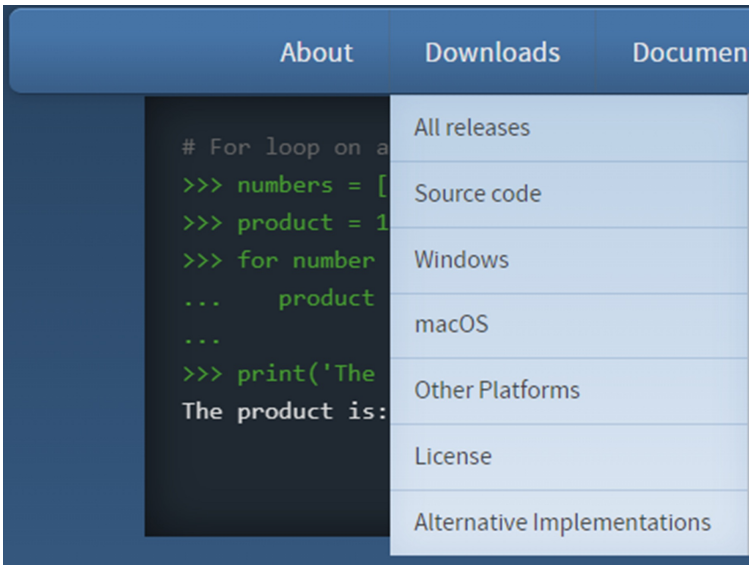
If this does not work then you may need to install Python using pip, follow these instructions,

1. Open a terminal window.
2. Enter the command `apt install python3-pip`
3. Press the enter key

For further details check out this useful article: https://ninja-ide.org/install-python-on-chromebook-without-linux/.

# Other versions

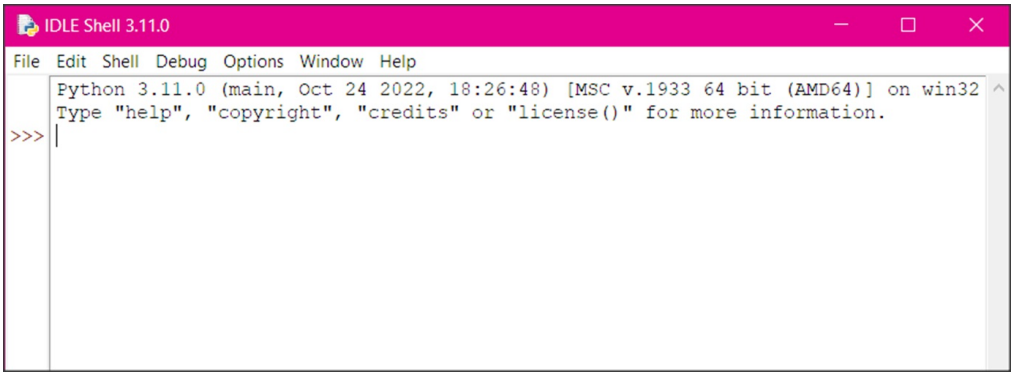
There are many versions of Python available for many operating systems, including those that run on tablets and mobile devices. Check out the Alternative Implementations on the downloads page to find out more.




# Testing your installation of Python

Now to test that Python has installed correctly and is working. Complete the following steps,

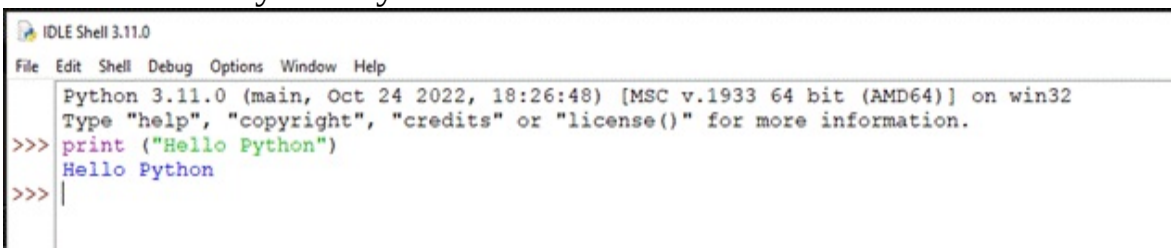
1. Open the Python program which, depending on which operating system you are using, may be in a recently added section, the desktop or the main menu.
2. Double-click the IDLE program. Python loads in an **IDLE Shell** window. The shell window is an interface in which you can enter Python code one line at a time. You write the line of code, press enter, and then the code runs, displaying the output in the shell window.




Let’s write a simple program to test that Python is working.

3. In the IDLE Shell window, click where the >>> is displayed. The >>> is called a prompt.
4. Type in this line of text exactly as it is shown here. This is programming code!

```
print("Hello Python")
```

5. Press **Enter** on your keyboard.
6. 

   The Python Shell shows the words `Hello Python` underneath the line of code you just wrote.

You have successfully installed Python and written your first program. If you want to learn more about the Python programming language, then be sure to check out the projects, and of course, read and try all the examples in this book.

# Appendix C. Installing guizero on other operating systems and features of guizero

You may remember from chapter 1 that GUI stands for Graphical User Interface and is pronounced “gooey.” A GUI offers an easy way for the user to interact with a computer or device using graphics and images. Guizero is a programming library that contains all the code and widgets to create and build GUIs in Python.

There are several methods to install guizero onto your computer and it has even been designed so that you can simply download the program files and build GUIs.

Chapter 1 tells how to install guizero on a Windows computer. This appendix covers how to install guizero on other operating systems such as macOS, the Raspberry Pi, and Linux. It also covers the guizero Easy Install method which requires no installation; you simply download the required files and then guizero is ready to use.

The Easy Install method is useful if you are not permitted to download and install software on your computer. For example, if you do not have the required access permissions, or you are using a computer that you share with other people. Maybe you are using someone else’s computer and they do not want you to install additional software!

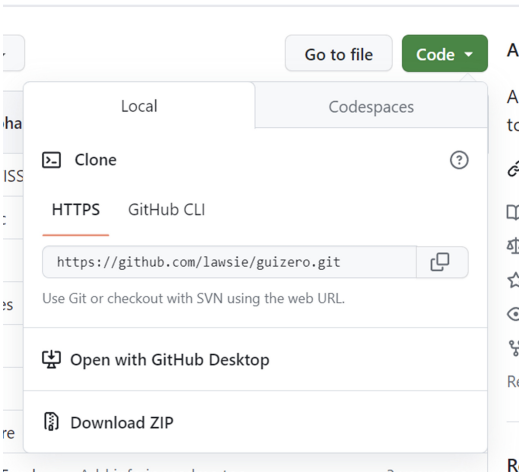
The last section of this appendix covers the key features of the guizero. Let us start with the Easy Install method first.

## Easy installation

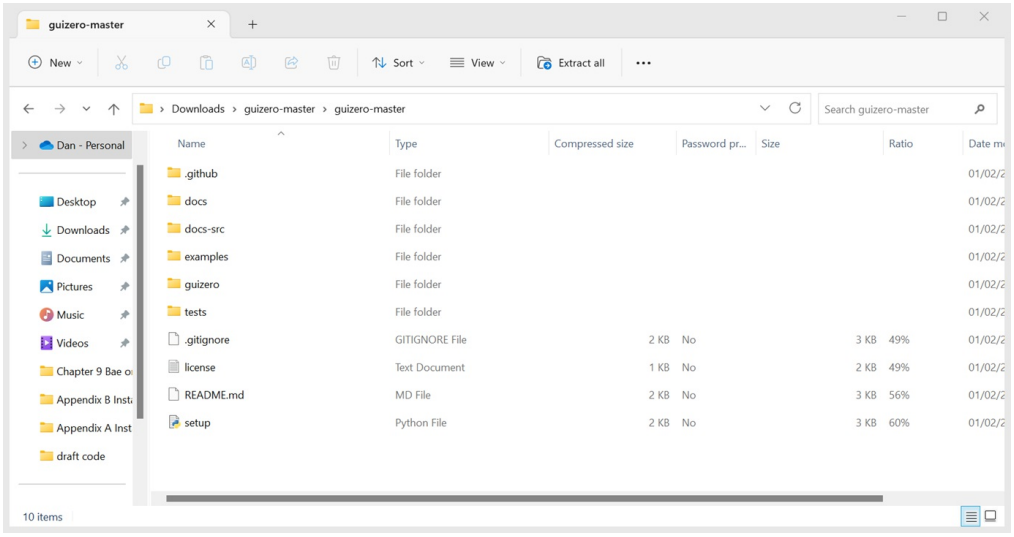
This installation is the simplest and easiest as it requires no installation of the software and therefore no special permissions or administrators’ rights. To

use this method, complete the following steps.

1. Open your internet browser.
2. Go to the link [https://github.com/lawsie/guizero](https://github.com/lawsie/guizero).
3. This opens the guizero page, which holds all the code and documents.
4. Find the green code button and click it.




5. Select the **Download Zip** option. This downloads the guizero files.
6. Open the ZIP folder. This folder contains a folder named guizero-master.
7. Open the **guizero-master** folder.




8. Select all the files and folders and copy all the files into a folder on your computer.

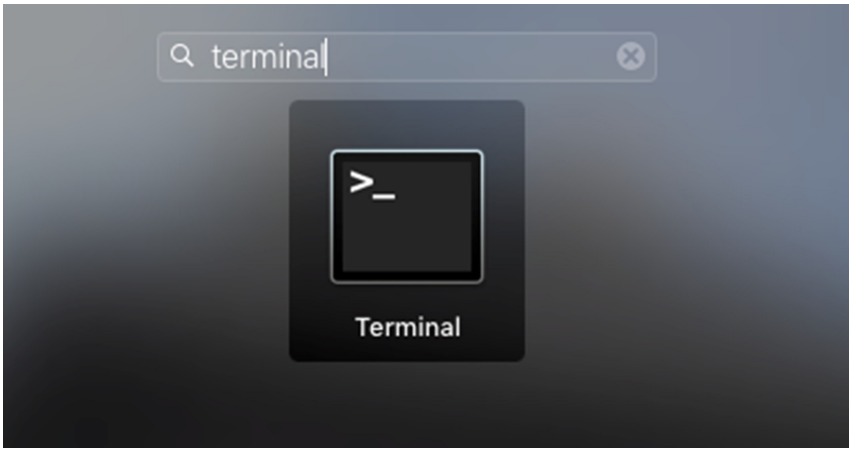
**Important**

Now all your GUI projects and code will run, *as long as you save your Python file into the same folder* as the guizero files. You can read more details at the official guizero website https://lawsie.github.io/guizero/.

# Installing guizero on macOS

To install guizero on macOS complete the following steps.

1. Open the terminal window click selecting **Applications** > **Utilities** > **Terminal** (or type **terminal** into the search bar).
2. In the terminal window type `pip3 install guizero`.
3. Press **Enter** to install guizero.




# Installing guizero on a Raspberry Pi

To install guizero on a Raspberry Pi complete the following steps.

1. Open the terminal window.
2. In the terminal window type `sudo pip3 install guizero`
3. Press **Enter**.
4. This installs guizero.

# Installing guizero on Linux

To install guizero on a Linux device complete the following steps.

1. Open the terminal window.
2. Ensure that `tkinter` is installed by entering `sudo apt install python3-tk`.
3. Install guizero by typing `pip3 install guizero` or `sudo pip3 install guizero`.

There are other versions of Linux. If you are using the Debian version, then you can install guizero in the terminal using the following command:

```
sudo apt-get install python-guizero
```

# Installing additional features

Guizero has additional features that are used to edit images and graphics within a GUI, such as resizing images or playing an animation. Each chapter in the book covers any additional image editing that you need to do. Most projects do not use animation. If you want to use these additional features of guizero, you will need to install guizero with the `pip` command (table C.1).

**Table C.1 Installing additional guizero features using the pip command**

| Operating system | Command |
|---|---|
| Windows and macOS | pip3 install guizero[images] |
| Linux or Raspberry Pi | sudo pip3 install guizero[images] |
| Any | pip3 install pillow |

**Note**

These additional image features are not available if you use the Easy Install method.

# Upgrading guizero

Over time, guizero will be updated and you will want to keep your version up to date. If you installed guizero using pip, then you can upgrade guizero in

the terminal window using the commands in table C.2.

**Table C.2 Upgrading guizero**

| Operating system | Command |
|---|---|
| Windows and macOS | pip3 install guizero --upgrade |
| Linux or Raspberry Pi | sudo pip3 install guizero --upgrade |

# Features of guizero

For ease of understanding guizero and its features, you can split guizero into two distinct groups. Group one contains the main categories of the software (table C.3). These categories are called elements and are the overall building features of a GUI. Elements allow you create interactivity between the user and the GUI. Interactivity is where, for example, the user presses a button, and the GUI displays a message via a popup. Or the user presses a button, and it triggers a sound to play.

These elements also enable you to build a GUI. For example, using the Layout element, you can adjust the size of the GUI window and position buttons and text within the GUI window. With the 'Events' , you can program an action to respond (called triggering an action) when something happens; for example, when a user double-clicks the left mouse button on a picture, some text is displayed.

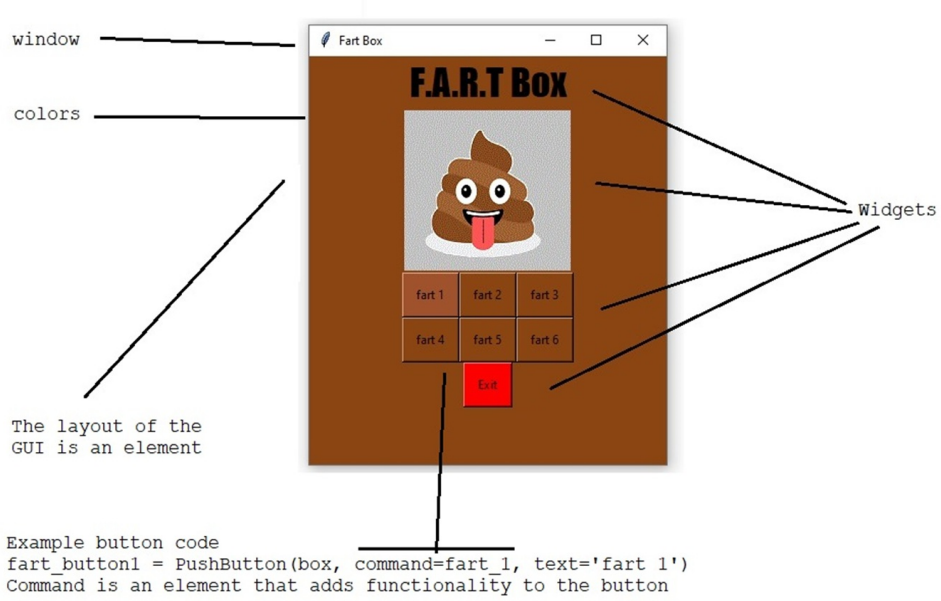
**Table C.3 The elements of guizero**

| guizero Element | Description | Example use with button widget |
|---|---|---|
| Colors | Edit the colors | Color can be set on most widgets, text, and backgrounds, customizing the feel of the GUI. |
| Commands | Create interaction in the GUI; for example, pressing a button or selecting an option | Used to assign functions to objects, how the GUI responds to a button press. |
| | | |

| | | |
|---|---|---|
| Event | Assigned to a widget that triggers an action based on mouse and keyboard inputs. | GUI displays an image when a button is clicked once and changes the image when the button is double clicked. |
| Images | Manage images within the GUI window | Display an image in your GUI, an animation or make a button a picture button. |
| Layout | Controls how widgets are arranged in the GUI | Stacks the buttons in a particular order or uses auto layout to arrange the buttons. |
| Loops | Used to repeat a section of the program code and ensure the GUI always responds to user. | GUI enters a loop which waits for events (user input) to happen and then respond. |
| Pop-Ups | Windows that pop up with a message or information for the user | Informing / updating the user. Popups can use the warn, info and error popups, prebuilt into Windows OS. |
| Sizes | Used to set a widget's width and height | Makes a button a certain size within the GUI. |
| Widgets | The objects that appear in your GUI (see table 1.2) | Used to build the GUI and add interactivity. |
| Windows | Creates extra windows in your GUI | The space in which the GUI and its features are held. |

The second group is widgets. Widgets are how you physically build the GUI; they are the main building blocks of a GUI. There are fifteen widgets that you can use. Each one is an object (think of objects as an item) that appears within the GUI, everything from the app itself to text boxes, buttons, and pictures. Some of the widgets are used with the Elements of the GUI. For example, the Box widget is a container that holds other widgets and is useful for neatly grouping them together. The Box widget is used as part of the Layout Element to create and customize the layout of the GUI. Table C.4

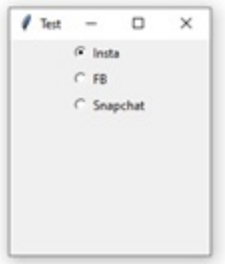
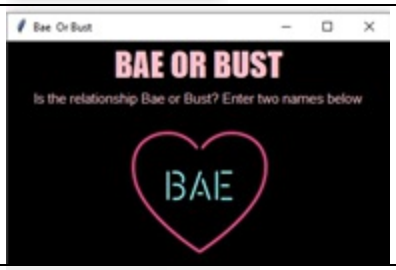
describes what all the guizero widgets do.

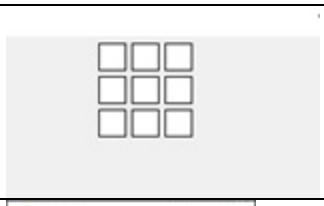



Some widgets are similar but have different properties. For example, the PushButton widget creates a clickable button, whilst the `ButtonGroup` widget creates a button with options where the user can select from several choices. You will use the widgets throughout the various projects, and you can refer to tables C.3 and C.4 as required.

**Table C.4 The guizero widgets**

| guizero Widget | Description | Example use |
|---|---|---|
| App | The main window of the GUI that holds all the other widgets |  |
| Box | An invisible container that holds other widgets, useful for grouping objects together | Useful for organizing and layout of the widgets in the GUI window. |
| ButtonGroup | A group of radio buttons that enable the user to | |

| | | |
|---|---|---|
| | make a choice |  |
| CheckBox | A box that can be ticked or unticked |  |
| Combo | Displays a drop-down box where a single item can be selected from a list |  |
| ListBox | Displays a list of items where one item can be selected from |  |
| MenuBar | Creates a drop-down menu at the top of the GUI |  |
| Picture | Displays an image in the GUI |  |
| PushButton | A button with either text or an image that runs a function when pressed |  |
| Slider | Displays a bar that can be | |

| | | |
|---|---|---|
| | used to select a specific value within a range |  |
| Text | Displays non editable text in your app |  |
| TextBox | Displays a box which the user can type into |  |
| TitleBox | An invisible container where other widgets can be grouped together and display a title |  |
| Waffle | Displays a grid of squares |  |
| Window | Creates a new window in the GUI |  |

# Appendix D. Downloading code and resources from GitHub

All the program codes and resources in this book are stored on a website called GitHub. This appendix introduces you to GitHub and walks you through how to access each of the chapter project files and download them. GitHub is a website used by coders and developers to manage and store their code. Often a website like GitHub are referred to as a repository or repo for short. So why not use another cloud-based storage instead, such as OneDrive, Google, or Dropbox?

The reason is because GitHub uses version control. Version control is a way to keep track of all different versions of documents, such as program files. Consider the example where you start writing a program and over the course of developing it you make 10 changes to the code. If you are using GitHub, then each of the previous 9 versions are stored and available on GitHub. This means that if you break the program code you can return to a previous version that worked. Furthermore, you can share any of these versions with other people without the worry that they will destroy or break the current code.

#### Sharing files on GitHub

Sharing uses two techniques which enable people to collaborate safely without risk of damaging each other's code or contributions:

A process called **branching** allows others to duplicate a program and then amend, correct, and update, making changes to the code without affecting the original version.

A process called **merging** allows changed code to be submitted back to the original code as a suggested edit. The original programmer, if appropriate, can then merge the code back into the main source code making it the updated version.

The good news is that all these changes are tracked and can be reversed if needed.

This appendix tells you how to get the project files from GitHub.

# Accessing the project files

To access the files from each chapter project, open the following GitHub page:

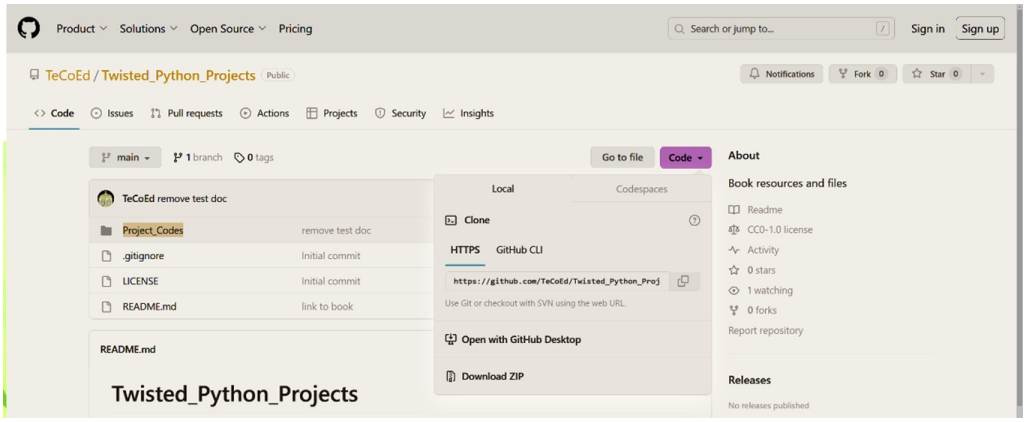
https://github.com/TeCoEd/Twisted_Python_Projects

This takes you to the main page of the book's repository where it is possible to download all the code and resources. The repository will also always contain the most up to date versions of the code.

# Downloading all the resources

To download all of the Python program files and the other resources, such as sound files and images, press the Code button which will display a menu (figure D.1).

**Figure D.1 Press the Code button. You will get a dropdown menu with options.**




Locate the **Download ZIP** option at the bottom of the menu and press it. This will start the process of downloading a ZIP folder of all the files to the Downloads folder on your computer. Once complete, you can extract and

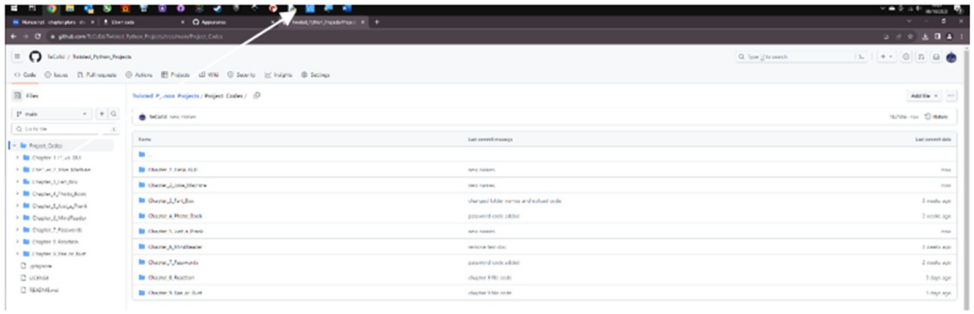
open the folder and use the files as required. This is the fastest way to get all the program code and resources from the book.

# Downloading individual code files

You may need to only download one resource or file instead of the complete set. To do this,

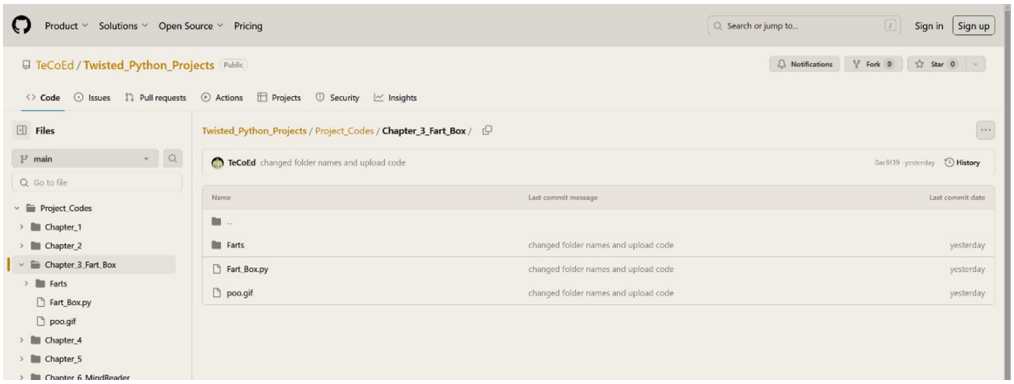
1. Click the **Project_Codes** link which will open and display the chapter folders (figure D.2).

**Figure D.2 The Project_Codes folder contains code for all the chapters**



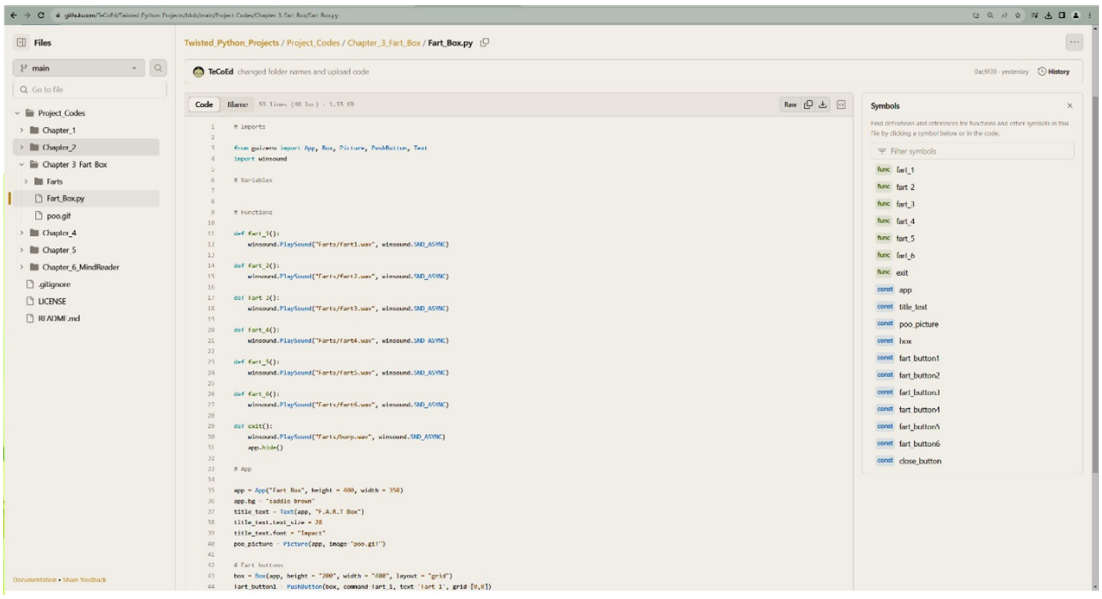
2. Then select the required chapter folder and click it, for example, chapter 3 Fart_Box (figure D.3), which opens the folder and displays all the resources and the code for that project.

**Figure D.3 Open the chapter folder**



3. To download the Python program file named Fart_Box.py, first click the file. This opens the code in a new window within GitHub (figure D.4).

**Figure D.4 The Fart_Box.py program code**




4. Next locate the download button toward the top right of the window and press it (figure D.5). Pressing this button downloads the program file to your Downloads folder, where you can access the file.

**Figure D.5 Press the download button to download the file**

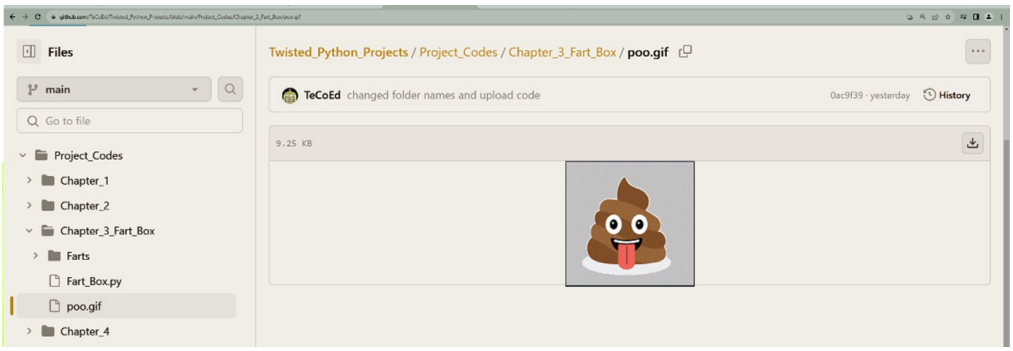

5. An alternative option to access the code is to click the middle icon, the Copy raw file option. Pressing this button copies the code. Open your Python editor and you can then directly paste the program code into it.

# Downloading image and sound files.

Files which are not program files, such as the images and sound files, can be downloaded in a similar way. For example, the Chapter_3_Fart_Box folder contains an image file. To download this image,

1. Click the file name of the image. This opens the image in a new window where you can view it (figure D.6).
2. Click the download icon to save the image to your Downloads folder.
3. An alternative download method is to right-click on the image and select the “Save image as…” option from the drop down menu. Select the folder location where you want to save the image and press Save.

**Figure D.6 Press the download button to download the file**




To download a sound file, follow these steps:

1. If there are several audio files, first open the folder that contains the audio files.
2. Click the name of the audio file that you want to download.
3. A new window opens. This window displays the words “View raw”.
4. Click either the View raw or the download icon to download the file to your device.

Remember that for guizero to access and use the downloaded files they need to be in the appropriate folder. (guizero will not automatically link to the files in the Downloads folder.)

Ensure that you move the files to the required folders. These folders are referenced in each chapter. You can use your own folder locations and file names, but ensure that you edit the file location information in your program code to match this.