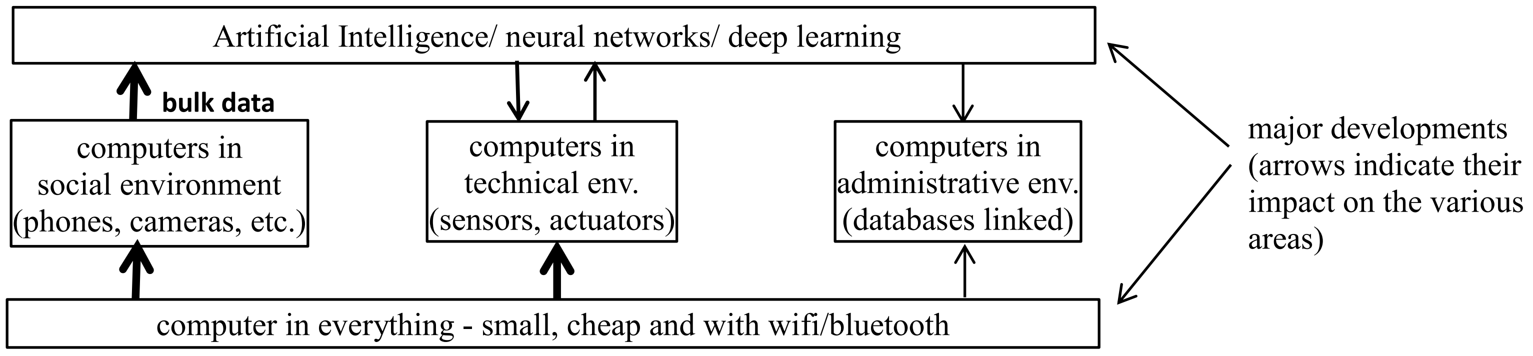
Handout Workshop 2019 - ***Microcomputers in Technical Applications***
by Chris Hendriks (hendrikschris@yahoo.com)

1. Introduction – a new information revolutions
Three major developments will together form a new information revolution:
1. Microcomputers will be present in everything, all devices in home and in industry – these computers generate an enormous amount of data.
2. Every computer, also those in devices, will be connected to the internet (Internet of Things)– this causes data to be available everywhere.
3. Data will be analyzed by Artificial Intelligence (AI) software. This opens the possibility to deal with more data as compared to the use of traditional software; and consequently, new applications are being developed. In technical applications the input for AI systems comes from intelligent sensors and the output is used to feed into intelligent actuators.


Artificial Intelligence/ neural networks/ deep learning
bulk data
computers in social environment (phones, cameras, etc.)
computers in technical env. (sensors, actuators)
computers in administrative env. (databases linked)
major developments (arrows indicate their impact on the various areas)
computer in everything - small, cheap and with wifi/bluetooth

The focus of the workshop is: computers in the technical (industrial) environment. In this environment large systems can be distinguished since every device (sensor or actuator) is connected to one or more central computer systems. The flow of information corresponds to the basic architecture of a robot (and a human!): intelligent sensors feed data into a (central) AI system which feeds intelligent actuators. In the workshop we will focus on intelligent sensors and actuators and their communication with a (central) system.

AI was initially 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. The technical/industrial environment followed. Initiatives in the area of linked administrative databases are following in for instance stock markets.
AI will be *the* major development for the coming decades. But AI can only show its relevance by receiving large amounts of data!

The drawback of this high degree of integrations is the increased risk of misuse of information and spreading of malware. Hence, security becomes a major issue.

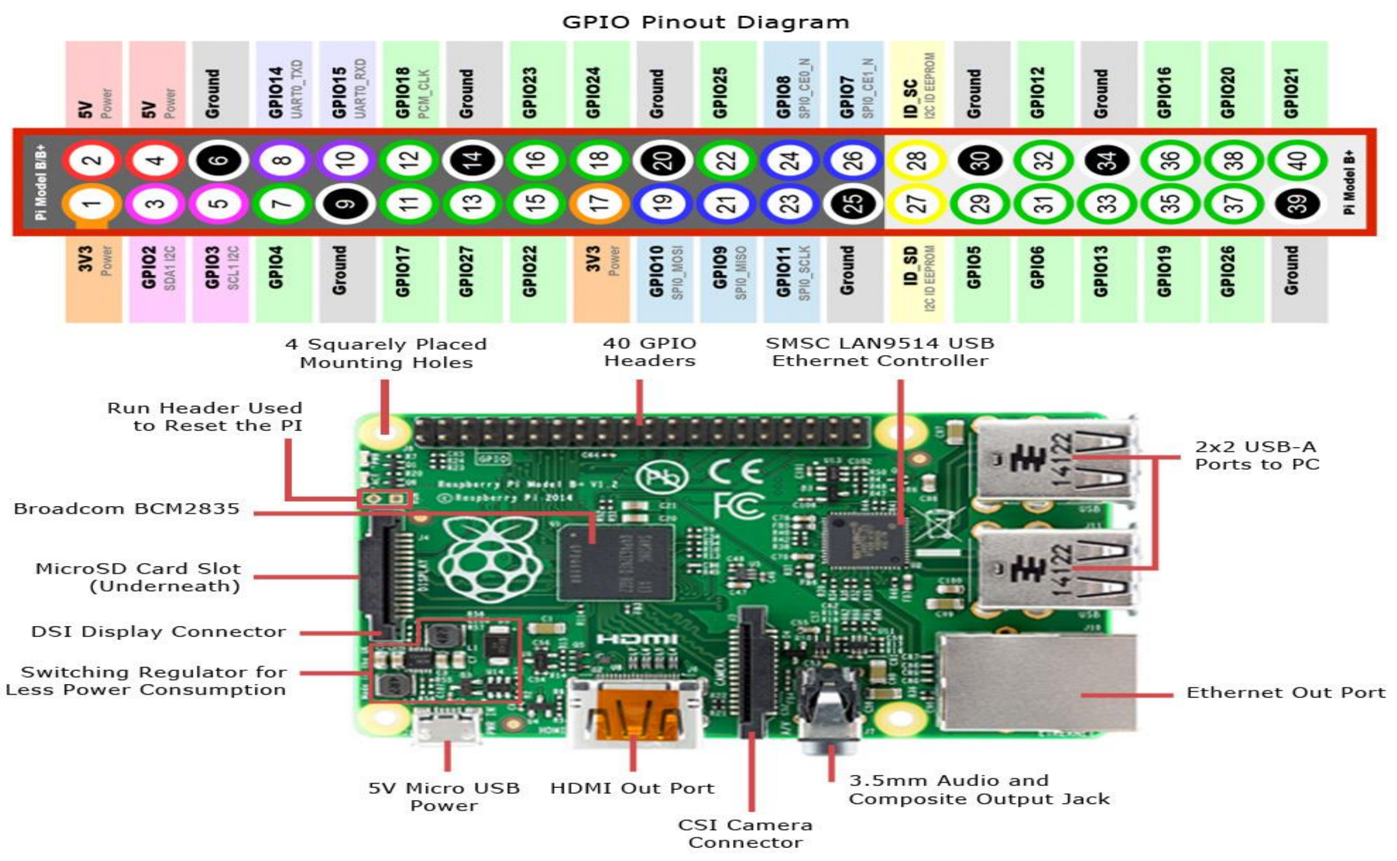
2. Raspberry Pi – single-board computer
A controller (for sensor or actuator) in the development phase needs a human interface (terminal/keyboard or notebook) but in the operational phase the controller consists of a few chips communicating with central computers through wifi or Bluetooth and with the sensors/actuators through direct wiring. The raspberry Pi is a mix between a controller (focusing on I/O activities) and a *general-purpose* computer (focusing on human interaction). The explosive growth in the number of computers/controllers like this have resulted in an enormous drop in price ($10 - $50) and an increase in availability of software.

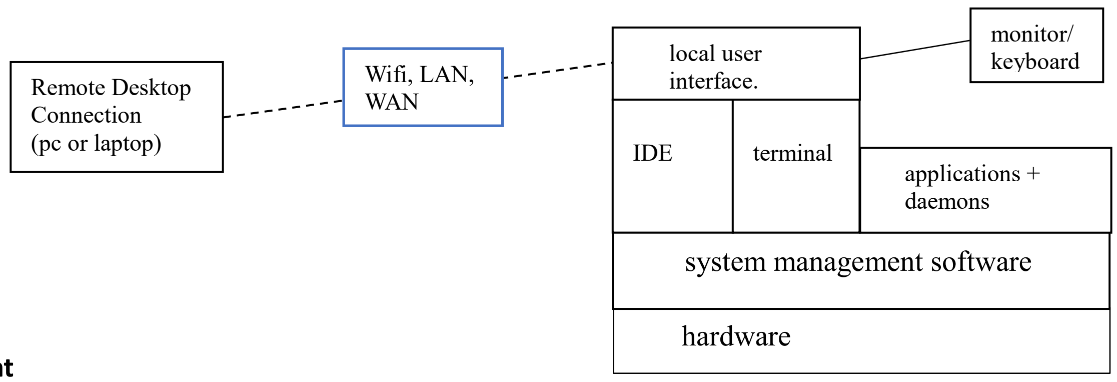
The Raspberry Pi is a typical representative of the marriage between the controller and the general-purpose computer. It is supplied with an Operating System from the Linux family. When opening the (virtual) terminal the regular Linux commands can be executed. For convenience we usually work from an 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 laptop through an ethernet cable or Wifi network. For implementing the system software and initial programming a terminal and keyboard are to be connected to the Raspberry.

In typical applications 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.

We will have a look at the Raspberry PI (RPi) in more detail. Apart from the processor and memory, the General-Purpose Input Output (GPIO) bus is an important component of the RPi. There are two ways of numbering the GPIO pins – by counting on the board (BOARD numbering) or by using their name consisting of the letters 'GPIO' and a number (we call this BCM numbering; in a program only, the number is used). In most exercises we will use BOARD numbering. General freeware (public domain and open source software) uses BCM numbering because of its independency of model and type.
Characteristics of the GPIO bus:

- port 3 and 5 have a pull-up resistor; the others have pull-down resistors;
- all ports can be programmed to be input or output ports; which state is safe when connecting external circuits?
- some groups of pins can also be programmed to be used for specific protocols. In the workshop we will use for instance the I2C (Inter Integrated Circuit) protocol available on pin 3 and 5**.**





3. Software development

3.1 IDE

For programming the Raspberry we use Python. Like most other modern languages Python is based on Object Oriented Programming (OOP). OOP software is based on *objects*. A possible metaphor to illustrate the concept of an object is the following:
You have an office that you can ask things to be done (e.g. executing administrative operations). You can however only give your orders through a counter. That's where you hand in your order plus the necessary data and that is where you receive the results. The advantage of an object is its clear interference with the rest of the software. If 'the office' does its job well, you can rely on it for 100 %. Usually an object is built up of other objects: a building has a counter where you hand in your orders; inside, other offices with counters are being used to execute specific parts of your order. And so on. We will use this metaphor later on again when we discuss the concept of *importing* objects.

The easiest introduction to Python[1] 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, amongst many other things, interpret variables or do math and evaluate expressions. For example:

```
>>> name = "Sarah"
>>> "Hello " + name
'Hello Sarah'
```

Alternatively, you can write a program with the text editor and run (or debug) it afterwards.
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 the following program and then run it after saving:

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

3.2 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 is shown in the box.
Also check what happens if you leave out one or both of the indentations.

```
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)
```

Variables
To save a value to a variable, assign it like this (see box):

```
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 how you can also include comments; sometimes it is a convenient way to exclude part of the program.
"""

If statements
You can use if statements for control flow (see box):

```
name = "Joe"                 # or choose any other name
if len(name) > 3:            # 'len' is a standard function; returns 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 (like the 'for'-loop). The following code will print 'hi' ten times (see box):

```
counter = 10
while (counter>0):           # <: smaller than; >: larger than; ==: equal to; !=: not equal to
   print("hi")
   counter=counter-1
```

Python Lists
The *'list'* can be written as a series of comma-separated values (items) between square brackets
Creating a list is as simple as putting different comma-separated values between square brackets.

To access a value in a lists, use the square brackets along with the index.
To add an element in a list use the append() method.

```
l = ['physics', 'chemistry', 1997, 2000]          # elements can be of different types
l.append(3.5)
print ("l[0] is:",l[0],"; and l[4] is:",l[4]) )
```

When the code in the box is executed, it produces the following result: l[0] is: physics ; and l[4] is: 3.5

[1] A good book for learning Python is: http://greenteapress.com/thinkpython/thinkpython.pdf

Functions

The syntax of a function is shown in the box.
Note the indentation and the colon!

```
def functionname( parameters ):
        .
        statements
        .
        return [expression]
```

Once the basic structure of a function is finalized, you can execute it by calling it from the program (object) it is defined in 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 a call to user defined function!")     # Now you call printme
```

You can return a value from a function as follows:

```
def sum(arg1, arg2):                                # Add both the parameters and return them.
  total = arg1 + arg2
  print ("Inside the function : ", total)
  return total
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
```

Objects and classes; modules, packages and libraries; import.

Objects are an encapsulation of variables and/or functions into a single entity (the office with the counter in our earlier metaphor).
A Class is like an object constructor, or a "blueprint" for creating objects.
Modules, packages and libraries are a way to group objects.
Modules in Python are simply Python files with a .py extension. The name of the module will be the name of the file. A Python module can have a set of functions, classes or variables defined and implemented. Any program (script) is a module.
Packages are namespaces (= common collection of names that are used) which can contain multiple modules (and/or other packages). They can be considered as directories.
The term library does not have any specific contextual meaning in Python. When used in Python, a library is used loosely to describe a collection of the core modules. Sometimes it is used as synonym for package.
We can use a module we created earlier by using the import statement. For instance: import time
While importing a module we can give it a different name. For instance: import time as t

We can refer to a particular function from an imported module of a package by using the *dot*-operator:

```
import time as t            # imports the module ‘time’ under the name ‘t’
t.sleep(5)                  # makes the program sleep for 5 seconds; ‘sleep’ is an object within the object ‘time’
```

Alternatively, we can import ‘sleep’ as a separate module:

```
from time import sleep as s    # imports the module ‘sleep’ from ‘time’ under the name ‘s’
s(5)                           # makes the program sleep for 5 seconds
```

We can also import all functions of a module with ‘from time import *’; in that case we can only refer to a function by the name it has in the module.

```
from time import *          # imports all modules from time
sleep(5)                    # makes the program sleep for 5 seconds
```

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* (see box).

```
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. an incorrect value was entered for 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.

Consider the following code (the variable __name__ gets the value '__main__' if the module is executed as a program (what is the advantage of defining the function p outside the main program?**)**:

```
def p():                              # the function p is defined
        i=1
        while True:
                print(i)
                i=i+1
if __name__ ==  '__main__':           # if the object was started as a program the code is executed
        try:
                p()                   # call the function p
        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
                print("end")
```

3.3 Terminal mode
We can also work in the terminal by clicking on the terminal icon. Since the operating system is Linux-like we can enter most Linux commands through the terminal. For instance, before switching off we give the command: sudo shutdown now ('sudo' stands for 'superuser do').

4 On/off outputs and inputs
Most ports of the General-Purpose Input Output bus (GPIO) can be input or output; some are Ground (0 V) or 5 V or 3.3 V. The program determines whether a port is input or output. In both cases a logical 0 is equal to 0 V and a logical 1 is equal to 3.3 V. Some ports have a pull-up[2] (e.g. port 5), others have an internal pull-down (e.g. port 40).

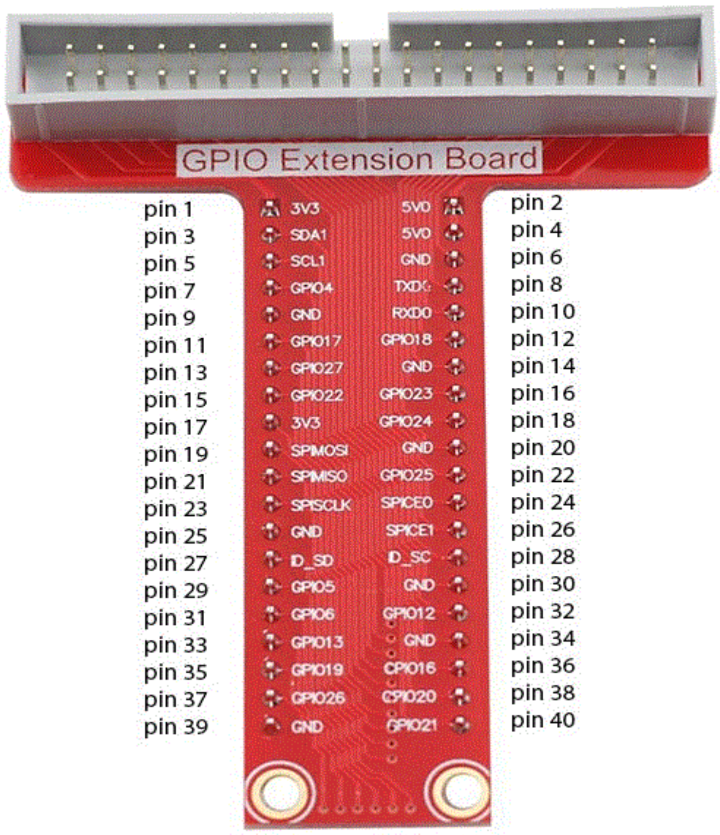
[2] A pull-up resistor is connected between input or output and 3.3 V; in that way the port becomes 1 if nothing is connected to it (input) or no output value is specified. In a similar way a pull-down makes a floating port equal to 0.

4.1 Switching a LED on and off using the Raspberry PI

This experiment demonstrates how to attach a LED to the GPIO 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 or Vcc. **A resistor is necessary to limit the current.** In the case of the traffic light the three LED's have a common ground.
Build the following circuit:
- take one of the colours of the traffic light (for instance the R-pin) and connect **it through a resistor** (any resistor between 300 and 500 Ohm will do) to a GPIO pin; you can choose any GPIO input/output pin – let's take pin 40;
- connect the common ground-pin of the traffic light (GND) to a ground output of the GPIO connector.

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)

```
import RPi.GPIO as GPIO       # Import GPIO library
Red = 40                      # Use pin 40 to connect to Red LED
GPIO.setmode(GPIO.BOARD)      # Use board pin numbering
GPIO.setup(Red,GPIO.OUT)      # Setup GPIO Pin 40 to OUT
GPIO.output(Red,True)         # Turn on GPIO pin 40
```



Run your program. We just told the RPi to supply a voltage of 3.3 V to our circuit using GPIO pin 40. Change the program by making pin 40 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.
Add this line to your program and switch the light on again.

We used the name 'GPIO' for the object we created by importing 'RPi.GPIO'. You can give the object any name. Give the object another name and run the program again (note: the name you give the object has to be used everywhere in the program where you refer to this object)

4.2 A complete cycle for the traffic light

Here is a slightly more advanced script that blinks the led on for 2 seconds and then off.
Extend your code as shown in the box (the added code is in **bold**; the comments are optional).

```
import RPi.GPIO as GPIO       # Import GPIO library
import time                   # Import 'time' library
Red = 40                      # Use pin 40 for the Red LED
timeOnR = 2                   # a variable makes it easier to change it later
GPIO.setmode(GPIO.BOARD)      # Use board pin numbering
GPIO.setup(Red, GPIO.OUT)     # Setup GPIO Pin Red to OUT
GPIO.output(Red,True)         # Switch on pin Red
time.sleep(timeOnR)           # Wait
GPIO.output(Red,False)        # Switch off pin Red
GPIO.cleanup()
```

Now we want to let the 3 LEDs go on for 2, 3 and 4 seconds respectively. Build the circuit by connecting the Y and G pin to port 38 and 36 respectively (**through resistors**). The program could look as shown in the box. Try the program.

The added code is in **bold**

```
import RPi.GPIO as GPIO                 # Import GPIO library
import time                             # Import 'time' library

Red = 40                                # Use pin 40 to connect to the Red LED
Yellow = 38
Green = 36
timeOnR = 2                             # using a variable makes it easier to change it later
timeOnY = 3
timeOnG = 4

GPIO.setmode(GPIO.BOARD)                # Use board pin numbering
GPIO.setup(Red, GPIO.OUT)               # Setup GPIO Pin Red to OUT
GPIO.setup(Yellow, GPIO.OUT)
GPIO.setup(Green, GPIO.OUT)

GPIO.output(Red,True)                   # Switch on pin Red
time.sleep(timeOnR)                     # Wait
GPIO.output(Red,False)                  # Switch off pin Red

GPIO.output(Yellow,True)                # Switch the yellow LED on and off
time.sleep(timeOnY)
GPIO.output(Yellow,False)

GPIO.output(Green,True)                 # Switch the green LED on and off
time.sleep(timeOnG)
GPIO.output(Green,False)

GPIO.cleanup()
```

Now we see that three 'blocks' of code are nearly the same. Would it not be possible to write it only once (in this case the 'blocks' of code are small but usually they are much larger)? Consider the program in the box. Test it.

```
import RPi.GPIO as GPIO                 # Import GPIO library
import time                             # Import 'time' library
Red = 40                                # Use pin 40 to connect to the Red LED
Yellow = 38
Green = 36
timeOnR = 2                             # using a variable makes it easier to change it later
timeOnY = 3
timeOnG = 4
GPIO.setmode(GPIO.BOARD)                # Use board pin numbering
GPIO.setup(Red, GPIO.OUT)               # Setup GPIO Pin Red to OUT
GPIO.setup(Yellow, GPIO.OUT)
GPIO.setup(Green, GPIO.OUT)

def switchLED (colour,timeOn):          # switchLED  is an arbitrary name
        GPIO.output(colour,True)        # Switch on pin colour
        time.sleep(timeOn)              # Wait
        GPIO.output(colour,False)       # Switch off pin colour

switchLED (Red,timeOnR)                 # Switch the red LED on and off
switchLED (Yellow,timeOnY)              # Switch the yellow LED on and off
switchLED (Green,timeOnG)               # Switch the green LED on and off

GPIO.cleanup()
```

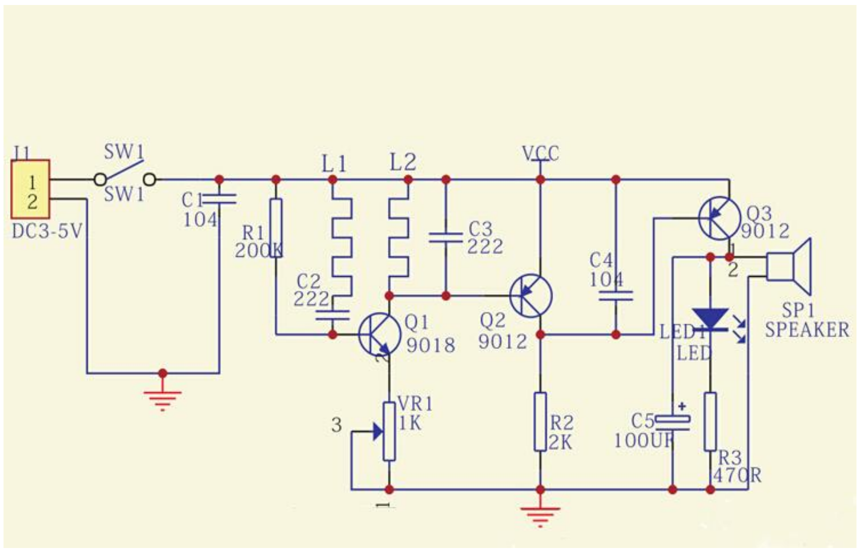
The code within 'def' is named a function. The function is called three times (once for each of the three colors).

4.3 Blinking yellow light
When the traffic light is not functioning the yellow light blinks.
Write the code to make the yellow light blink for 10 times with 2 seconds on and 1 second off.
Use the while statement to create a loop.

4.4 Use a metal detector as a switch
Use the metal detector to detect a car waiting in front of the traffic light and switch the light to green. Build the following circuit: the metal detector gets its own 5 V power (the red wire is 5V and the black wire is 0V; power is taken from the power rail; this power must share a common Ground with the Pi; why?). Use the output of the metal detector as input to the Pi (you can use any GPIO input/output pin – in the following example we use pin 32 = GPIO12).

Use the code in the box to read the input of the metal detector and try the program by moving a metal object over the detector.

```
import RPi.GPIO as G
import time as t
d = 32
G.setmode(G.BOARD)
G.setup(d,G.IN)
Try:
   while (True):
      print (G.input(d))
      t.sleep(1)
except KeyboardInterrupt:
   print ('All done')
G.cleanup()
```



Write a program that makes the traffic light remain red until a car arrives. When a car arrives, the light goes to green, then to yellow and then to red again. It should remain red for at least 10 seconds and until another car arrives.

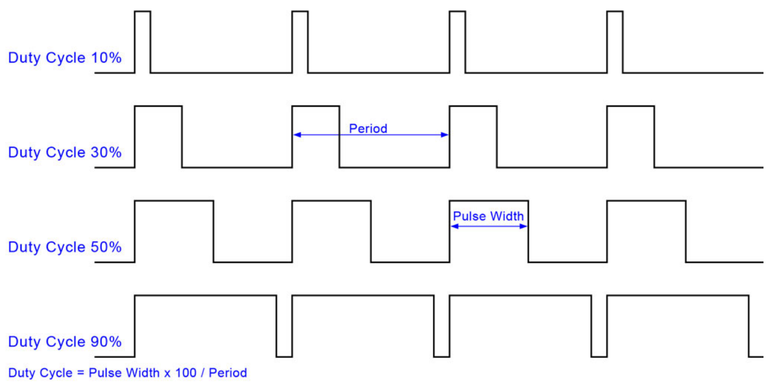
4.5 Objects and Classes - functions and methods
Like other modern programming languages Python is *object oriented*. That means that everything in Python is an object. Data are objects, the program itself is an object and functions are objects. All of these objects have types and unique IDs.
In Python an important category of objects is the function. The counter-metaphor of the object was discussed earlier. In this metaphor we compare the function-object with a room with a counter: through the counter you can ask the code of the function to be executed. Calling the function and handing in some parameter values is like going to the counter giving the parameters values and asking the function to be executed. One or more values might be returned. The original parameters are not affected.
Objects can 'contain' other objects. We can visualize this as a hall in which a number of rooms are built. The hall also has a counter through which everything within the hall can be accessed. The hall can be part of a larger hall, and so on.
It is up to the designer of the software to choose the halls and rooms in such a way that the software is testable, re-usable and maintainable.

Going back to our traffic lights. Suppose we have a crossing of two streets (Kingstreet and Queenstreet). That means we have 4 traffic lights (for example Kingstreet-north, Kingstreet-south, Queenstreet-east and Queenstreet-west). For each traffic light we use the functions for a complete cycle, blinking yellow and car detection. How can we write the program in such a way that it remains maintainable (which means it has an orderly structure)? Of course we could define functions as we did before and have an extra variable referring to the traffic light we are working on. More attractive however is to make a blueprint for a general traffic light and use it to make 4 different objects, one for each traffic light. The blueprint is called a *class*. The class mechanism makes it easier to make different objects that have to a large extend the same structure. The class itself is no object, it only describes the structure of the objects that are made of it afterwards. In that sense it is really a blueprint. The functions defined in a class are called *methods*. The object made from a class is called an instance of the class.
Classes are not only valuable when a number of objects are needed that are more or less similar, they become essential when a number of objects are needed that run in parallel. We will discuss this situation in more detail in chapter 12 Real-time application.

```
class TrafficLight:
        def  __init__(self,red,yellow,green,detector):
                self.red=red                            # the parameters that will be passed on are copied
                self.yellow=yellow                      # to parameters within the object that is made of
                self.green=green                        # the class later; in the metaphor of the room: a copy
                self.detector=detector                  # is made of the parameters instead of the original
                                                        # parameters being used
        def cycle(self):
                GPIO.output(self.red,True)              # Switch on pin Red
                time.sleep(timeOnR)                     # Wait
                GPIO.output(self.red,False)             # Switch off pin Red
                        :
        def blink(self):                                # the blink method is defined
                        :
        def detect(self):                               # the method that detects whether a car has arrived
                        :
# imports and initializations
                        :
# 4 objects are instantiated, one for each traffic light
north = TrafficLight(NorthRed,NorthYellow,NorthGreen,NorthDetector)
south = TrafficLight(SouthRed,SouthYellow,SouthGreen,SouthDetector)
east = TrafficLight(EastRed,EastYellow,EastGreen,EastDetector)
west = TrafficLight(WestRed,WestYellow,WestGreen,WestDetector)

# remainder of program code in which we for instance start a cycle at the lights of
# Kingstreet-north by the statement: north.cycle();
```

5 PWM (pulse width modulation)



PWM is used in many systems. It means that a periodic signal is high for 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.

5.1 Changing the duty cycle

In the following examples we use a multi-color LED. The multi-color LED we will be using consists of 3 LEDs, one for each of the primary colors red, green and blue. It can have a common – (ground) or a common + (V).
The R, B and G pins are connected through 330 Ohm resistors to the output pins of the Raspberry (attention: one type of RGB leds has resistors included; the other one has not – in that case external resistors should be included).
PWM is available through the pigpio library. The pigpio library generates an accurate pwm signal (we could also generate a pwm signal in software but that is less accurate). **Pigpio uses BCM numbering**.
Since pigpio affects the hardware directly we have to start a specific program at the highest command level. For this purpose, open the terminal and enter the shell command: sudo pigpiod ('sudo' stands for 'superuser do'; pigpiod is the daemon for pigpio). At the end of our program we have to terminate the daemon with the shell command: sudo killall pigpiod.

Let's make a PWM signal at the output port 21 (BCM numbering; that is 40 in BOARD numbering) of 1 Hz. The output port is connected through a 330 Ohm resistor to the red input of the multicolor led.
We create a PWM instance (object) with the command pwm=pigpio.pi() whereby pwm is an arbitrary name.
The code in the box creates the PWM signal with a duty cycle of 25 %:

```
import pigpio
import time

pin_r=21
pwm_r=pigpio.pi()                          # object is made as instance from class
pwm_r.set_mode(pin_r,pigpio.OUTPUT)        # makes the port an output port
pwm_r.set_PWM_frequency(pin_r,1)           # frequency is set to 1 Hz
pwm_r.set_PWM_dutycycle(pin_r,25)          # duty cycle is set to 25 %
time.sleep(5)
pwm_r.set_mode(pin_r,pigpio.INPUT)         # pwm signal is removed by making pin_r input
pwm_r.stop()                               # object is removed
```

don't forget to enter the command sudo pigpiod in the terminal; this creates a daemon

See what happens if we change the duty cycle to 90 % after 10 sec. using the command pwm_r.set_PWM_dutycycle(pin_r,90) and including another sleep period.
Try other values for the duty cycle.

Try the following exercises:

a. the three leds go on and off after each other;
b. the three leds go on and off so that colors are mixed;
c. make the red light increase in intensity from off to on, in 10 seconds). You will notice that the brightness we experience is not linear; adjust the minimum and maximum value of the brightness in such a way that it gives a better visual effect.
d. make the three leds slowly increase in intensity and after that slowly decrease in intensity (after each other or simultaneously).
e. is it possible to increase the intensity of 'red' during 5 seconds and at the same time increase the intensity of 'green' in 7 seconds?

5.2 Changing the frequency

Replace the LED with a small speaker (buzzer) and give the output a PWM signal with a frequency of 100 Hz and a duty cycle of 50%. What happens if you change the frequency keeping the duty cycle the same?

5.3 Mixing colors

Now suppose we want to switch on one light and have it go repeatedly through a cycle of increasing brightness in 2 seconds and decreasing brightness in 2 seconds and, while this is going on, we want to start 1 second later with another light going through the same cycle. With the sequential programming we used so far this is very difficult – certainly if the cycles differ in duration and if there are more than two.
We would like to have a feature that allows us to start one process and while this process is running, we start another process independently. This feature is offered by 'threading' (discussed in detail in chapter 12 *Real-time applications).* The example in the box shows how to start a thread (you can try this exercise after reading chapter 12; don't forget to switch the ports to *input* and stop the objects).

```
import pigpio
from threading import Thread
import time as t
red= 40
blue=38
:
def completeCycleRed():
:
def completeCycleBlue():
:
threadRed=Thread(target=completeCycleRed)
threadRed.start()
t.sleep(1)
threadBlue=Thread(target=completeCycleBlue)
threadBlue.start()
:
```

6 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 about 5% the servo angle will be at its minimum (-90 degrees); if the duty cycle is 7.5 % the servo will be at its center position (0 degrees) and if it is about 10 % it will be at its maximum (+90 degrees).
Since the range of the duty cycle in the PWM method in pigpio is 0 – 255: 7.5% is about 18, 5% is about 12 and 10% is about 24.
Connect the bottom servo (red: positive of external power supply; black: ground of external power supply and Raspberry; yellow: PWM input signal). We will use port GPIO21 in BCM numbering (in BOARD numbering pin 40) of the Raspberry for the PWM signal.
Enter and run the program in the box (make sure you know the meaning of each of the statements). In the program we use the keyboard keys (in this case ‘ a’ and ‘ s’ but you can take any other combination) to rotate the servo.
Note: you will notice that the servo has a hysteresis (that means that when it changes direction, it takes an extra step); this hysteresis comes from the internal gearbox.

Connect the other two servos. Chose pins to supply the PWM signal. Write a program that allows a user to make the three servos turn in either direction.
The movement to any position is controlled by repetitive pressing of keys on the keyboard (choose keys for each of the movements). Obviously three different objects are needed, each with its own unique name. Test the program by making the robot arm pick up an object and place it at a predefined position.

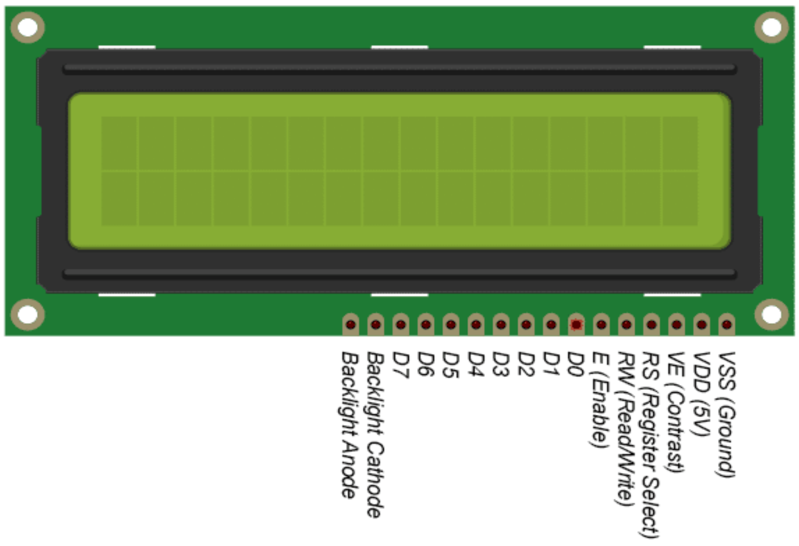
Extend the program in such a way that all keystrokes are added to a list. Print the list at the end.

Extend the program in such a way that after pressing the ‘q’ key the program asks whether you want to repeat the movement of the arm. If ‘y’ is hit the servos move to the start position and make the movement stored in the list. Hint: include a small delay between the steps read from the list; otherwise the movement goes too fast. Also, if time allows: various optimizations are possible like making the delay dependent on the kind of movement – continuous or near reaching a stop position – and deleting unnecessary movements from the list.

start the pigpiod deamon from the terminal

```
import  pigpio
import  time
pin  =  21
d=18
pwm=pigpio.pi()
pwm.set_mode(pin,pigpio.OUTPUT)
pwm.set_PWM_frequency(pin,50)
pwm.set_PWM_dutycycle(pin,d)
c=' '
while (c!='q'):
   if c=='a': d=d-1
   if c=='s': d=d+1
   if d<8:
      d=8
      print ("lower limit reached")
   if d>28:
      d=28
      print ("upper limit reached")
   pwm.set_PWM_dutycycle(pin,d)
   print (“The present position is: “,d)
   c=input("enter next move : ")
pwm.set_mode(pin,pigpio.INPUT)
pwm.stop()
```

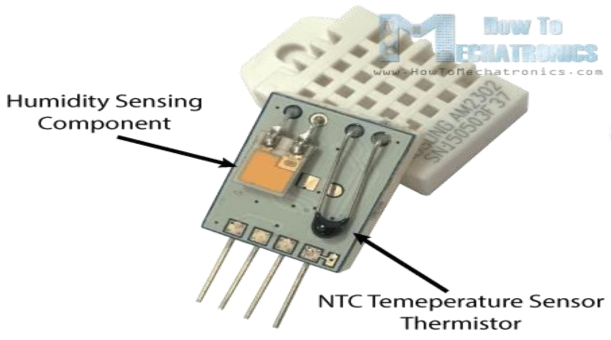
7 LCD display with parallel input

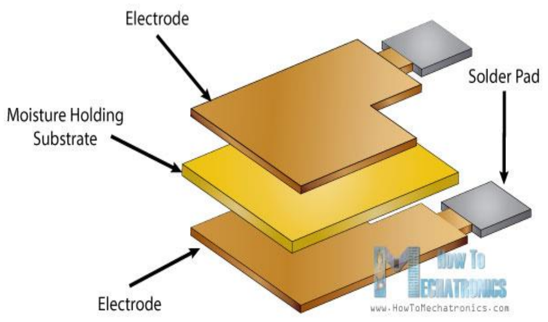


We will import software that uses the LCD display with the following connections to the GPIO pins (BOARD numbering):
1 - Ground; common to the Raspberry and the separate power supply
2 - VCC; we will use a separate 5V power supply
3 - (contrast) → connected to Ground
4 - RS → 40 Register Select; 0: Command, 1: Data
5 - (read/write select) → connected to Ground
6 - E → 38 Enable data transfer
11 - D4 → 37 databit 0
12 - D5 → 35 databit 1
13 - D6 → 33 databit 2
14 - D7 → 31 databit 3
15 (5V) and 16 (0V) are connected to the backlight.

1. After making the connections run 'LCDdisplay.py'. Try to understand the program (appendix LCDdisplay.py)
2. You can use the functions from 'LCDdisplay.py' by importing the code in your program (import LCDdisplay as lcd - 'lcd' is an arbitrary name) and calling the functions as for instance lcd.lcd_init()).
Write a program that shows some text on the first line and blinks another text on the second line.

8 Temperature and humidity sensor
The DHT11 consist of a humidity sensing component, an NTC temperature sensor (or thermistor) and an IC on the back side of the sensor. NTC stands for *negative temperature coefficient* which means that the resistance decreases as the temperature increases.


Humidity Sensing Component
NTC Temeperature Sensor Thermistor



The humidity sensing component 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 an 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[3] of semi-conductive materials such as ceramics or polymers in order to provide larger changes in the resistance with just small changes in temperature.

[3] Sintering is the process of compacting and making solids at low temperature – without melting.

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.

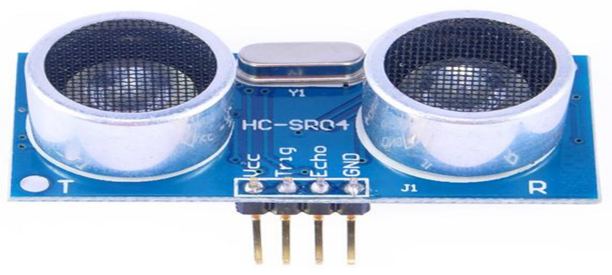
Connect the data output of the DHT11 (pin in middle)[4] to pin 40 of the Raspberry (pin 40 = GPIO21; the software we will use uses BCM numbering). Connect Vcc (5 V) and ground (pin near led). Test the software in the box.

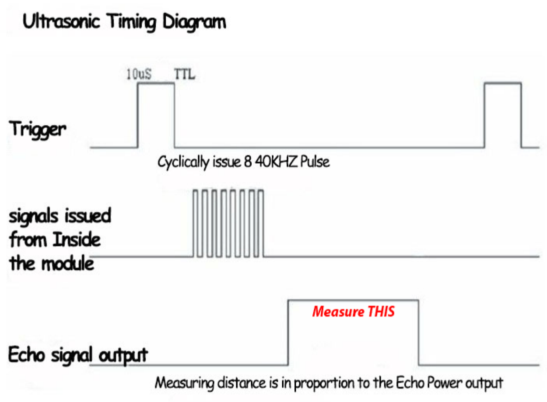
```
import Adafruit_DHT
import time

i=0
while (i<10):
    humidity, temperature = Adafruit_DHT.read_retry(Adafruit_DHT.DHT11, 21)
    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 "T = "+ str(temp)).
If time allows add the traffic light in such a way that the yellow and red indicate that the humidity gets over 80 % or 90 % respectively.

9 Distance measurement




Ultrasonic Timing Diagram
10uS
TTL
Trigger
Cyclically issue 8 40KHZ Pulse
signals issued from Inside the module
Measure THIS
Echo signal output
Measuring distance is in proportion to the Echo Power output

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 (e.g. 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. This is done as follows: in a "while" loop we record the last timestamp for a given condition with the time.time() function. 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 echo 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 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.

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

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

$$17150 \times Time = Distance$$

[4] Some types have 4 pins; top view (with raster): left = 5 V, second = data (with 5 k pull up resistor), pin 3 not connected and pin 4 to ground.

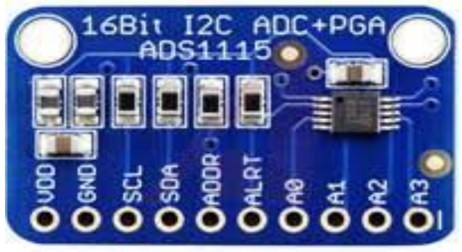
Run the program in the box. Adjust the port numbers if necessary. Take some measurements with a solid object between 5 cm and 50 cm (the actual upper limit is higher). For measuring distances: a sheet of A4 paper is 29.7 cm x 21.0 cm.
What is the accuracy of the measurement device? What is the main source of inaccuracy? How can the measurement be made more accurate?

```
import RPi.GPIO as GPIO                    # import libraries
import time

GPIO.setmode(GPIO.BOARD)
TRIG = 40                                  # trigger
ECHO = 38                                  # echo
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()
```

10 Connecting the analog world
Most of the surrounding world is analog. So, we need special integrated circuits to connect an analog output to the Raspberry Pi or any other computer. Some sensors have the conversion integrated in the device – like the temperature and humidity sensor we used earlier. But in other cases, we have to take care of the AD (analog-digital) conversion ourselves. The ADS1115 is a popular chip used for this conversion.



The ADS1115 has the following specifications:
- it has a 16-bit internal configuration register which allows us to program, among others, the gain of the input signal, the number of samples per second, the start of the conversion and whether we want the conversion to be continuous.
- it uses a 4-wire bus protocol (a 'bus' is a communication system that allows the transfer of data between a master and one of the slaves). The protocol is referred to as I2C (Inter-IC bus). Apart from the common 3.3 V and Ground the bus consists of a serial clock (supplied by the master) and a serial data line.
- there are 4 analog inputs (A0 – A3).

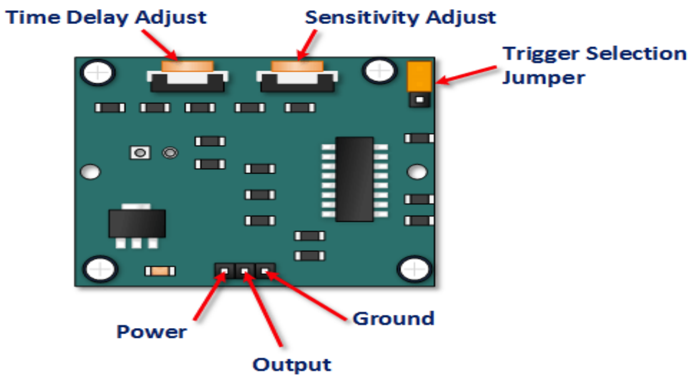
Connect 5V and ground from the Raspberry to the AD converter. Connect the 3,3 V from the Raspberry to the A0 input of the AD converter. Connect the serial data (SDA) and serial clock (SCL) of the ADS1115 to the Raspberry.
Run the program in the box and explain the observations.

Connect a 10k resistor, a photo resistor and a LED in series with the 10k resistor connected to 3.3 V and the LED to Ground. Measure the voltage over the LED and over the photo resistor when light intensity varies (hint: you can vary the light received by the photo resistor by putting anything on top of it).

```
import Adafruit_ADS1x15                    # import library
import time
adc=Adafruit_ADS1x15.ADS1115()             # create instance (object); adc is an arbitrary name
while True:
   v_binary = adc.read_adc(0)              # read voltage through ADC input 0 as a binary number
   v_analog= (v_binary/32768)*4096         # convert the binary value to an analog value (32768 = 4096 mV)
   v_analog=round(v_analog)                # v_analog is rounded to the nearest whole number
   print ("analog voltage = ",v_analog, " mV")
   time.sleep(1)
```

11 Motion sensors

The motion sensor (also called PIR sensor – passive infrared sensor) detects a change in infrared (heat) radiation in relation to position. This change is interpreted as motion. The white cover works as a lens.



Turn 'Time Delay Adjust' - the time the output remains high after detecting motion – fully left (min. 2.5 sec.).
Turn 'Sensitivity Adjust' fully left (minimal).
The PIR uses a 5 V power supply.
Use a GPIO port without a pull up resistor (e.g. port 40).

Write a program that prints the output of the PIR every second.
Interpret the result (does the PIR measure movement permanently?).
If time allows you can extend the program in such a way that a LED is switched on whenever motion is detected. How can you make the LED being on permanently when there is permanent movement?

12 Real-time applications

In real-time environments – as in multi-user environments - it is often necessary to have more than one object being executed at the same time. The mechanism of *threads* makes this possible. Of course, there is only a single processor doing the work but by dividing the time in small time-slots and running each of the threads in a time slot the threads run virtually in parallel. Simplified: with 3 threads every third slot is allocated to a particular thread so it looks as if there are three processors each running at one third of the speed.

Sometimes a thread is waiting for an external event to take place. In that case the thread has to be made inactive. When the event takes place, the inactive state has to be interrupted so that the thread continues processing again.

12.1 Sharing variables among multiple threads

Sometimes variables have to be shared among multiple threads. To use a variable in more than one thread it must be defined as *global.*

Run the program in the box. The program output shows the increase of *cycle* by 1; every 5 seconds *cycle* increases by 5 from the thread (as long as it is running). Both threads (main and

```
from threading import Thread
import time
global cycle                                  # cycle is defined as global variable
cycle=0                                       # cycle gets the initial value 0

def FiveSecond():                             # function is defined
        global cycle                          # variable shared with main thread is used
        while (cycle<40):                     # loop: every 5 sec 'cycle' is incremented by 5
                time.sleep(5)
                cycle = cycle + 5
                print ("5 Second Thread cycle + 5: ", cycle)
        print("Thread terminates")

FiveSecondThread = Thread(target=FiveSecond)          # create instance of Thread
FiveSecondThread.start()                      # start thread

while (cycle<50):                             # main program
        cycle = cycle + 1
        print ("Main Program increases cycle + 1: ", cycle)
        time.sleep(1)                         # one second delay
print ("Main Program terminates")
```

FiveSecondThread) modify the variable ‘cycle’ and print the value. What happens if the FiveSecondThread continues till cycle is 60?

Threads can be created for a variety of functions. 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 1: the blinking code runs in a separate thread; the main program communicates with the user; a global variable is used for the frequency;
Hint 2: the function ‘input’ returns a string; to get an integer use: int(input());
Hint 3: be aware that ‘pigpio’ uses BCM numbering (so BOARD numbering port 40 is BCM numbering GPIO21);
Hint 4: if the program terminates in an unexpected way you can terminate the daemon by entering ‘sudo killall pigpiod’ as a shell command (in the terminal); after that you can start the daemon again with the shell command ‘sudo pigpiod’.

12.2 Interrupt Driven Threads
When we are waiting for an action to take place – for instance 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: interrupts.
GPIO interrupts allow threads 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 based on “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”.

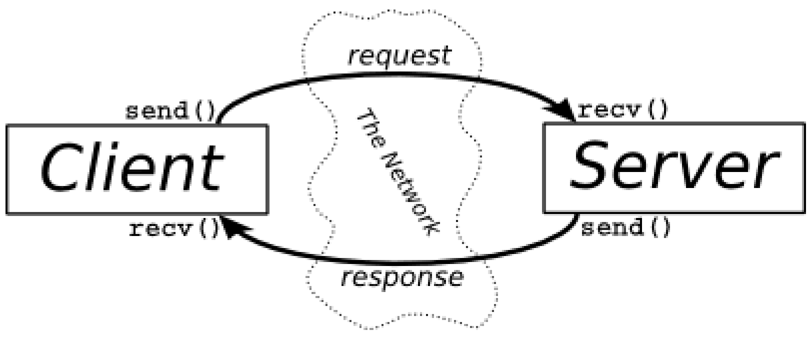
How do we detect an interrupt? Obviously looping to check for an interrupt would defeat the point of using it, we would be “polling” the interrupt function instead of the GPIO pin. However, computers have in hardware and system software a function implemented that stops a thread from running until the event occurs (no CPU time is wasted since other threads 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.

Modify the example of the previous section in such a way that some event is to take place before the incrementing of ‘cycle’ in the FiveSecondThread is executed once. For generating an event you can make the input high with a pull-up resistor and switching it to 0. The code in the box shows the modification of the thread that waits for the interrupt (hint: make the main program run until a KeyboardInterrupt terminates it – this allows you for more time to generate the interrupt).

```
def FiveSecond():                       # function is defined
        global cycle                    # the same variable as in the main thread is used
        G.wait_for_edge(p, G.BOTH)      # wait for any edge at port p; alternatively: G.RISING or G.FALLING
        cycle = cycle + 5
        print ("5 Second Thread cycle + 5: ", cycle)
```

Adjust the program in such a way that the interrupt can be repeated. Every time an interrupt is given the counter should be incremented by 5 just once. What do you notice if no precautions are being taken? How can multiple interrupts be avoided?

13 Communication
The Raspberry Pi has on-board Wifi and Bluetooth. For the communication example we will use TCP/IP over wifi.
Data exchange in Python is based on sockets. A socket is an endpoint for communication between two machines. The connection between two sockets can be considered as a (bi-directional) pipe: what goes in at one side comes out at the other side. The medium can be the Local Area Network, Wide Area Network or the Internet. Also Bluetooth programming in Python follows the socket programming model.



As said before a socket represents an endpoint of a communication channel. A channel is always formed between a client and a server:
Server: A server is a machine that waits for client requests and serves or processes them.
Client: A client on the other hand is the requester of the service.

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 is terminated by the user software.

The class 'socket' is a predefined Python class in the Raspberry Pi. By importing it we have all the communication methods available (for TCP/IP but also for Bluetooth and other protocols).

A TCP/IP address - in version 4 - is represented as a string of 4 octets (together forming the 32-bit address). For example 192:168:0:181.
The **IP address identifies the device** e.g. the computer.
However, an IP address alone is not sufficient for running network applications, as a computer can run **multiple applications** and/or **services**.
Just as the IP address identifies the computer, the network **port** identifies the **application or service** running on the computer.

The following analogy illustrates the meaning of address and port number:
If you have an apartment block the IP address corresponds to the street address. All of the apartments share the same street address. However, each apartment also has an apartment number which corresponds to the Port number.
A Port number uses 16 bits - so Ports can have values from **0** to **65535** decimal.
Port numbers up to 49151 have specific functions. Above that number they can be used by user programs.
In summary: a socket is the combination of **IP address plus port number.**

The examples in the box show how to establish a connection using a TCP/IP socket, sending some text from the client to the server, the server returning the same text in uppercase. The client terminates with 'q' and sends disconnecting code.

Server:

```
import socket                           # the class 'socket' is imported
server_addr = ("192.168.0.xxx", 50000)  # the IP address of the server should be entered + any port of server
mySocket = socket.socket()              # the object 'mySocket' is created (instance of method socket of class socket)
mySocket.bind(server_addr)              # mySocket is linked to the given address
print("waiting for connection")
mySocket.listen(1)                      # wait for connection request; 1 = the number of simultaneous connections
conn, client_addr = mySocket.accept()   # conn = new socket object;
print ("Connection from: ", str(client_addr))
while True:
        data = conn.recv(1024).decode() # transmitted message is decoded
        if not data:                    # when client sends termination code
                break                   # exit from loop (server closes connection)
        print ("from connected user: ", data)
        data = data.upper()             # message converted to uppercase
        print ("sending: ", data)
        conn.send(data.encode())        # encoded data are sent
conn.close()
```

Client:

```
import socket
server_addr=("192.168.0.xxx",50000)          # the IP  address of the server + port number of server should be entered
mysocket=socket.socket()
mysocket.connect(server_addr)
message=input (“message to be sent --→ “)
while message!="q":
        mysocket.send(message.encode())      # client sends message
        data=mysocket.recv(1024).decode()    # client receives data from server
        print("received from server: ",data)
        message=input("next message to be sent ---> ")
mysocket.close()                             # disconnect is sent to server; object is terminated
```

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 IP address plus 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. In the line ‘mySocket.listen(1)’ the ‘1’ indicates that only 1 connection is accepted
The client-socket that is used to establish an outgoing connection connects to the server using the specified server address (in the workshop set-up: IP address 192.168.0.xxx[5] and the port number 50000). Now what to do if you only know the name of the RPi? By running the command ‘ifconfig’ you can find the various IP addresses. Under ‘wlan0’ the line ‘inet’ shows the IP address (192.168.0.xxx ; the last 4 octets are a number allocated by the router - it is different for everybody

Test the program above by having two Pi’s communicating (determine who is server and who is client). Which one, the server or the client, should be started first?

Write a client and a server program in such a way that the client asks the server for the position of a switch at the server and switches a LED on or off accordingly.

14 The 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, i.e. 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)
# with the construction %s %i the value of i is converted into a string and included in the text; in that way the pictures
# get successive numbers and in that way unique names
```

[5] This is the address range of the wifi network; the address might be different in a different setup.

Appendix 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 (based on BOARD numbering)
LCD_RS = 40            # Register Select: low = command, high = data
LCD_E  = 38            # Enable (toggling this input means data is being transfered)
LCD_D4 = 37            # the next 4 inputs carry the data
LCD_D5 = 35
LCD_D6 = 33
LCD_D7 = 31

# 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 2019",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 Technical",LCD_LINE_1)
                    lcd_string("Applications",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 started as program the code 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)
```