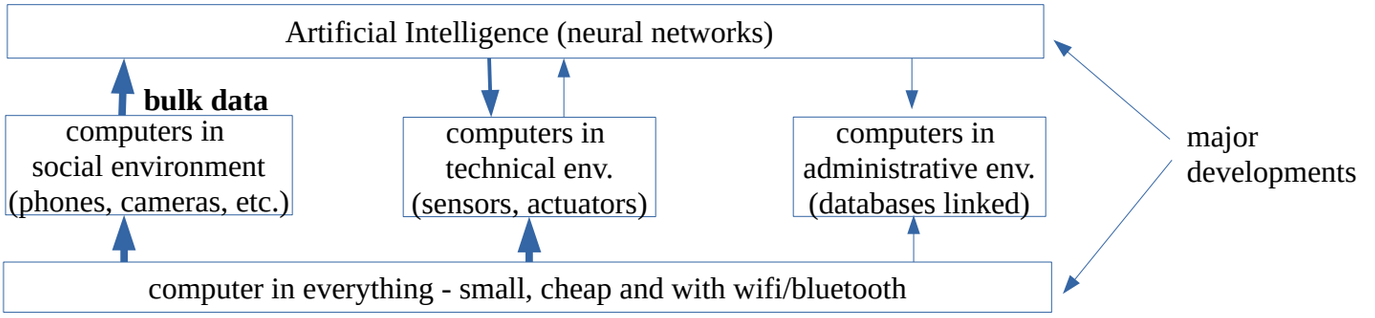
Handout Workshop *Raspberry Pi in a technical environment*
by Chris Hendriks (hendrikschris@yahoo.com)

1. Introduction – major computer (r)evolutions

- *Business* Computer; halfway the 20th century: mini and mainframe – some with private network
- *Personal* Computer; end 20th century: desktop, notebook, tablet, phone – all connected to internet
- *Thing* Computer (other names: SBC – single-board computer, controller, cyber-physical systems, embedded systems); 2010 onwards: computers in devices – all connected to internet (*IoT – internet of things*)

The next major revolution will be: AI (Artificial Intelligence) for evaluating large data sets → later AI in every device.
AI was boosted by the (valuable) bulk data produced in the social environment (apps on phone or tablet, increased web surfing, cameras everywhere). Bulk data could only be handled by AI software. Availability of AI is now affecting the structure of control systems in the technical environment. Initiatives in the area of linked administrative databases are following.



For the technical environment this means: control and intelligence moves into every piece of equipment and every sensor/actuators. Everything communicates with everything. A controller (for sensor or actuator) in the development phase needs a human interface (terminal/keyboard or notebook) but in the operational phase the controller consists of a few chips communicating with 'the world' through wifi or bluetooth.
The explosive growth in the number of computers and number of internet connections results in a drop in price ($10 - $50) and an increase in availability of software. But: security becomes a major issue.

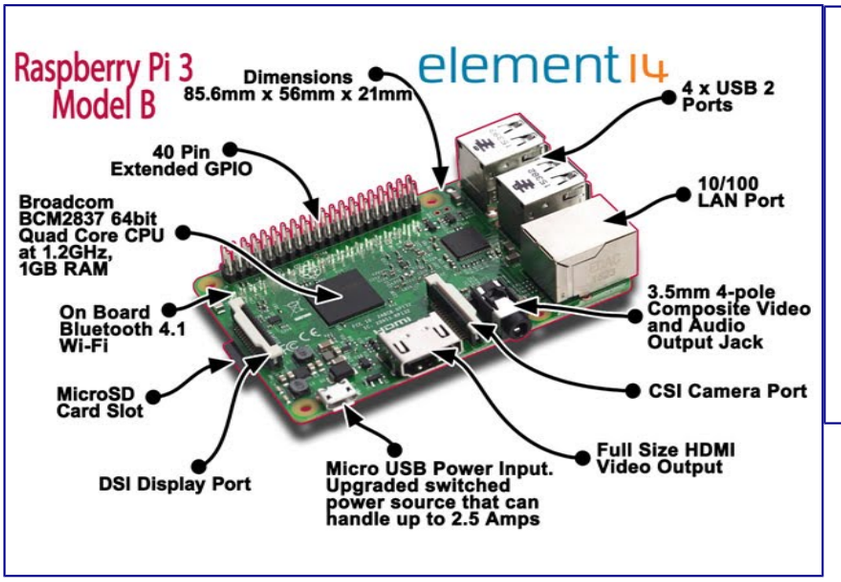
For research this development has a similar impact: every component is controlled by a computer and bulk data is produced (as main output or side product). There will be a transition from isolated measurements to a 5 dimensional data space (in a single experiment multiple parameters are measured at multiple points in space and at multiple moments in time).

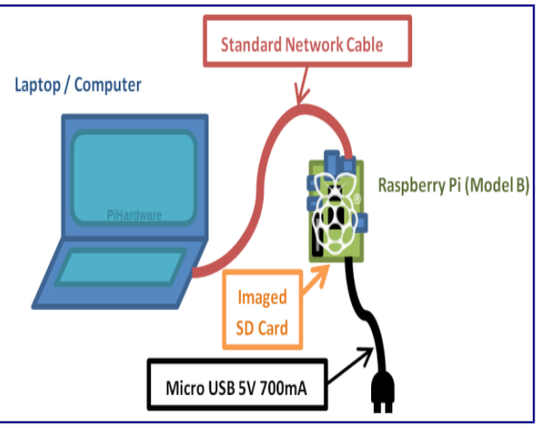
2. Raspberry Pi – single-board computer

The Raspberry Pi is a small computer. It consists of hardware and software.
The software can vary. The Operating System is from the Linux family. When opening the (virtual) terminal the regular Linux commands can be executed. For convenience we usually work from an IDE (integrated development environment) or IDLE (integrated development and learning environment). An IDE allows you to write and run a program. We will use IDE's with Python as its programming language. Python is a programming language from the C++ family enriched with I/O features for connection to physical systems. Usually the Raspberry is connected to a terminal or laptop through a network cable.

In a more advanced structure a general network is used (one or more Raspberries connected through a router connected to the internet). Instead of the network cables a Wifi network can be used. It has advantages to have the raspberry close to the sensor/actuator since usually the cabling between the Raspberry and the plant is more

complex than the hardware for wireless communication. The Raspberry will need a local power supply – however, local power is needed anyway for the sensors and actuators.
For the initial programming of the Raspberry a monitor plus keyboard is convenient – although not absolutely necessary.





3. IDE
For programming the Raspberry we use Python, an Object Oriented Programming (OOP) language.
The easiest introduction to Python is through an IDE (Integrated Development Environment). Through the raspberry icon and selecting 'programming' you can choose an IDE (Python 3 IDLE or Thonny).
The IDE gives you a REPL (Read-Evaluate-Print-Loop) which is a prompt for entering Python commands. As it's a REPL you even get the output of commands printed to the screen without using print. We can use this REPL to, among many other things, interpret variables or do math. For example:

```
>>> 1 + 2
3
>>> name = "Sarah"
>>> "Hello " + name
'Hello Sarah'
```

Alternatively you can write a program with the text editor and run (or debug) it afterwards.

4. Python programs in IDLE
To create a Python program (which is simply a text file) in the Python IDLE, click File > New File and you'll be given a blank window (in Thonny the blank window is already there). This is an empty file, not a Python prompt. You write a Python file in this window, save it, then run it and you'll see the output in the other window.

For example, in the text window, type:

```
n = 3
m = 16
print("The sum is ",n+m)
```

Then save this file (File > Save) and run (Run > Run Module) and you'll see the output in your original Python window. You can also choose *Debug* to have it checked for errors (bugs) first.

5. Basic Python code
Indentation

Python uses indentation to show that code belongs to an earlier statement (nesting). Use the **TAB** key to get the proper indentation. For example a for loop in Python:

```
for i in range(10):          # i starts at 0 and is incremented every loop until i=9
    print("Hello")           # this print loop is executed with i=9 for the last time
     print(i)
```

Also check what happens when you leave out one or both of the indentation.

Variables

To save a value to a variable, assign it like this:

```
name = "Bob"      # the type str (string) is assigned automatically
age = 15          # the type int (integer) is assigned automatically
print (name,"is",age,"years old")
```

Comments

Comments are ignored in the program but are there for you to leave notes. They are denoted by the hash # symbol. Multi-line comments can also use triple quotes like this:

```
"""
This is a very simple Python program that prints "Hello".
That's all it does.
"""
```

If statements

You can use if statements for control flow:

```
name = "Joe"      # or choose any other name
if len(name) > 3: # 'len' is a standard function returning the length of a text variable
    print("Nice name,",name)
else:
    print("That's a short name,",name)
```

While statement

The while statement also controls the program flow (it allows creating a loop like the 'for'-loop):

```
counter = 10
while (counter>0):
    print("hi")
    counter=counter-1
```

will print 'hi' ten times.

Functions

The syntax of a function is:

def functionname(parameters):
  # function documentation
  function_suite
  return [expression]

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call the function printme():

```
def printme(str):
   # This prints a string passed into this function as a parameter
   print(str)
   return;
printme("I'm first call to user defined function!")  # Now you call printme
printme("Again second call to the same function")    # you call printme again
```

When the above code is executed, it produces the following result:

I'm first call to user defined function!

Again second call to the same function

You can return a value from a function as follows:

```
# Function definition is here
def sum(arg1, arg2):          # Add both the parameters and return them.
   total = arg1 + arg2
   print ("Inside the function : ", total)
   return total;
# Now you can call sum function
result_addition = sum( 10, 20 );
print ("Outside the function : ", result_addition)
```

When the above code is executed, it produces the following result −
Inside the function : 30
Outside the function : 30

Exception handling
Assuming we want to ask the user to enter an integer number. If we use *input()*, the input will be a string, which we have to cast into an integer. If the input has not been a valid integer, we will generate (raise) a ValueError. We show this in the following interactive session:

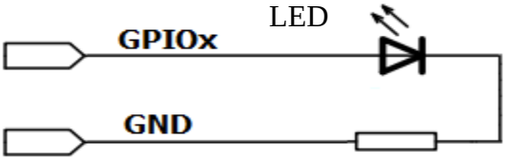
```
>>> n = int(input("Please enter a number: "))
Please enter a number: 23.5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '23.5'
```

With the aid of exception handling, we can write robust code for reading an integer from *input*:

```
while True:
    try:
        n = input("Please enter an integer: ")
        n = int(n)
        break
    except ValueError:
        print("No valid integer! Please try again ...")
print ("Great, you successfully entered an integer!")
```

It's a loop, which breaks only, if a valid integer has been given.
The example script works like this:
The while loop is entered. The code within the try clause will be executed statement by statement. If no exception occurs during the execution, the execution will reach the break statement and the while loop will be left. If an exception occurs, i.e. in the casting of n, the rest of the try block will be skipped and the except clause will be executed. The raised error, in our case a ValueError, has to match one of the names after except. In our example only one, i.e. "ValueError:". After having printed the text of the print statement in the *except*-block, the execution does another loop. It starts with a new i*nput()*. The loop continues indefinitely until the *break* is executed.

6. Switching a LED using the Raspberry PI



This experiment demonstrates how to attach a LED to the expansion connector on your Raspberry PI and to make it blink with a simple Python program.

In order to switch a LED on and off programmatically we need to connect it between a general-purpose input/output pin (GPIO pin) and the ground. A resistor is necessary to limit the current.

Build the following circuit
- the long pin of the LED is connected to a GPIO pin (the long pin needs a high voltage for the LED to conduct); you can choose any GPIO input/output pin – let's take pin 3 (see appendix 1: overview 'GPIO pin numbering');
- the short pin of the LED is connected to ground through a resistor (any resistor between 300 and 500 Ohm will do).

Now we need to make the GPIO pin an output and change the state of the pin between 1 and 0 to switch the LED on and off. Type the following code into the text window (you may leave out the comments; see also *Appendix 2 GPIO and other libraries*):

```
import RPi.GPIO as GPIO       # Import GPIO library
GPIO.setmode(GPIO.BOARD)      # Use board pin numbering
GPIO.setup(3,GPIO.OUT)        # Setup GPIO Pin 3 to OUT
GPIO.output(3,True)           # Turn on GPIO pin 3
```

Run your program. You should see your LED light up (notice that even when the output is not 'True' the LED might shine slightly; this is because of the internal pull-up resistor of the port).
We just told the RPi to supply voltage (+3.3v) to our circuit using GPIO pin 3. Change the program by making pin 3 False (= 0 Volt) and run the program again.
A warning was given since the GPIO port was still in use . This can be avoided by adding GPIO.cleanup() at the end of your program.

7. Blinking Light

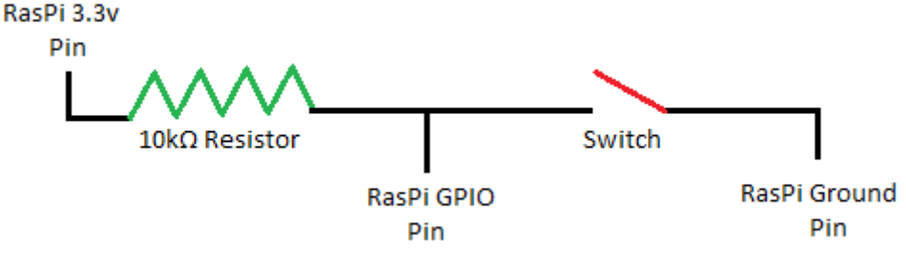
Here is a slightly more advanced script that blinks the led on and off. The only real difference is that we are gathering user input and using the sleep function to set the amount of time the light is on or off.
Type the following code into the window (the comments are optional):

```
import RPi.GPIO as GPIO              # Import GPIO library
import time                          # Import 'time' library.
GPIO.setmode(GPIO.BOARD)             # Use board pin numbering
GPIO.setup(3, GPIO.OUT)              # Setup GPIO Pin 3 to OUT

# Define a function named Blink()
def Blink(numTimes,timeOn,timeOff):
      for i in range(0,numTimes):              # Run loop numTimes
            print ("Iteration ",str(i+1)) # Print current loop
            GPIO.output(3,True)                # Switch on pin 3
            time.sleep(timeOn)                 # Wait
            GPIO.output(3,False)               # Switch off pin 3
            time.sleep(timeOff)                # Wait
      print ("Cycle done")

# read total number of blinks + length of on and off time
iterations = input("Enter total number of times to blink: ")
timeOn = input("Enter time LED is on(seconds): ")
timeOff = input("Enter time LED is off(seconds): ")
# Start Blink() function. Convert user input from strings to
# numeric data types and pass to Blink() as parameters
Blink(int(iterations),float(timeOn),float(timeOff))
GPIO.cleanup()
```

Adjust the program in such a way that after a complete cycle of going on and off the given number of times, there follows another complete cycle with a period that is twice as long.

8. Switch

Use a switch to build the following circuit. Use GPIO pin 3 (although any other would do as well).



Use the following code to read the input:

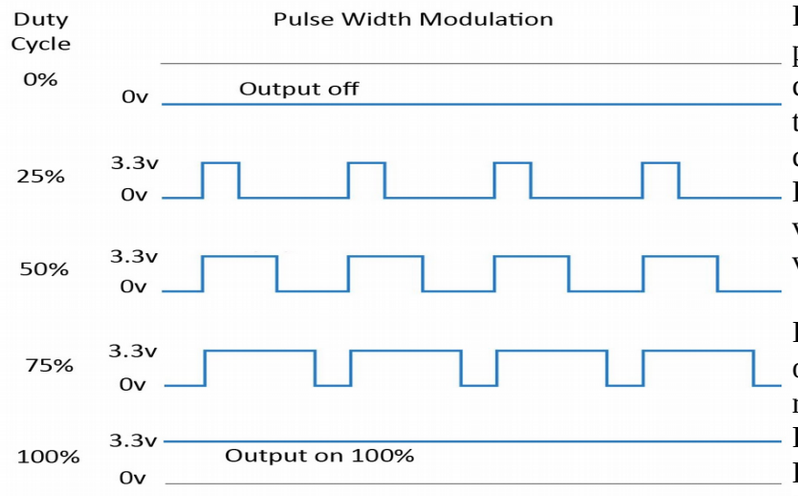
```
import RPi.GPIO as G
import time as t
G.setmode(G.BOARD)                  # board numbering
G.setup(3,G.IN)                     # makes 3 an input
try:
      while (True):
            print(G.input(3))       # prints port 3
            t.sleep(1)
except KeyboardInterrupt:           # break loop with CTRL-C
      print ('All done')
G.cleanup()
```

Write a program that makes the LED go on when the switch is on and go off when the switch is off.

9. Using a photoresistor as a light sensor

The photoresistor we are using has a resistance of 0.2 Mohm in dark and 2 to 5 Kohm in light (depending on the brightness of the light). To show the effect of light we built the following circuit: from 0 V to the photoresistor and from the other side of the photoresistor to port 40 (we use port 40 because this one does not have an internal pull-up or pull-down). From this port also through a 10 Kohm resistor to 3.3 V. The light sensor acts as a switch: it has a high resistance when in the dark and a lower resistance when in the light. Run the same program as in exercise 8 (port 40 is programmed input). Light will make the input low.
How can you adjust the device so that it has a hysteresis (the dark → light transition is at a higher light intensity as compared to the light → dark transition)?

10. PWM (pulse width modulation)



PWM is used in many systems. It means that a periodic signal is high for a a certain part of the cycle and low for the rest of it. The percentage of time that the signal is high is called the duty cycle.
By having a frequency of 50Hz or more and varying the duty cycle the average voltage is varied so we can use PWM to dim a light.

In the following example we use pin 3 as an output. The output is connected through a resistor of 330 Ohm and a LED to ground.
PWM is available through the GPIO library.
Let's take a PWM signal at the output port 3 of 1 Hz. We create a PWM instance (object) on pin 3 with a 1 Hz signal with the GPIO command (see Appendix 3) `pwm=GPIO.PWM(3,1)` whereby pwm is an arbitrary name. With pwm.start and pwm.ChangeDutyCycle we can set and change the duty cycle (i.e the percentage of time the LED is on and off). Lets start with a duty cycle of 25 %:

```
import RPi.GPIO as G
import time as t
G.setmode(G.BOARD)
G.setup(3,G.OUT)        # pin 3 is connected to the LED through a resistor
pwm=G.PWM(3,1)
pwm.start(25)
t.sleep(10)
pwm.stop()
G.cleanup()
```

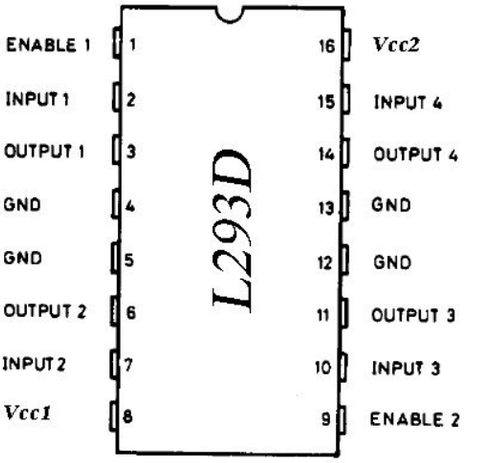
See what happens if we change the duty cycle to 60 % after 10 sec. using the command `pwm.ChangeDutyCycle(60)` and including another sleep period.
Try other values for the duty cycle.
Write a program that makes the light slowly go brighter and then slowly dim again by taking a PWM signal of 50 Hz and varying the duty cycle.

Measure with the Volt-Amperemeter the power taken by the LED+ resistor.

Replace the LED with a small speaker and give output 3 a PWM signal with a frequency of 50 Hz and a duty cycle of 50%. What happens if you change the frequency keeping the duty cycle the same (use ChangeFrequency(freq) to change the frequency)?

A. The Raspberry Pi and mechanics

A1. Controlling a dc motor



The Raspberry can not supply the current for running a motor. Therefore we use an separate 5 V power supply and the L293D to control the motor (the L293D can control two motors in direction and speed; we will however use it for only one motor, running in one direction and with a constant speed). We make the following connections to the L293D
1 to 3.3 V of Raspberry;
2 to common GND (Raspberry and motor);
3 and 6 go to the motor (3 to black and 6 to red);
4 and 5 to common GND;
7 to the output port (eg port 3) of Raspberry
8 and 16 go to the separate power supply
This time we will enter commands in the Python shell (although you could also run the commands as a program of course).
Enter:

```
import RPi.GPIO as GPIO
GPIO.setmode (GPIO.BOARD)
GPIO.setup(3,GPIO.OUT)
GPIO.output(3,True)
```

The motor starts running now.
As soon as you enter `GPIO.output(3,False)` the motor stops again.
Write a program that starts the motor when you enter 'go' and stops the motor after 5 seconds.

A2. Moving a robot arm

The 3 DOF (degrees of freedom) robot arm uses 3 servo motors.
The position of the servo motor is set by a PWM signal based on a frequency of 50 Hz. With a duty cycle of 5 the servo angle will be minimum, if the duty cycle is 7.5 the servo will be at its center position (90 degrees) and if it is 10 it will be at its maximum.

Let us take an accuracy of 3.6 degrees. A change in position of 3.6 degrees corresponds with a change in pulse width of 0.02 ms which is equal to a change in duty cycle of 0.1.
Connect any servo (red: positive of external power supply; black: ground of external power supply and Raspberry; yellow: PWM input signal). We will use port 3 of the Raspberry for the PWM signal.
Run the following program:

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BOARD)
GPIO.setup(3,GPIO.OUT)        # pin 3 is connected to the servo
pwm=GPIO.PWM(3,50)            # a 50 Hz PWM signal is being used
pwm.start(5)                  # the servo moves to the 0 degrees position
time.sleep(5)
pwm.ChangeDutyCycle(7.5)      # the servo moves to 90 degrees
time.sleep(5)
pwm.ChangeDutyCycle(10)       # the servo moves to 180 degrees
time.sleep(5)
pwm.stop()                    # the PWM signal is stopped
GPIO.cleanup()
```

Now to make the servo start at the middle position and than move 3 steps of 3.6 degrees to the right we can use the following code lines (include a ‘sleep’ time where necessary):

```
pwm.start(7.5)
pwm.ChangeDutyCycle(7.5+3*0.1)      # 3.6 degrees = 0.1 change duty cycle
```

Write a program that allows a user to make the servo turn first to one direction and than to the other direction. The movement to the respective positions is controlled by repetitive pressing of keys on the keyboard. When another key is hit the servo moves to the start position and makes the same movement automatically (hint: register the key strokes by adding them to a ‘list’ – see appendix 6).

Make the robot arm pick up an object and place it at a predefined position. Use the keyboard to activate the servo’s.

Write a program that registers the movement that the arm makes when controlled by the keyboard. Afterwards the robot does the same job by itself. When you have it working optimize the path of the robot arm by activating two servo’s at the same time and making larger steps where possible.

B. Sensors and local displays

B1. Using an LCD display with parallel input

We will import software that uses the LCD display with the following connections to the GPIO pins (BOARD numbering):
1 - Ground; common to the Raspberry and the separate power supply
2 - VCC; we will use a separate 5V power supply
3 - (contrast) → connected to Ground
4 - RS → 3 Register Select; 0: Command, 1: Data
5 - (read/write select) → connected to Ground
6 - E → 5 Enable data transfer
11 - D4 → 7 databit 0
12 - D5 → 11 databit 1
13 - D6 → 13 databit 2
14 - D7 → 15 databit 3

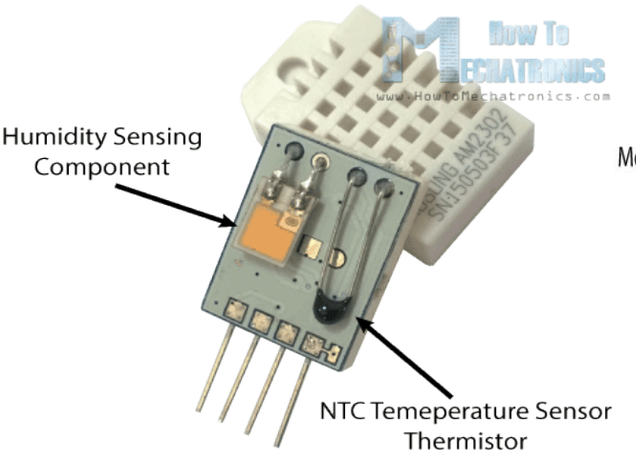
15 (5V) and 16 (0V) are connected to the backlight.
1. After making the connections run 'LCDdisplay.py'.
2. You can use the functions from 'LCDdisplay.py' by importing it in you code. Run the following code in the python shell (open the shell in the folder where the program is found):

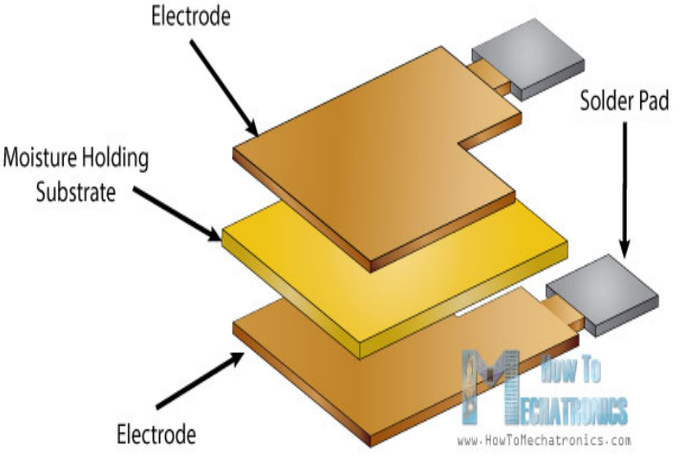
```
>>> import LCDdisplay as lcd
>>> lcd.lcd_init()                                  # the screen becomes blanc
>>> lcd.lcd_string(“-any text-”,0x80)          # writes a text on line 1
>>> lcd.lcd_string(“-any text-”,0xC0)          # writes a text on line 2
```

3. You can also import 'LCDdisplay' in your program and make use of the functions. Write a program that shows some text on the first line and blinks another text on the second line.
4. If time allows use two displays to make a news ticker (news scroll vertically over 2 displays).

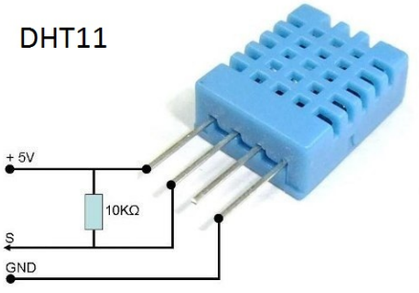
B2 Temperature and humidity sensor

The DHT11 consist of a humidity sensing component, a NTC temperature sensor (or thermistor) and an IC on the back side of the sensor.





For measuring humidity they use the humidity sensing component which has two electrodes with moisture holding substrate between them. So as the humidity changes, the conductivity of the substrate changes or the resistance between these electrodes changes. This change in resistance is measured and processed by the IC which makes it ready to be read by the raspberry. For measuring temperature these sensors use a NTC temperature sensor or a thermistor. A thermistor is actually a variable resistor that changes its resistance with change of the temperature. These sensors are made by sintering of semiconductive materials such as ceramics or polymers in order to provide larger changes in the resistance with just small changes in temperature. The term “NTC” means “Negative Temperature Coefficient”, which means that the resistance decreases with increase of the temperature.



The DHT11 sensor has four pins: (top view) 1 Vcc (= 5 V), 2 data, 3 not connected, 4 GND. A pull-up resistor from of 10K Ohms is required to keep the data line high and in order to enable the communication between the sensor and the Raspberry.
The DHT11 sensors have their own single wire protocol used for transferring the data. This protocol requires precise timing and the timing diagrams for getting the data from the sensors can be found from the datasheets of the sensors. However, we don’t have to worry much about these timing diagrams because we will use the *DHT library* which takes care of everything.

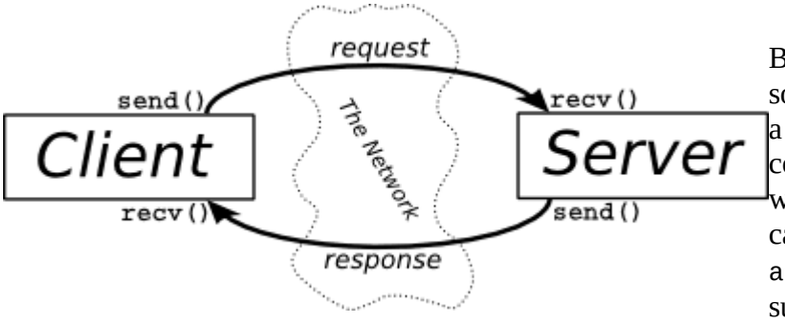
Connect the data output of the DHT11 to pin 40 of the Raspberry (pin 40 = GPIO21; the software we will use uses BMC numbering) and test the following software:

```
import Adafruit_DHT
import RPi.GPIO as GPIO
import time

i=0
while (i<10):
    humidity, temperature = Adafruit_DHT.read_retry(Adafruit_DHT.DHT11, 21)
    #  Adafruit_DHT.DHT11 is the type of sensor; 21 is the BMC GPIO number
    humidity = round(humidity, 2)
    temperature = round(temperature, 2)

    if humidity is not None and temperature is not None:
        print ("Temperature: ",temperature)
        print ("Humidity: ",humidity)
    else:
        print ("No data received")
    time.sleep(3)
    i=i+1
```

See what happens when you breathe over the sensor.
Adjust the program in such a way that temperature and humidity are displayed on the LCD display (hint: to convert the variable ‘temp’ into a string use str(temp); to concatenate two strings use “temperature “+ str(temp)).
If time allows add a second display in such a way that an alarm is given whenever the temperature or the humidity gets over or under a certain value.

C. Communication

The raspberry Pi has on-board Wifi and Bluetooth. Wifi requires a router but allows ‘unlimited’ connection to the rest of the world. Bluetooth has a more local focus. It allows one-to-one communication up to hundred meters at a data rate of 1 to 3 Mbps. The Raspberry Pi uses Bluetooth Low Energy (BLE) which has a slightly lower bandwidth.



Bluetooth programming in Python follows the socket programming model (like IP). In this model, a socket represents an endpoint of a communication channel. Sockets are not connected when they are first created, and are useless until a call to either `connect` (client application) or `accept` (server application) completes successfully. Once a socket is connected, it can be used to send and receive data until the connection fails due to link error or user termination.

The class ‘socket’ is a predefined Python class in the Raspberry Pi. By importing it we have all the communication methods available (for Bluetooth but also for TCP/IP and other protocols). A bluetooth address is represented as a string of the form “xx:xx:xx:xx:xx:xx”, where each x is a hexadecimal character; xx represents one octet of the 48-bit address. Bluetooth devices will always be identified using an address string of this form. Choosing a device really means choosing a bluetooth address. If only the user-friendly name of the target device is known, then two steps must be taken to find the correct address. First, the program must scan for nearby Bluetooth devices. Next, the program uses the routine `lookup_name()` to connect to each detected device, requests its user-friendly name, and compares the result to the target name.

The following examples show how to establish a connection using an RFCOMM socket, transfer some data, and disconnect.

Server:

```
import socket                                   # socket is a predefined class on the Pi
hostMACaddress = 'B8:27:EB:01:D4:6D'
port = 3                                # this is an arbitrary choice
backlog = 1                             # the server listens to 1 client at a time
size = 1024                             # max. size data packets

s = socket.socket(socket.AF_BLUETOOTH, socket.SOCK_STREAM, socket.BTPROTO_RFCOMM)
                        # a socket object is created
                        # socket.AF_BLUETOOTH – (Address Family) bluetooth
                        # socket.SOCK_STREAM – type of connections
                        # socket.BTPROTO_RFCOMM – address is string and port is number
s.bind((hostMACaddress, port))                  # socket object connects to client
s.listen(backlog)                       # socket object listens to 1 client at a time
print("We are waiting for data..")
client, address = s.accept()                    # connection accepted; new object ‘client’
inst.
data = client.recv(size)                        # data received
data = data.decode('UTF-8')                     # decode from UTF-8 to string
print (data)
print("done and closing the socket")
client.close()
s.close()
```

Client:

```
import socket
serverMACAddress = 'B8:27:EB:01:D4:6D'
port = 3
s = socket.socket(socket.AF_BLUETOOTH, socket.SOCK_STREAM, socket.BTPROTO_RFCOMM)
s.connect((serverMACAddress,port))
message = ("a")
s.send(bytes(message, 'UTF-8'))
print("data sent")
s.close()
```

The server-socket that is used to accept incoming connections must be attached to operating system resources with the bind method. ‘bind’ takes in a tuple specifying the address of the local Bluetooth adapter to use and a port number to listen on. Once a socket is bound, a call to `listen` puts the socket into listening mode and it is then ready to accept incoming connections.
The client-socket that is used to establish an outgoing connection connects to its target with the `connect` method, which also takes a tuple specifying an address and port number. In the example above the client tries to connect to the Bluetooth device with address ‘ `B8:27:EB:86:E6:F4’` on port 3.

- Write a client or server program based on Bluetooth sockets to enter a text on the keyboard of one Raspberry Pi system (client) and display it on the screen of the other system (server).
- Modify the programs for client and server in such a way that the server replies the client with some text.
- Modify the programs in such a way that the humidity is measured by the server after a request from the client, sent to the client and displayed on the screen of the client.
- If time allows modify the programs in such a way that one client requests the humidity measurement from two or three servers.

D. Using a camera

First of all, with the Pi switched off, you’ll need to connect the Camera Module to the Raspberry Pi’s camera port, then start up the Pi and ensure the software is enabled (sudo raspi-config; Interfacing Options; Camera).

Enter the following code:

```
from picamera import PiCamera       # import the class PiCamera from the picamera library
import time
camera = PiCamera()                 # an object 'camera' is created
camera.start_preview()              # the lens is opened
time.sleep(2)                       # 2 sec are given for adjusting parameters
camera.capture('/home/pi/Desktop/image.jpg')   # a picture is taken
camera.stop_preview()               # the lens is closed
```

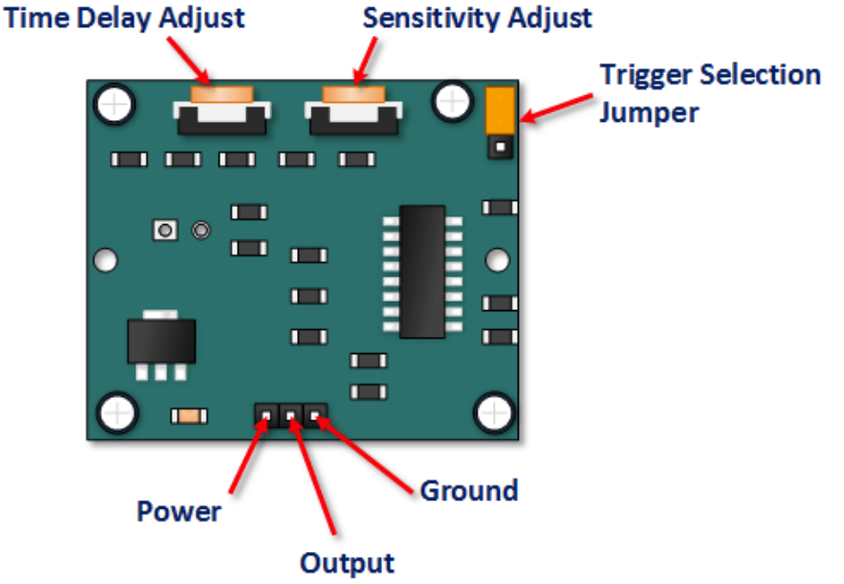
You can rotate the image by `90`, `180`, or `270` degrees by including: camera.rotation = 180 (or any other value). You can view the picture from the (raspberry) desktop, ie the path given in the capture method.

Now try adding a loop to take five pictures in a row:

```
for i in range(5):
      time.sleep(2)
      camera.capture('/home/pi/Desktop/image%s.jpg' % i)
```

The variable `i` contains the current iteration number, from `0` to `4`, so the images will be saved as `image0.jpg`, `image1.jpg`, and so on.

Use the motion detector to have the sequence of pictures start when movement is detected ('power' can be connected to the 5 V of the Raspberry Pi; the output turns high when motion is detected).
Turn 'Time Delay Adjust' - the time the output remains high after detecting motion – fully left (minimum = 3 seconds). After being set to 1 and returned to 0 the output remains 0 for 3 seconds.
Turn 'Sensitivity Adjust' - the distance over which motion is detected – fully left (minimum = 3 meter). Use a GPIO port without a pull up resistor (eg port 40). Test the motion detector separately first.



Run the software first in the polling mode and after that in the interrupt mode (see appendix 5; first test the interrupt software separately).

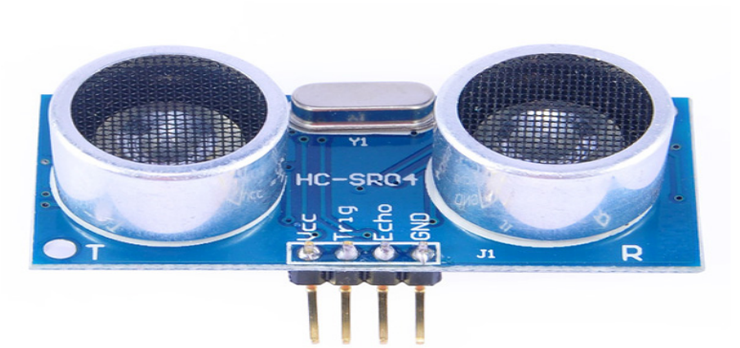
In control situations a picture or video would usually be analyzed with software based on Artificial Intelligence. Deep Learning (often based on Neural Networks) is rapidly gaining popularity.

E. Distance measurement

Use the Raspberry Pi distance sensor (ultrasonic sensor HC-SR04). It sends an ultrasonic pulse and receives it. The timing of this process is received from and passed on to the Raspberry Pi. Apart from the 5 V power supply and ground it uses an output from the Pi (trigger) and supplies an input to the Pi (echo). 'Echo' is a 5 V output from the sensor so a voltage divider (eg 1K and 2K) is to be used (the inputs for the Pi should have a 3.3 V maximum).

The HC-SR04 sensor requires a short 10 µsec pulse to trigger the module, which will cause the sensor to start the ranging program (8 ultrasound bursts at 40 kHz) in order to obtain an echo response. So, to create our trigger pulse, we set the trigger pin high for 10 µsec then set it low again.
Now that we've sent our pulse signal we need to listen to our input pin, which is connected to ECHO. The sensor sets ECHO to high for the amount of time it takes for the pulse to go and come back, so our code therefore needs to measure the amount of time that the ECHO pin stays high. We use the "while" string to ensure that each signal timestamp is recorded in the correct order. The time.time() function will record the latest timestamp for a given condition. For example, if a pin goes from low to high, and we're recording the low condition using the time.time() function, the recorded timestamp will be the latest time at which that pin was low.
Our first step must therefore be to record the last low timestamp for ECHO (pulse_start) e.g. just before the return signal is received and the pin goes high. After that we need the last high timestamp for ECHO (pulse_end).

We will take the speed of sound to be 343 m/s (although it it is variable, depending on what medium it's traveling through, in addition to the temperature of that medium).
We also need to divide our time by two because what we've calculated above is actually the time it takes for the ultrasonic pulse to travel the distance to the object and back again.



$$34300 = \frac{Distance}{Time/2}$$

$$17150 = \frac{Distance}{Time}$$

$$17150 \times Time = Distance$$

```
import RPi.GPIO as GPIO                    # import libraries
import time
GPIO.setmode(GPIO.BOARD)
TRIG = 3                                   # give a name to GPIO Pins
ECHO = 5
GPIO.setup(TRIG,GPIO.OUT)                  # set GPIO direction (IN / OUT)
GPIO.setup(ECHO,GPIO.IN)

print ("Distance Measurement In Progress")
GPIO.output(TRIG, False)
print ("Waiting For Sensor To Settle")
time.sleep(2)
GPIO.output(TRIG, True)                    # a short pulse is given to start the measurement
time.sleep(0.00001)
GPIO.output(TRIG, False)
while GPIO.input(ECHO)==0:                 # the start of the echo pulse is measured
      pulse_start = time.time()
while GPIO.input(ECHO)==1:                 # the end of the echo pulse is measured
      pulse_end = time.time()
pulse_duration = pulse_end - pulse_start # pulse duration is calculated
distance = pulse_duration * 17150          # distance is calculated
distance = round(distance, 2)              # we round the distance in 2 decimal places
print ("Distance:",distance,"cm")
GPIO.cleanup()
```

What is the accuracy of the measurement device? How can the device be calibrated?

F. Threading

Programs, **Processes** and **Threads** are three basic concepts of the operating systems. A *program* is an executable file usually stored on a disk or a secondary memory on your computer. It contains the set of instructions written to perform a specific job on your computer. A *process* is an executing instance of a program. A *thread* is the smallest executable unit of a process. A process can have multiple threads.
In the program that activates multiple threads you need to import the treading module. We will also import time to create delays. To use a variable in multiple threads it must be defined as *global.*

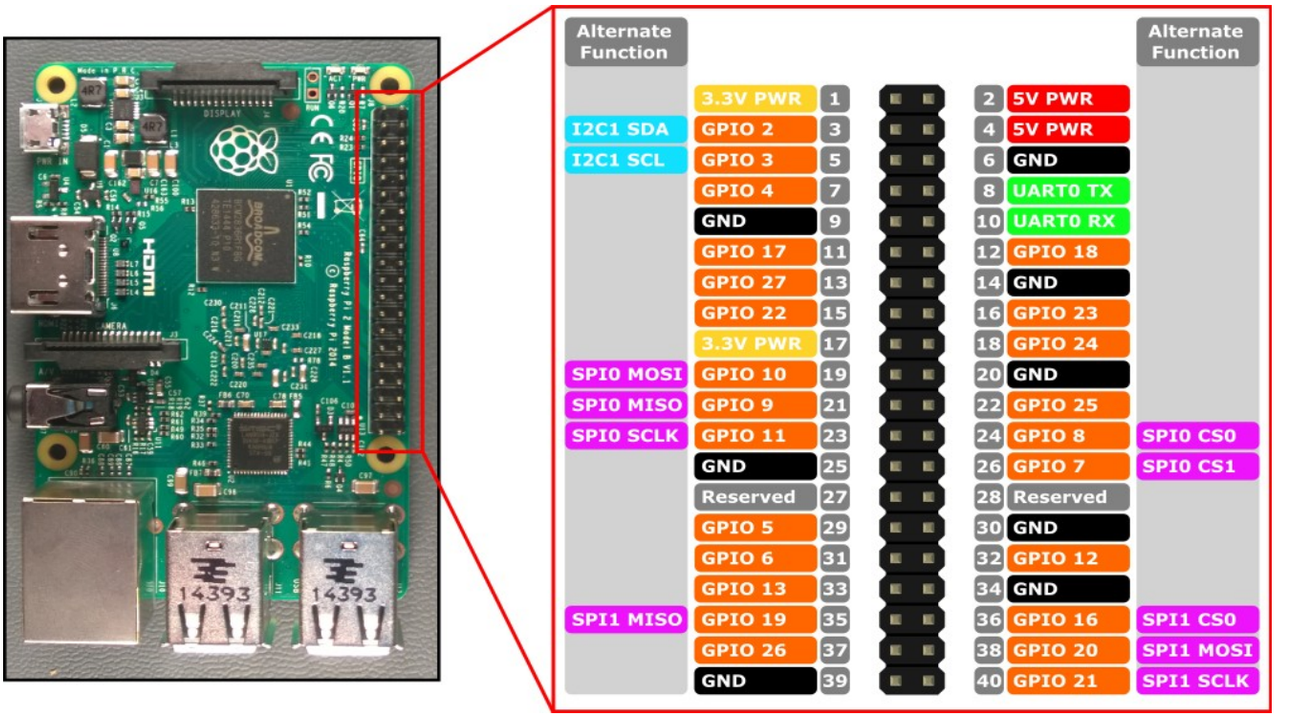
```
from threading import Thread
import time
global cycle                        # cycle is defined as global variable
cycle=0                       # cycle gets the initial value 0
class Hello5Program:                # the class Hello5Program is defined
    def __init__(self):             # the initialization method is defined
        self.thread_running= True
    def terminate(self):                 # the termination method is defined
        self.thread_running = False
        print(" 5 sec thread terminated")
    def run(self):                  # the method that is being executed as main code is
defined
        global cycle                # the same variable as in the mail thread is used
        while (self.thread_running):
            time.sleep(5)
            cycle = cycle + 5
            print ("5 Second Thread cycle + 5: ", cycle)
FiveSecond = Hello5Program()  # create instance of previously defined class
FiveSecondThread = Thread(target=FiveSecond.run)     # create thread
FiveSecondThread.start()            # start thread
Exit = False                        # exit flag
while Exit==False:
    cycle = cycle + 1
    print ("Main Program increases cycle + 1: ", cycle)
    time.sleep(1)                   # one second delay
    if (cycle > 50): Exit = True    # exit program

FiveSecond.terminate()        # terminate the thread by calling the terminate method
time.sleep(6)                       # an delay is included to make sure the
FiveSecondThread has finished
print ("Goodbye :)")
```

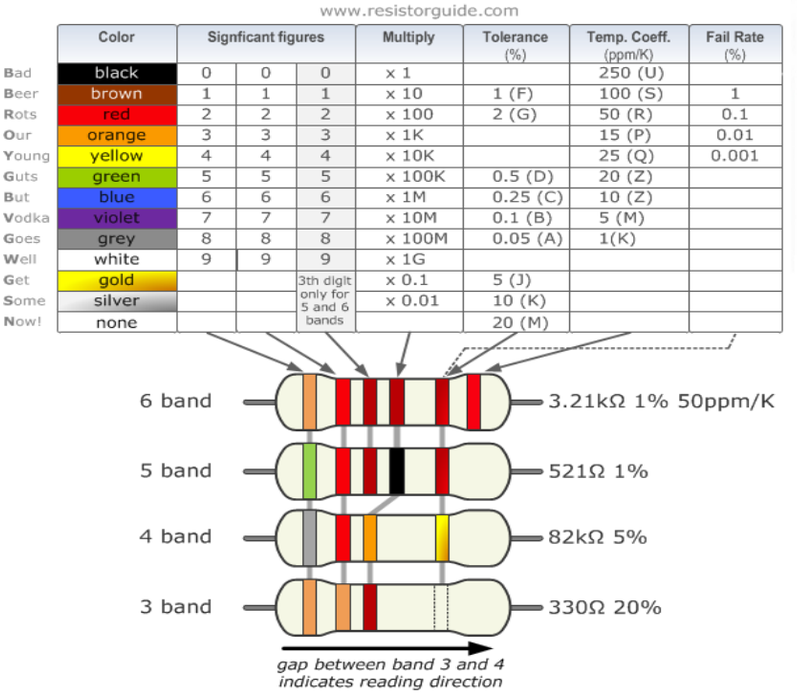
The program output shows the increase of *cycle* by 1 then by 5 when the thread runs. Both threads (main and FiveSecondThread) modify the variable ‘cycle’ and print the value. The extra delay after termination of the FiveSecondThread makes sure that the main program finishes after FiveSecondThread (FiveSecondThread makes a another complete cycle after execution of the terminate method).

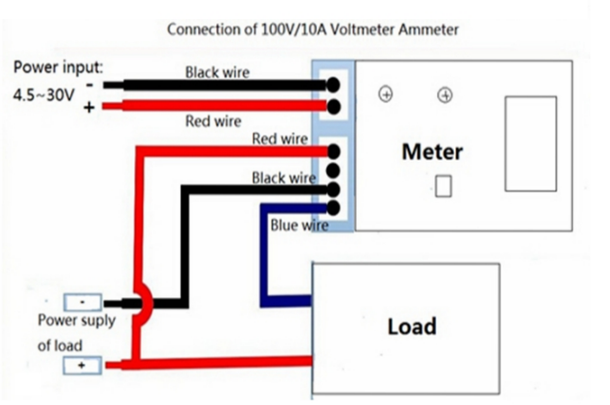
Many threads can be created to do different functions. A thread can call any function in the main program. Sensors can be read, a request from another station through the network can be handled or other actions can be taken while the main program or other threads run at the same time.

Modify the program in such a way that a LED continues blinking while a new value for the frequency can be entered at any time. (hint: the blinking code runs in a separate thread; the main program communicates with the user; a global variable is used for the frequency).

Appendix 1 GPIO pin numbering – Raspberry Pi 3B; resistor colour code; volt-ampere meter







Appendix 2 GPIO and other libraries

To import the RPi.GPIO module:
import RPi.GPIO as GPIO
By doing it this way, you can refer to it as just GPIO (you can use any name) through the rest of your script.
There are two ways of numbering the IO pins on a Raspberry Pi within RPi.GPIO. The first is using the BOARD numbering system. This refers to the pin numbers on the P1 header of the Raspberry Pi board. The other one is BCM which is an internal numbering. We will use BOARD.

To specify which one you are using (mandatory):
GPIO.setmode(GPIO.BOARD)

It is possible that you have more than one script/circuit on the GPIO of your Raspberry Pi. As a result of this, if RPi.GPIO detects that a pin has been configured to something other than the default (input), you get a warning when you try to configure a script. To disable these warnings:
GPIO.setwarnings(False)

You need to set up every GPIO pin (channel) you are using as an input or an output. To configure a channel as an input:
GPIO.setup(channel, GPIO.IN)
To set up a channel as an output:
GPIO.setup(channel, GPIO.OUT)
You can also specify an initial value for your output channel:
GPIO.setup(channel, GPIO.OUT, initial=GPIO.HIGH)

To read the value of a GPIO pin:
GPIO.input(channel)
To set the output state of a GPIO pin:
GPIO.output(channel, state) # state can be 0 / GPIO.LOW / False or 1 / GPIO.HIGH / True.

At the end of any program, it is good practice to clean up any resources you might have used. This is no different with RPi.GPIO.
GPIO.cleanup()

To create a PWM instance:
p = GPIO.PWM(channel, frequency)
To start PWM:
p.start(dc) # where dc is the duty cycle (0.0 <= dc <= 100.0)
To change the frequency:
p.ChangeFrequency(freq) # where freq is the new frequency in Hz
To change the duty cycle:
p.ChangeDutyCycle(dc) # where 0.0 <= dc <= 100.0
To stop PWM:
p.stop()

To import the time module*: import time*
The method **sleep()** suspends execution for the given number of seconds. The argument may be a floating point number to indicate a more precise sleep time.
time.sleep(t) # t in seconds

Appendix 3 Python objects and classes

Python is an object-oriented programming language. **Object-oriented programming** (OOP) focuses on creating reusable patterns of code, in contrast to procedural programming, which focuses on explicit sequenced instructions. When working on complex programs in particular, object-oriented programming lets you reuse code and write code that is more readable, which in turn makes it more maintainable.
One of the most important concepts in object-oriented programming is the distinction between classes and objects, which are defined as follows:

- **Class** — A blueprint created by a programmer for an object. This defines a set of attributes that will characterize any object that is instantiated from this class.
- **Object** — An instance of a class. This is the realized version of the class, where the class is manifested in the program.

Classes and objects are used to create patterns (in the case of classes) and then make use of the patterns (in the case of objects).

Classes
Classes are like a blueprint or a prototype that you can define to use to create objects.
We define classes by using the `class` keyword, similar to how we define functions by using the `def` keyword.
Let's define a class called `Shark` that has two functions associated with it, one for swimming and one for being awesome:

```
class Shark:
    def swim(self):
        print("The shark is swimming.")
    def be_awesome(self):
        print("The shark is being awesome.")
```

Because these functions are indented under the class `Shark`, they are called methods. **Methods** are a special kind of function that are defined within a class.
The argument to these functions is the word `self`, which is a reference to objects that are made based on this class. To reference instances (or objects) of the class, `self` will always be the first parameter, but it need not be the only one.
Defining this class did not create any `Shark` objects, only the pattern for a `Shark` object that we can define later. That is, if you run the program above at this stage nothing will be returned.
Creating the `Shark` class above provided us with a blueprint for an object.

Objects
An object is an instance of a class. We can take the `Shark` class defined above, and use it to create an object or instance of it. We'll make a `Shark` object called `sammy`:

```
sammy = Shark()
```

Here, we initialized the object `sammy` as an instance of the class by setting it equal to `Shark()`.

Now, let's use the two methods with the `Shark` object `sammy`:

```
sammy = Shark()
sammy.swim()
sammy.be_awesome()
```

The `Shark` object `sammy` is using the two methods `swim()` and `be_awesome()`. We called these using the dot operator (`.`), which is used to reference an attribute of the object. In this case, the attribute is a method and it's called with parentheses, like how you would also call with a function.
Because the keyword `self` was a parameter of the methods as defined in the `Shark` class, the `sammy` object gets passed to the methods. The `self` parameter ensures that the methods have a way of referring to object attributes.
When we call the methods, however, nothing is passed inside the parentheses, the object `sammy` is being automatically passed with the dot operator.
Let's add the object within the context of a program:

```
class Shark:
    def swim(self):
        print("The shark is swimming.")
    def be_awesome(self):
```

```
        print("The shark is being awesome.")
sammy = Shark()
sammy.swim()
sammy.be_awesome()
```

Let's run the program to see what it does. The output we will get is:
The shark is swimming.
The shark is being awesome.

Passing parameters
Let's create a class that uses a `name` variable that we can use to assign names to objects. We'll pass '`name`' as a parameter. By setting `self.name` equal to `name we can use the name internally`:

```
class Shark:
    def __init__(self, name):      # the underscores indicate that 'init' is predefined
        self.name = name
    def swim(self):                # Reference the name
        print(self.name," is swimming.")
    def be_awesome(self):          # Reference the name
        print(self.name," is being awesome.")
sammy = Shark("Sammy")             # __init__() is invoked automatically
sammy.swim()
sammy.be_awesome()
```

We can run the program now and the output will be:
Sammy is swimming.
Sammy is being awesome.
We see that the name we passed to the object is being printed out. We defined the `__init__` method with the parameter name (along with the `self` keyword) and defined a variable within the method.
If we wanted to add another parameter, such as `age`, we could do so by also passing it to the `__init__` method:

```
class Shark:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Then, when we create our object `sammy`, we can pass Sammy's age in our statement:

```
sammy = Shark("Sammy", 5)
```

Working with More Than One Object
Classes are useful because they allow us to create many similar objects based on the same blueprint.
To get a sense for how this works, let's add another `Shark` object to our program:

```
class Shark:
    def __init__(self, name):
        self.name = name
    def swim(self):
        print(self.name," is swimming.")
    def be_awesome(self):
        print(self.name," is being awesome.")
sammy = Shark("Sammy")
sammy.be_awesome()
stevie = Shark("Stevie")
stevie.swim()
```

We have created a second `Shark` object called `stevie` and passed the name `"Stevie"` to it. In this example, we used the `be_awesome()` method with `sammy` and the `swim()` method with `stevie`.
Let's run the program. It will give the output
Sammy is being awesome.
Stevie is swimming.
The output shows that we are using two different objects, the `sammy` object and the `stevie` object, both of the `Shark` class. Classes make it possible to create more than one object following the same pattern without creating each one from scratch.

Appendix 4 LCDdisplay.py

```
# The program consists of the following components:
# - imports of the objects we are going to use
# - some definitions to make the program more readable and maintainable
# - defining functions
#       * main: the main program; it is composed of an initialization + an endless loop (the loop is left with an exception )
#       * lcd_init: initialization
#       * lcd_byte: transfer of a byte
#       * lcd_toggle_enable: a supporting function used in lcd_byte
#       * lcd_string: transfer of a string consisting of a number of bytes
# - the actual program consisting of a call to ‘main’, the code to be executed with an exception and the termination.

import RPi.GPIO as GPIO      #import of GPIO module from RPi library
import time                  # import of time module

# Define GPIO to LCD mapping
LCD_RS = 3          # Register Select: low = command, high = data
LCD_E  = 5          # Enable (toggling this input means data is being transfered)
LCD_D4 = 7          # the next 4 inputs carry the data
LCD_D5 = 11
LCD_D6 = 13
LCD_D7 = 15

# Define some device constants
LCD_WIDTH = 16          # Maximum characters per line
LCD_CHR = True          # Register Select: high = data
LCD_CMD = False         # Register Select: low = command
LCD_LINE_1 = 0x80       # LCD RAM address for the 1st line
LCD_LINE_2 = 0xC0       # LCD RAM address for the 2nd line

# Timing constants
E_PULSE = 0.0005        # 2 constants to create a proper toggle signal
E_DELAY = 0.0005

def main():                                             # Main program block
        lcd_init()                                      # Initialize display
        while True:
                lcd_string("workshop 2018",LCD_LINE_1)  # Send some text to be displayed in line 1
                lcd_string("Raspberry Pi",LCD_LINE_2)   # Send some text to be displayed in line 2
                time.sleep(3)                           # 3 second delay
                lcd_string("in physical",LCD_LINE_1)
                lcd_string("environment",LCD_LINE_2)
                time.sleep(3)                           # 3 second delay

def lcd_init():
        GPIO.setmode(GPIO.BOARD)        # Use BOARD GPIO numbers
        GPIO.setup(LCD_E, GPIO.OUT)     # E is set to output
        GPIO.setup(LCD_RS, GPIO.OUT)    # RS is set to output
        GPIO.setup(LCD_D4, GPIO.OUT)    # DB4 is set to output
        GPIO.setup(LCD_D5, GPIO.OUT)    # DB5 is set to output
        GPIO.setup(LCD_D6, GPIO.OUT)    # DB6 is set to output
        GPIO.setup(LCD_D7, GPIO.OUT)    # DB7 is set to output
        lcd_byte(0x33,LCD_CMD)          # 0011 0011 Initialize
        lcd_byte(0x32,LCD_CMD)          # 0011 0010 Initialize
        lcd_byte(0x06,LCD_CMD)          # 0000 0110 Cursor move direction
```

```
        lcd_byte(0x0C,LCD_CMD)          # 0000 1100 Display On,Cursor Off, Blink Off
        lcd_byte(0x28,LCD_CMD)          # 0010 1000 Data length, number of lines, font size
        lcd_byte(0x01,LCD_CMD)          # 0000 0001 Clear display
        time.sleep(E_DELAY)

def lcd_byte(bits, mode):
        GPIO.output(LCD_RS, mode)       # selects for command or data
        GPIO.output(LCD_D4, False)      # make data inputs 0
        GPIO.output(LCD_D5, False)
        GPIO.output(LCD_D6, False)
        GPIO.output(LCD_D7, False)
        if bits&0x10==0x10: GPIO.output(LCD_D4, True)     #make the data input 1 if corresponding bit is 1
        if bits&0x20==0x20: GPIO.output(LCD_D5, True)
        if bits&0x40==0x40: GPIO.output(LCD_D6, True)
        if bits&0x80==0x80: GPIO.output(LCD_D7, True)

        lcd_toggle_enable()             # Toggle 'Enable' pin

        GPIO.output(LCD_D4, False)      # same as above for low bits
        GPIO.output(LCD_D5, False)
        GPIO.output(LCD_D6, False)
        GPIO.output(LCD_D7, False)
        if bits&0x01==0x01: GPIO.output(LCD_D4, True)
        if bits&0x02==0x02: GPIO.output(LCD_D5, True)
        if bits&0x04==0x04: GPIO.output(LCD_D6, True)
        if bits&0x08==0x08: GPIO.output(LCD_D7, True)

        lcd_toggle_enable()             # Toggle 'Enable' pin

def lcd_toggle_enable():                # Toggle enable
        time.sleep(E_DELAY)
        GPIO.output(LCD_E, True)        # E goes up
        time.sleep(E_PULSE)
        GPIO.output(LCD_E, False)       # E goes down
        time.sleep(E_DELAY)

def lcd_string(message,line):                       # Send string to display
        message = message.ljust(LCD_WIDTH," ")      # text is left justified with spaces added
        lcd_byte(line, LCD_CMD)                     # sends the code for line 1 or line 2 as a command
        for i in range(LCD_WIDTH):                  # sends the characters one by one
                lcd_byte(ord(message[i]),LCD_CHR)   # ‘ord(message[i]’ is the character at position i

if __name__ == '__main__':                          # if object is started as a program the code that follows is executed
        try:
                main()                              # call the main function
        except KeyboardInterrupt:                   # pressing a key causes a keyboard interrupt
                pass                                # with a keyboard interrupt the loop is left
        finally:                                    # the code in the finally clause is always executed
                lcd_byte(0x01, LCD_CMD)             # command ‘clear display’ is sent
                lcd_string("Goodbye!",LCD_LINE_1)   # text is sent
                GPIO.cleanup()                      # all ports used in program are set to input (safer)
```

Appendix 5 Interrupt Driven Objects

When we are waiting for an action to take place – ie a GPIO pin to change state – we can poll the pin permanently using an infinite loop, but that utilizes a lot of CPU power and makes it even impossible to use the computer for other tasks. There is however another way to deal with this kind of situations: interrupt based software objects.

GPIO interrupts allow a program to wait for GPIO events. Instead of repeatedly checking a pin, the code waits for a pin to be triggered, essentially using zero CPU power. Interrupts are known as “edge detection”; an edge defining the transition from high to low “falling edge” or low to high “rising edge”. A change in state, or transition between low and high, is known as an “event”.

How do we detect an interrupt? Obviously looping to check for an interrupt would defeat the point of using it, we would be “polling” the interrupt function instead of the GPIO pin. However, computers have in hardware and system software a function implemented that stops the program until the event occurs (no CPU time is wasted since other programs can still run) or starts a “call-back” function as soon as an interrupt occurs (the main program can continue with other things).

The following example illustrates what is happening: suppose you are waiting for a letter: polling is the act of you waiting at home all day, holding open the letter box and peering out waiting for the postman to arrive. An interrupt in this scenario would be a camera that watches the street for the postman. When it spies the postman it calls your mobile phone (the call-back) to let you know the postman is 10 minutes from your doorstep.

Let us take the example of some action to be taken as soon as motion is detected (no other work is done by this program):

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BOARD)
EVENT_PIN = 3
GPIO.setup(EVENT_PIN, GPIO.IN)
print ("PIR Module Test (CTRL+C to exit)")
time.sleep(2)
GPIO.wait_for_edge(3, GPIO.BOTH)    # alternatively: GPIO.RISING or GPIO.FALLING
print ("Motion Detected”)
GPIO.cleanup()
```

Alternatively the program can continue with other activities and when the event occurs a ‘sidestep’ is taken to the interrupt routine (‘MOTION’ is our call-back function):

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BOARD)
EVENT_PIN = 3
GPIO.setup(EVENT_PIN, GPIO.IN)
def MOTION(PIR_PIN):                    # 1 parameter is transferred when called
    print ("Motion Detected at pin ",PIR_PIN)
print ("PIR Module Test (CTRL+C to exit)")
time.sleep(2)
print ("Ready")
try:
    GPIO.add_event_detect(EVENT_PIN, GPIO.BOTH, callback=MOTION) # EVENT_PIN is passed as
                                        # parameter to the callback function
    while 1:                            # a loop is used here but anything can be done
        time.sleep(100)                 # instead
except KeyboardInterrupt:
    print ("Quit")
    GPIO.cleanup()
```

Appendix 6 Lists in Python

Python Lists
The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type. Creating a list is as simple as putting different comma-separated values between square brackets. For example −
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5];
list3 = ["a", "b", "c", "d"]
Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Accessing Values in Lists
To access values in lists, use the square brackets for slicing along with the index or indices to obtain the value available at that index. For example −
list1 = ['physics', 'chemistry', 1997, 2000]
list2 = [1, 2, 3, 4, 5, 6, 7]
print ("list1[0]: ", list1[0])
print ("list2[1:5]: ", list2[1:5])
When the above code is executed, it produces the following result −
list1[0]: physics
list2[1:5]: [2, 3, 4, 5]

Updating Lists
You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example −
aList = [123, 'xyz', 'zara', 'abc'];
aList.append(2009);
print ("Updated List : ", aList)
When we run above program, it produces the following result −
Updated List : [123, 'xyz', 'zara', 'abc', 2009]

Delete List Elements
To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you know what value to remove. For example −
list1 = ['physics', 'chemistry', 1997, 2000];
print list1
del list1[2];
print ("After deleting value at index 2 : ")
print list1
When the above code is executed, it produces following result −
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
Alternatively: list1.remove(1997) has the same effect.