

# Raspberry Pi and Qt Tutorial

by Timothy Johnson and Michael Parkinson

Imagine if Google Chrome or Microsoft Word had a text-based user interface, a black and white screen that could only display text and be controlled with a keyboard. In the 1970's, this was the only type of computer program in existence, but now most programs have a graphical user interface. Previously you learned the basics of programming in Python, and we used a text-based interface (the console) because it is easy to learn. Today, you are going to learn how to make your programming more modern and user-friendly by incorporating a graphical user interface. You will also learn how to integrate your programming with the real world using electronics.

First though, let's get started by familiarizing yourself with the Raspberry Pi 3, the device that you will be using to develop your project. It is a \$35 microcomputer that runs a version of Linux called Raspbian, and has Wi-Fi, Bluetooth, an HDMI display output, and 4 USB ports. Besides how cheap it is, the main attraction of the Raspberry Pi is its GPIO ports, which make it easy to connect and control electrical circuits without needing any other hardware.

## Connecting to the Pi

RealVNC is the software we are going to use to remotely connect to the Pi. It is free, compact, and easy to use, not complex software like AutoCAD or LabVIEW. If your laptop has an Ethernet port, you can connect a Pi to your laptop using an Ethernet cable and use your laptop as a monitor, keyboard, and mouse on the Pi. If your laptop doesn't have an Ethernet port, it has at least one USB port, so ask your instructor for a USB to Ethernet adapter that you can use.

You can download the installer for the VNC Viewer from the Resources page on eLearning. The installers for both Windows and macOS should be posted there.<sup>1</sup> If you are running Windows, you may need to install software called Bonjour to make the Ethernet connection work, which can also be downloaded from eLearning. (If you have iTunes installed you can skip this step because you already have Bonjour.)

Once you have downloaded the VNC viewer (and installed Bonjour if necessary), launch the VNC Viewer and accept any prompts it gives you. Type `raspberrypi.local` in the textbox at the top of the window and press Enter to connect to it. If it asks whether it is safe to connect, select "Continue". When it prompts for authentication, the password is `raspberry`, and you can optionally select the "Remember password" option so you don't want to have to enter it again.

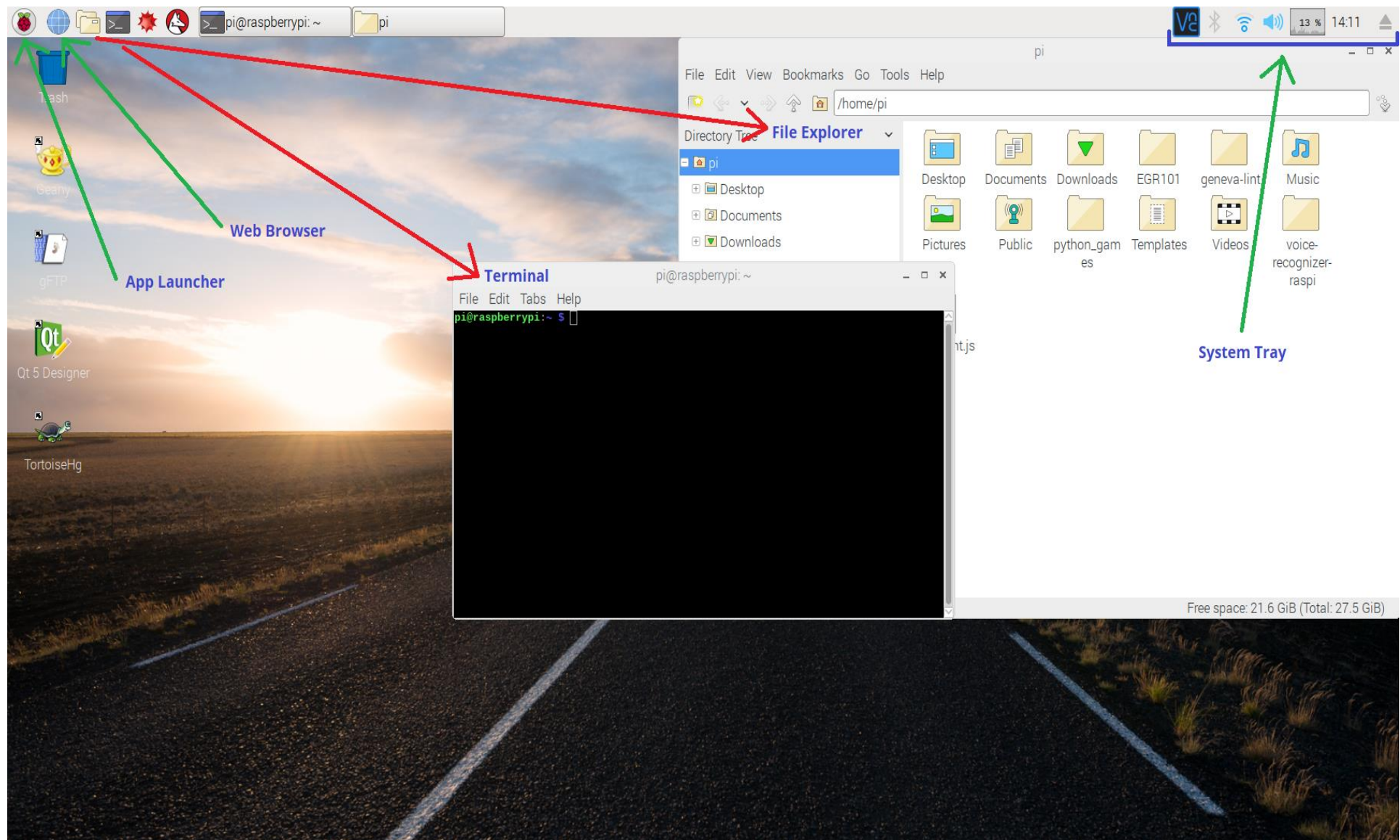
## Introduction to Raspbian

On the next page there is a picture of the Raspberry Pi desktop. To launch apps, you can either use the shortcuts on the desktop or select them from the **app launcher**. To open the app launcher, click on the raspberry button at the top left corner of the screen. The programs you will need to use for this project can be found in the Programming section of the app launcher.

---


<sup>1</sup> If you need a different installer for VNC Viewer, go to <https://www.realvnc.com/en/connect/download/viewer/>

# Raspberry Pi Desktop



## Controlling Electronics with GPIO in Python

The Raspberry Pi has 40 GPIO (General-Purpose Input/Output) pins, to which you can connect electrical components to build circuits. 14 of these pins are reserved for special uses, and the other 26 are for general purpose, as shown in the diagram below. The 3.3V and 5V pins provide a source of constant power, while the pins numbered GPIO1-26 are off by default and must be programmed to turn on.

| Raspberry Pi 3 GPIO Header |                       |  |                       |      |
|----------------------------|-----------------------|--|-----------------------|------|
| Pin#                       | NAME                  |  | NAME                  | Pin# |
| 01                         | 3.3v DC Power         |  | DC Power 5v           | 02   |
| 03                         | GPIO02 (SDA1, I2C)    |  | DC Power 5v           | 04   |
| 05                         | GPIO03 (SCL1, I2C)    |  | Ground                | 06   |
| 07                         | GPIO04 (GPIO_GCLK)    |  | (TXD0) GPIO14         | 08   |
| 09                         | Ground                |  | (RXD0) GPIO15         | 10   |
| 11                         | GPIO17 (GPIO_GEN0)    |  | (GPIO_GEN1) GPIO18    | 12   |
| 13                         | GPIO27 (GPIO_GEN2)    |  | Ground                | 14   |
| 15                         | GPIO22 (GPIO_GEN3)    |  | (GPIO_GEN4) GPIO23    | 16   |
| 17                         | 3.3v DC Power         |  | (GPIO_GEN5) GPIO24    | 18   |
| 19                         | GPIO10 (SPI_MOSI)     |  | Ground                | 20   |
| 21                         | GPIO09 (SPI_MISO)     |  | (GPIO_GEN6) GPIO25    | 22   |
| 23                         | GPIO11 (SPI_CLK)      |  | (SPI_CE0_N) GPIO08    | 24   |
| 25                         | Ground                |  | (SPI_CE1_N) GPIO07    | 26   |
| 27                         | ID_SD (I2C ID EEPROM) |  | (I2C ID EEPROM) ID_SC | 28   |
| 29                         | GPIO05                |  | Ground                | 30   |
| 31                         | GPIO06                |  | GPIO12                | 32   |
| 33                         | GPIO13                |  | Ground                | 34   |
| 35                         | GPIO19                |  | GPIO16                | 36   |
| 37                         | GPIO26                |  | GPIO20                | 38   |
| 39                         | Ground                |  | GPIO21                | 40   |

Rev. 2  
29/02/2016

www.element14.com/RaspberryPi

### GPIO Safety Precautions

- Check with an instructor before connecting anything to the pins labeled 5V. The 3.3V pins are safe but the 5V pins connect directly to wall power and have nothing to limit their current.
- Never connect an LED directly to a GPIO pin without a resistor. 3.3V is too high for most LEDs, which can be fixed by adding a resistor in series that is a few hundred ohms (220 is a good value).

LEDs and motors are examples of an output devices because the Pi outputs voltage to them and doesn't receive any data from them. Buttons, switches, and sensors are examples of input devices because the Pi reads the voltage of them. Our first Python program on the Pi will demonstrate how to read signals from a button.

To write Python code on the Pi, we will use a program called Thonny. It is listed in the Programming section of the app launcher as "Thonny Python IDE" (IDE stands for Integrated Development Environment). The interface of Thonny is similar to the repl.it website which was used in the previous lecture. The large box in the center of the window is where you write your code and the tabs at the top allow you have multiple files open at once. At the bottom of the window is the *shell*, which is where the output of your code will appear when you run it.

Launch Thonny and type the following Python code into an empty file (you don't need to type the comments):

```
import RPi.GPIO as GPIO # Import module with a different name
import time
GPIO.setwarnings(False) # Disable unnecessary warnings
GPIO.setmode(GPIO.BCM) # Indicate numbering scheme

button_pin = 20 # Pin number as labeled on the RasPiO Pro Hat
GPIO.setup(button_pin, GPIO.IN, pull_up_down=GPIO.PUD_DOWN) # Default to ground

while True:
    state = GPIO.input(button_pin) # True - 3.3V wire, False - Ground wire
    print(state)
    time.sleep(0.01) # Wait a small amount of time to avoid bouncing
```

Notice that this code imports the `RPi.GPIO` module which controls the GPIO pins on the Pi, and configures it using the functions `GPIO.setmode` and `GPIO.setwarnings`. Anytime you use the GPIO module you'll want to

call the same two functions at the top of your file. Next it configures GPIO pin #20 as an input, and loops forever to print out whether the button connected to this GPIO pin is pressed.

In order for this to work you'll need to connect a button to your Pi. Insert one into your breadboard and press it in firmly to make sure the pins are connected well. Connect a wire from one side of the button to GPIO20, and from the other side to 3.3V as shown in the picture. In Thonny, save the file you have created by clicking the green save button or pressing Ctrl+S. Then run the code by clicking the play button and you should see the output in the shell change from "0" to "1" while you press the button. Running your program saves it automatically.

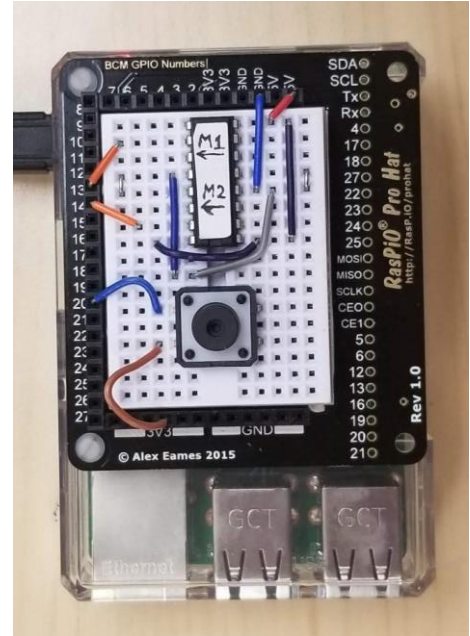
When the button is pressed there is an electrical connection between 3.3V and GPIO20, which the Pi reads as a "1". If GPIO20 is connected to GND (the ground pin), then the Pi will read "0". However, when the button is not pressed then GPIO20 has no electrical connection at all, so how does the Pi know what signal to read? It doesn't, but the argument `pull_up_down=GPIO.PUD_DOWN` that is passed to `GPIO.setup` solves this problem. It tells the Pi that when nothing is connected, it should *pull down* to ground and always read "0". Otherwise the Pi would see random voltages and the input would float back and forth between 0 and 1.

Now change the script from only reading "1" while the button is pressed to toggling between "False" and "True" whenever the button is pressed. The code below only toggles if the button is currently pressed and the last time the button was checked it was not pressed. After making this change you should see much fewer lines of output, switching between "True" and "False" each time the button is pressed.

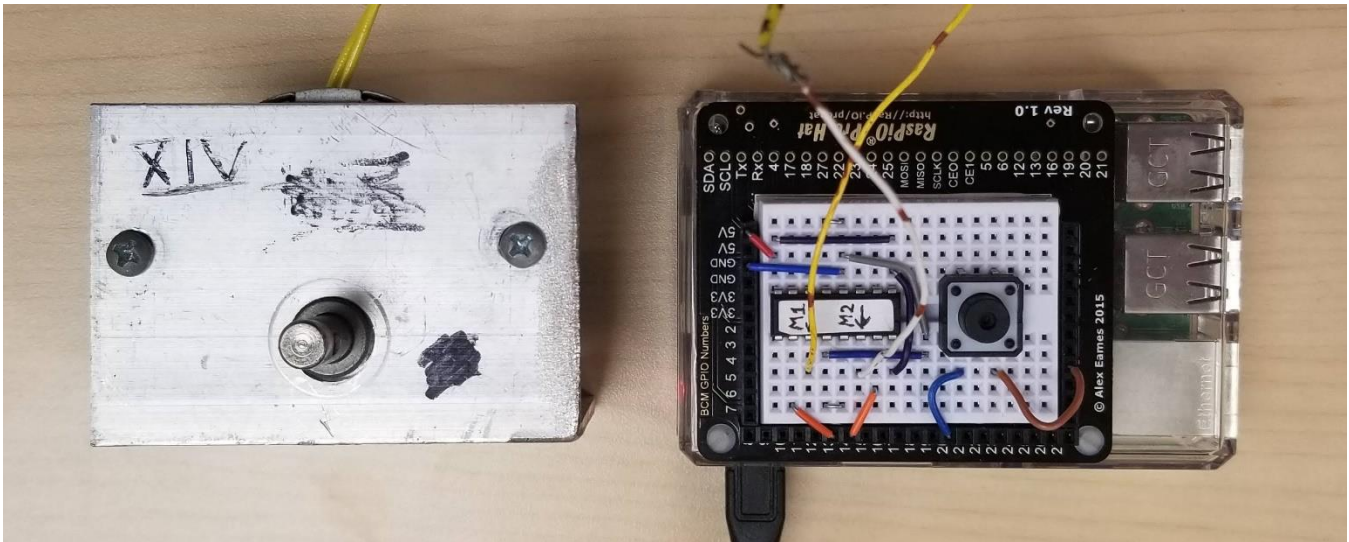
```
...
toggled = False # When button is pressed, this will flip
previous_state = False # State of button when last checked
while True:
    state = GPIO.input(button_pin)
    if (previous_state == False and state == True): # Button has been pressed
        toggled = not toggled # Flip
        print(toggled)
    previous_state = state # Update last state
    time.sleep(0.01)
```

What if we could do something exciting when the button is pressed like turning a motor on and off? Well we can, using the L293D motor control chip that is already set up on your breadboard. This chip makes it easy to turn a motor on and off and control its direction using GPIO pins, while safely running it off the 5V pin which makes the motor faster than if it only ran off 3.3V. The way that the motor control circuit is set up, you can use GPIO pins 13 and 14 on your Pi to control a motor, which you should have in your project bin. Connect your motor to the outputs M1 and M2, as shown in the picture on the next page.

When pins 13 and 14 have identical outputs (both false or both true) the motor will not run, and when the outputs are different (one true and the other false) the motor will turn on. The direction that the motor spins depends on which pin is on and which pin is off. In your code, setup GPIO pins #13 and #14 as output pins and use the `GPIO.output` function to change their value. Now the motor should toggle between on and off when you press the button.







```
...
button_pin = 20
motor_pin_1 = 13 # First of two pins connected to motor control chip
motor_pin_2 = 14 # Second of two pins connected to motor control chip
GPIO.setup(button_pin, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
GPIO.setup(motor_pin_1, GPIO.OUT)
GPIO.setup(motor_pin_2, GPIO.OUT)

toggled = False
previous_state = False
while True:
    state = GPIO.input(button_pin)
    if (previous_state == False and state == True):
        toggled = not toggled
        GPIO.output(motor_pin_1, False) # Always keep first motor pin off
        GPIO.output(motor_pin_2, toggled) # Turn second motor pin on or off
    previous_state = state
    time.sleep(0.01)
```

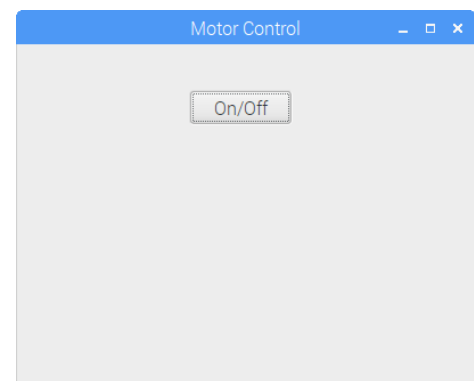
## Creating and Running a Graphical Interface in Python

To create a graphical user interface (GUI) on the Pi, we are going to use a toolkit called Qt. We will build a GUI for our motor control program and add features to it incrementally.

The Qt Designer program has already been installed on your Pi, and there should be a shortcut on the desktop to conveniently access it. When you run it, the “New Form” dialog will appear. Select the template called “Dialog without Buttons”, then click “Create” to start a new project.

You should now see a panel on the left labeled “Widget Box”, which contains a list of all the widgets (graphical interface components) that you can add to your project. In the center of the window is a window labeled “Dialog – untitled”, which is the start of the interface you will be designing. On the right side of the window there is a panel labeled “Property Editor”. This panel contains a list of properties that you can change to customize how the widgets in your project behave, like specifying a different background color or font size.

Scroll through the list of widgets in the left panel until you see “Push Button” in the category “Buttons”. Click on it and drag it into the



window in the center of the screen. Next, use the Property Editor panel to customize your interface. Select the Push Button widget in your window and change the *objectName* property to “powerButton”. This will be used later on to access the widget from your Python code. Scroll down to find the *text* property and change its value to “On/Off”. Also change the *checkable* property to True (make sure it is checked) to make the button toggleable. Click on a blank area of the window to deselect the Push Button and change the *windowTitle* property to “Motor Control”. You should notice that the button text and the window title have changed.

Once you have added a button to your interface, save the file you have created by clicking the green save button or pressing Ctrl+S. Make sure that you name the file “motor\_control.ui” and save it in the same folder as your Python script. Switch back to Thonny. You only need to add three simple lines of code, and now when you run your Python code you should see your GUI pop up.

```
import gui
import RPi.GPIO as GPIO
...
GPIO.setup(motor_pin_2, GPIO.OUT)

window = gui.create("motor_control.ui") # Load custom GUI from file
window.show() # Show custom GUI as an application

toggled = False
previous_state = False
while True:
    ...
    previous_state = state
    window.update() # Refresh widgets and process events
    time.sleep(0.01)
```

After you close your “Motor Control” window, you can still control the motor with the button on the breadboard. The program is still running because the code inside the `while True` loop runs forever. To force the program to stop when the close button is clicked, we can add a global variable `closed` that is initially False and stop the entire program once it becomes True. In order to set it to True when the close button is clicked, we can define a function that does this and pass the name of the function as an argument to the `gui.create` function. Try running this code before adding the lines that turn the motor off.

```
...
GPIO.setup(motor_pin_2, GPIO.OUT)

closed = False
def on_close():
    global closed # We are changing "closed" value in here, so it must be global
    closed = True # Indicate that the window is closing
    GPIO.output(motor_pin_1, False) # Turn the motor off
    GPIO.output(motor_pin_2, False) # Turn the motor off

window = gui.create("motor_control.ui", on_close) # Pass function to run when window is closed
window.show()

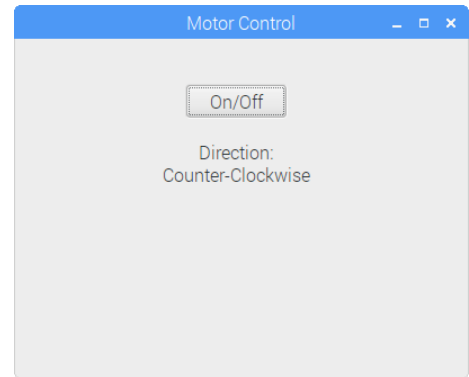
toggled = False
previous_state = False
while not closed: # Stop looping when window is closed
    state = GPIO.input(button_pin)
    ...
```

While your program is running, press the button on the breadboard to turn the motor on. Now if you close your “Motor Control” window, the program quits but the motor still stays on. Adding the lines that turn the motor off in the `on_close` function will fix this problem. Notice that defining `motor_pin_1` and `motor_pin_2` as global variables is not necessary, but defining `closed` as global is. This is because the value of `closed` is changed inside the function, but the motor pin variables stay constant.

So far the button in our window doesn't do anything interesting when you click on it. Let's make it turn the motor on and off, and change the behavior of the physical button to toggle the direction of the motor. The boolean state of `powerButton` can be retrieved using the `isChecked` function. We can use `if` and `else` statements to determine the correct outputs to pins 13 and 14.

```
...
while not closed:
    state = GPIO.input(button_pin)
    if (previous_state == False and state == True):
        toggled = not toggled
    if window.powerButton.isChecked(): # Should motor have power?
        if not toggled: # Should motor turn counter-clockwise?
            GPIO.output(motor_pin_1, False)
            GPIO.output(motor_pin_2, True)
        else: # Should motor turn clockwise?
            GPIO.output(motor_pin_1, True)
            GPIO.output(motor_pin_2, False)
    else: # Should motor not have power?
        GPIO.output(motor_pin_1, False)
        GPIO.output(motor_pin_2, False)
    previous_state = state
    time.sleep(0.01)
```

Switch back to Qt Designer and add two Label widgets to your window. Change the *text* property of the first one to "Direction:" and of the second one to "Counter-Clockwise". Change the *alignment* -> *Vertical* property to "AlignVCenter" for both labels, and set the *objectName* property of the second Label to "directionLabel" so that it can be accessed by that name in the Python code. Save your changes in Qt Designer and switch back to Thonny to write the following code. Now the Label you created should automatically update when you change the direction of the motor by pressing the button on the breadboard.



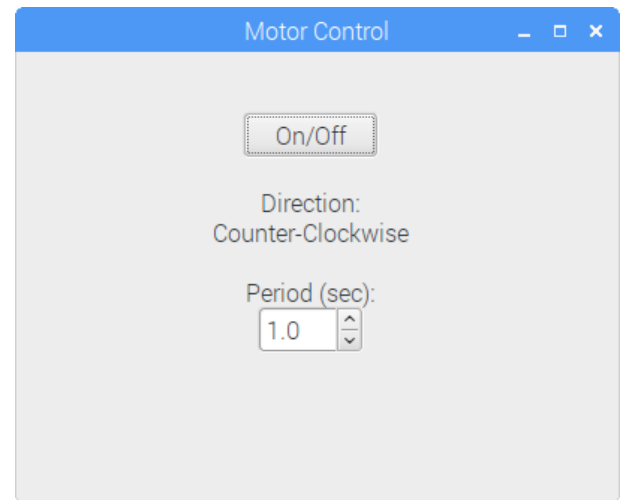
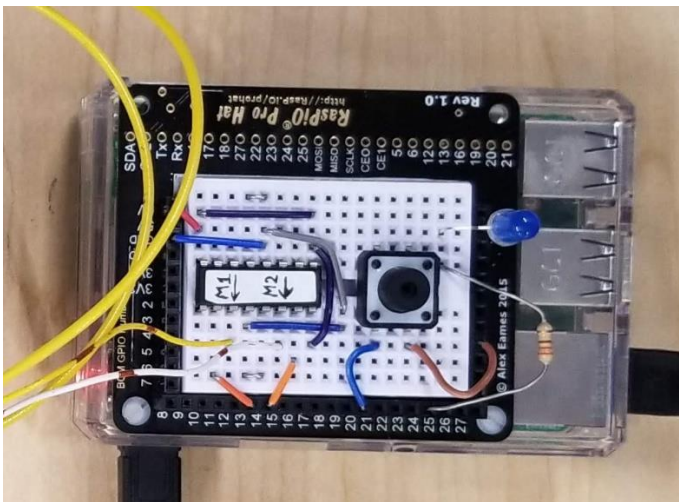
```
...
while not closed:
    state = GPIO.input(button_pin)
    if (previous_state == False and state == True):
        toggled = not toggled
        if toggled: # Display current direction on GUI
            window.directionLabel.setText("Clockwise")
        else:
            window.directionLabel.setText("Counter-Clockwise")
    if window.powerButton.isChecked():
        ...
```

## Handling Events in GUI

We've already seen that if we want to connect some code to an event that happens in the GUI, we can put it inside of a function like `on_close`. But how do we connect a function to run when a button other than the close button is clicked? The `window.connect_event` function can be used for that, which takes the name of the signal emitted by the widget (`clicked` for a button), the name of the function that should be run as the `target` parameter, and an optional list of arguments that should be passed to the function as the `args` parameter. In this case the target is the `print` function and one argument, a string, is being passed into it. Test this code to see that the string gets printed when you click on the "On/Off" button.

```
...
window = gui.create("motor_control.ui", on_close)
# When the GUI button is clicked, the program will do print("powerButton clicked")
window.connect_event(window.powerButton.clicked, target=print, args=["powerButton clicked"])
window.show()
...
```

Let's write our own function that will run when the "On/Off" button is clicked instead of the `print` function. Connect an LED and resistor to your breadboard; The short leg of the LED should be connected to GND, and the resistor should be connected to GPIO24. The LED does not turn on right away because we need to turn on GPIO24 in our program. We can configure the GPIO pin at the top of our program and define the `on_power_button_click` function to turn it on. If we connect this function to the `clicked` signal of the button, it will not run until the button is clicked. Now when you click the button in the GUI, the LED should toggle on and off along with the motor and graphical button.



```
...
motor_pin_2 = 14
led_pin = 24
GPIO.setup(button_pin, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
GPIO.setup(motor_pin_1, GPIO.OUT)
GPIO.setup(motor_pin_2, GPIO.OUT)
GPIO.setup(led_pin, GPIO.OUT)
...
def on_power_button_click(): # Runs when GUI button is clicked
    GPIO.output(led_pin, window.powerButton.isChecked()) # If button is on, light LED

window = gui.create("motor_control.ui", on_close)
# Point to custom function rather than built-in print function
window.connect_event(window.powerButton.clicked, target=on_power_button_click, args=[])
window.show()
...
```

Switch to Qt Designer and add a Double Spin Box widget to your window. Change the *objectName* property to "periodSpinBox", the *decimals* property to "1", the *singleStep* property to "0.1", and the *value* property to "1.0". These properties configure how many decimal points the number has, how much it increases/decreases when you click the up/down arrow, and what the initial value is. It would also be good to add a Label widget next to it with a *text* property of "Period (sec):". Save your changes in Qt Designer and switch back to Thonny.

So far we have not had to be concerned about using threads in our program to multitask because this has been handled automatically. If we want to make the LED blink inside the `on_power_button_click` function, this will require a loop that waits for a certain number of seconds to toggle the value of the GPIO pin controlling the



LED. This would make the function *blocking*, which means that it prevents other code from running, including the GUI which would be frozen as a result. We can fix this problem by running the loop that blinks the LED in a separate thread.

The `threading.Thread` function has `target` and `args` parameters which behave the same way as `window.connect_event` (they indicate the name of a function to run and an optional list of parameters to pass to the function). When the `start` function is called, the `pulse_led` function will run in a separate thread. It contains a `while` loop that runs as long as the motor is on and the window has not been closed. The number of seconds in `periodSpinBox` is used to control the length of the LED pulse cycle. The LED will be on for half the cycle and off for half of it. Run the code after adding threading to it and experiment with changing the blinking speed.

```
import threading
import gui
...
def on_power_button_click():
    threading.Thread(target=pulse_led, args=[]).start() # When button clicked, start thread

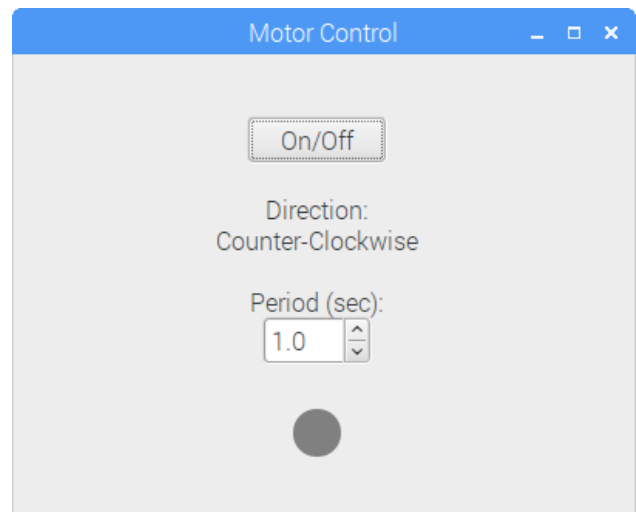
def pulse_led():
    while window.powerButton.isChecked() and not closed: # Stops looping if motor off or window closed
        t = window.periodSpinBox.value() / 2 # Get period in seconds; on and off for 1/2 period each
        GPIO.output(led_pin, True)
        time.sleep(t)
        GPIO.output(led_pin, False)
        time.sleep(t)

window = gui.create("motor_control.ui", on_close)
...
```

In addition to our physical LED, it would be nice to have a virtual LED in the GUI. Switch to Qt Designer and follow the steps on the next page to create an LED Indicator widget. Change the *objectName* property to “virtualLED” and save your changes in Qt Designer. Then switch back to Thonny and add the following lines to toggle your virtual LED alongside the physical one.

### Caution

Setting the period to 0.0 seconds is not recommended, the virtual LED and Spin Box widget will freeze because they can't update fast enough.



```
...
def pulse_led():
    while window.powerButton.isChecked() and not closed:
        t = window.periodSpinBox.value() / 2
        GPIO.output(led_pin, True)
        window.virtualLED.setChecked(True) # Turn on virtual LED
        time.sleep(t)
        GPIO.output(led_pin, False)
        window.virtualLED.setChecked(False) # Turn off virtual LED
        time.sleep(t)
...
```

### Creating an LED Indicator Widget

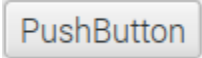
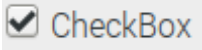
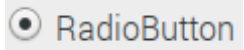




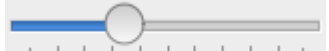


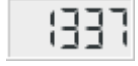
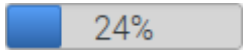
The Qt GUI library lacks an LED Indicator widget to show a Boolean value, but it is simple to create one yourself. Add a Push Button widget to your interface in Qt Designer and set the following properties:

| Name                        | Value                          | Notes   |
|-----------------------------|--------------------------------|---|
| <i>enabled</i>              | False                          | Prevents users from being able to toggle the LED by clicking on it              |
| <i>geometry -&gt; Width</i> | Same as <i>Height</i> property |   |
| <i>text</i>                 | Blank string                   |   |
| <i>checkable</i>            | True                           | Makes the LED toggleable with the <code>setChecked(<i>bool</i>)</code> function |

After changing these properties the button is square and does not yet look like an LED. To customize its appearance, right-click on the button and select “Change styleSheet”. Enter the following text into the “Edit Style Sheet” dialog:

```
QPushButton {  
background-color: gray;  
border-radius: 15px;  
}  
QPushButton:checked {  
background-color: limegreen;  
border-radius: 15px;  
}
```

Setting `border-radius` to the radius of the button widget (half the value of the *Width* property) makes it round. The colors of the LED are set with `background-color` and can be changed to whatever you want.

| Name                           | Appearance  | Description   | Input Functions  | Output Functions   | Signals                          |
|--------------------------------|---|---|--|--|----------------------------------|
| Push Button                    |    | Simple button that emits signal when clicked            | -  | -  | clicked                          |
| Check Box                      |    | Toggles a state between True and False                  | <code>bool isChecked()</code>                          | <code>setChecked(bool)</code>                            | <code>toggled(bool)</code>       |
| Radio Button                   |    | Same as checkbox, but only one can be checked at a time | <code>bool isChecked()</code>                          | <code>setChecked(bool)</code>                            | <code>toggled(bool)</code>       |
| Line Edit                      |    | Single line input for text                              | <code>string text()</code>                             | <code>setText(string)</code>                             | <code>textChanged(string)</code> |
| Spin Box                       |    | Input for integers                                      | <code>int value()</code>                               | <code>setValue(int)</code>                               | <code>valueChanged(int)</code>   |
| Double Spin Box                |    | Input for floats  | <code>float value()</code>                             | <code>setValue(float)</code>                             | <code>valueChanged(float)</code> |
| Dial                           |    | Rotating input for integers                             | <code>int value()</code>                               | <code>setValue(int)</code>                               | <code>valueChanged(int)</code>   |
| Horizontal/<br>Vertical Slider |   | Sliding input for integers                              | <code>int value()</code>                               | <code>setValue(int)</code>                               | <code>valueChanged(int)</code>   |
| Label                          |  | Displays one or more lines of text                      | <code>string text()</code>                             | <code>setText(string)</code>                             | -                                |
| LED Indicator*                 |  | Changes color depending on Boolean value                | <code>bool isChecked()</code>                          | <code>setChecked(bool)</code>                            | -                                |
| LCD Number                     |  | Displays an integer or a float                          | <code>int value()</code><br><code>float value()</code> | <code>display(int)</code><br><code>display(float)</code> | -                                |
| Progress Bar                   |  | Displays a percentage                                   | <code>int value()</code>                               | <code>setValue(int)</code>                               | -                                |

\* The LED Indicator widget does not exist out of the box in Qt Designer. Refer to the previous page for how to create one.