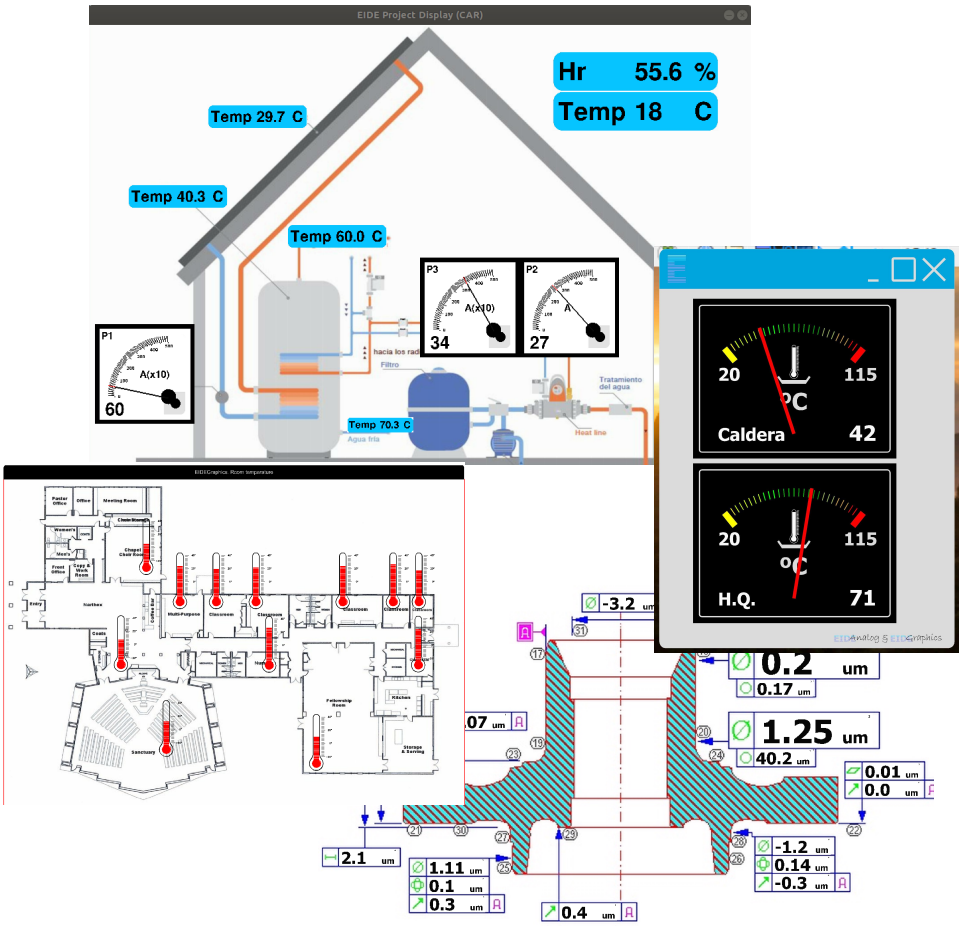
Data acquisition using Python and Raspberry Pi



EIDEAnalog library
(Making of)

Revisions

N°	Date	Paragraph	Description
0	May the 7th, 2020	-	First chapter issued.
1	May the 10th, 2020	-	Second chapter issued.
2	May the 18th, 2020	-	Third chapter issued. Misspelling and untranslated text fixed.

Table of Contents

1.- Brief introduction....4
1.1.- Blog declared goal....5
1.1.1.- OOP design pattern....6
1.1.2.- Data acquisition systems. A/D conversion....6
1.1.3.- Blog development. Pedagogics....7
1.1.4.- Conclusions. Goals....7
1.2.- Basic knowledge to follow the blog. Prerequisites....8
1.2.1.- Electronics. A/D converters. Binary numbers....9
1.2.2.- Raspberry....10
1.2.3.- Linux....12
1.2.4.- Python. OOP....14
1.2.5.- Other prerequisites....15
2.- Project: ADC library. Specs & tools....16
2.1.- Specs....16
2.2.- Raspberry....16
2.3.- ADC. Analog to digital converters....18
2.3.1.- oneWire bus....18
2.3.2.- ADS1115....20
2.3.3.- Arduino ProMini....22
2.4.- Programming language. Python. IDE....24
2.5.- Design using OOP....25
2.5.1.- Abstract....25
2.5.2.- Example....25
2.5.3.- OOP. Conclusions....27
3.- Library pattern. First approach....29
3.1.- Noun/verb....29
3.2.- Classes as services suppliers....30
3.2.1.- Bus classes. Sensor class. Conversion....30
3.2.2.- *readout* method....31
3.2.3.- *Calculation agent* classes....33
3.3.- Library pattern. Classes....34

1.- Brief introduction.

We will explain through this 'blog' how a *library* to help capturing physical parameters (analog-digital conversion) has been conceived and developed using Python as a programming language and OOP (Object Oriented Programming) as a technique. The hardware to be used is A) a Raspberry Pi card without added hardware for the oneWire bus, B) the same Raspberry plus an ADS1115 type analog to digital converter and C) the Raspberry connected to an Arduino ProMini; for the first option probes used are of the DS18B20 type, for the other two *combos* half a dozen conventional sensors are used - temperature, voltage - which will be opportunely depicted. As will be exhaustively explained in the blog, much emphasis is placed on the decision of the pattern of (the classes that make it up) the library and its development, issues that are usually the most complex in this type of programs.

The blog may be of interest:

- As a guide to programming practices for intermediate level courses. It may be even possible that in certain university degrees it will serve for the same purpose.

- For fans of (or professionals in) programming who wish to improve or test their knowledge of OOP.

- Engineers in the ‘data acquisition’ field. Although the hardware used has not an industrial grade (not, at least, the one that has been specifically used to prepare this blog), it is not ruled out that it may be of their interest.

Descriptions along the blog are intended to be as concise as the subject permits, which, as will be pointed several times, is not a simple issue.

(Should you just want to familiarize yourself with the use of the library and not with its design, please go directly to point 8, "Use of the library").

1.1.- Blog declared goal.

The stated objective of the blog is, therefore, to develop a -small- library written in Python to ease the use of various types of AD adapters, the popular ADS1115, the "oneWire" bus and an Arduino Promini. The first and the third are hardware additions to the Raspberry, while the oneWire bus is a Raspberry *native* option: loaded the driver, you can enable the pins of the GPIO connector that you deem appropriate to implement it.

Configuring the Raspberry: Depending on the version of the Raspberry you have and the Raspbian *distribution*, you will have your Raspberry configured for the i2c bus (necessary for the ADS1115 to work) and/or oneWire (necessary for the bus of the same name) and/or serial communication (necessary for the Arduino). See Annex II, “Configure your Raspbian/ Raspberry” for details.

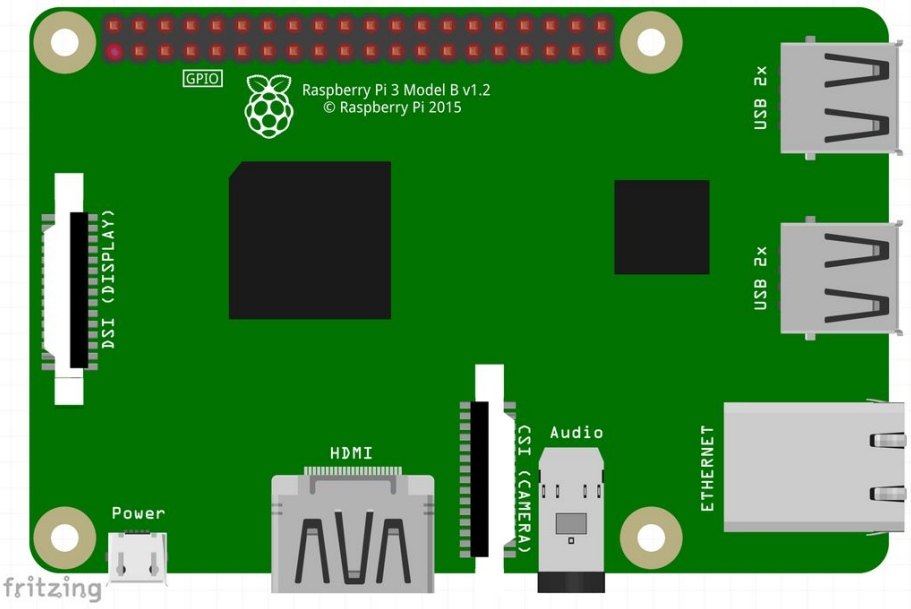


Figure 1. Raspberry 3B.

Yet, the blog is not limited to this: although the library is actually developed -as it is written- as outlined in the previous paragraph, there is another objective as important as the stated one, at least for who wants to learn some analysis and OOP programming. Let’s hope that the trip, encompassing the needs, analysis, problem decomposition into classes and the discussion of the final pattern of the library, is as useful as its -trip’s one- destination, which is the mere use of the library.

1.1.1.- OOP design pattern.

To focus the question: when addressing an OOP development analysis, one of the main issues is the *architecture* of the system, its pattern; i.e.: How do I find the classes that will be instantiated to give rise to the objects that make up my program?. How much should I crumble -granularity- the problem?. How many classes?. What are going to be the relationships among them? And the objects among themselves? How is it documented?

The specific matter that is addressed in this blog -the library itself- is quite simple: you can solve it with hardly a dozen classes (as we will see later). Its best feature is that it is fully developed (that the library code is available https://github.com/Clave-EIDEAnalog/EIDEAnalog -and that it works!). The reader, whose knees will probably tremble sometimes, is at least sure to come to an end.

In other words, the best virtue of this blog is that it is a fully developed example that does not limit to describe the basics of the OOP analysis, a complex issue that requires much more than this blog. If you want to follow it -the blog- please keep this in mind: it is a developed (programmed, working) example. Nothing more, nothing less.

1.1.2.- Data acquisition systems. A/D conversion.

If it is a matter of doing OOP pedagogy, one could have chosen a subject other than A/D (analog to digital) conversion . What does this have -that others do not- to have been chosen as the common thread of the blog?

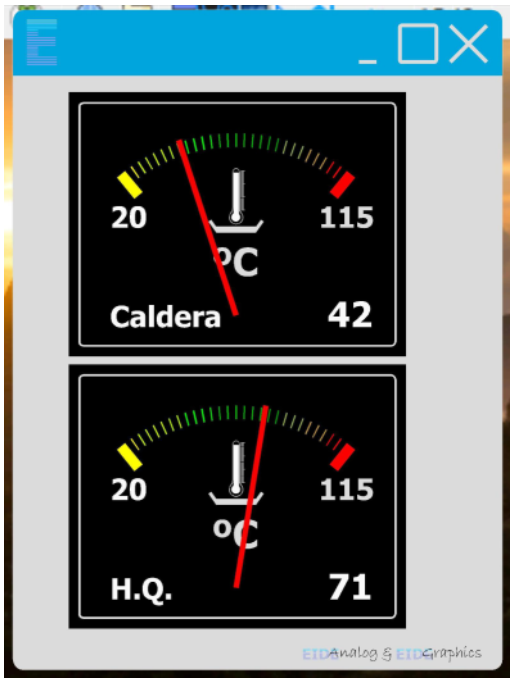


Figure 2. Raspberry data acquisition system.

That the blog authors know it well (and are perfectly aware that choosing a library to "acquire physical parameters", as stated in the third line of the first point of this blog, will

take away a good part of potential followers who, a different topic chosen, perhaps would have been encouraged to follow it).

On the other hand, may be that the matter also is attractive, as the result is connecting sensors to a computer (an issue that, in many ways, can be considered eccentric). This is an issue of interest to a good number of activities, both in the field of industry and training, and in this - that of training – to all the engineering and degrees in physical or chemical sciences. At the end of the day, and except for the subtleties of the A/D conversion device itself -which are specific to this discipline- the remaining is not much more than algebra and binary logic; once you are satisfied with the purpose of what you are doing -the library-, the way of doing it -analysis- is valid for similar problems, or of a similar approaches.

1.1.3.- Blog development. Pedagogics.

We have been careful in the writing of the blog, using a pedagogical style. It is not just a matter of writing some classes so that they can be incorporated into a project by a “copy-paste” (yet this can be done without a problem - and it works). The main goal of the blog is, as mentioned, to help anyone who wants to analyze a case on how to approach the analysis and development of a library using OOP techniques.

1.1.4.- Conclusions. Goals.

By the end of the blog you should have acquired certain skills regarding OOP (see 2.5.3, “OOP. Conclusion."). In addition you will learn:

- Computer data acquisition (basic concepts). ADC (Analog to digital converters).
- User level Raspberry management. (Very) basic concepts of Linux.
- (Rudiments of) operation of oneWire, i2c buses and serial communication (UART).
- Functioning of some commercial (and popular) sensors.
- Thermistors. NTC. Tabulated sensors.

1.2.- Basic knowledge to follow the blog. Prerequisites.

Blog authors have experienced more than once manuals that begin -enthusiastically- explaining that to use the computer mouse you have to operate it with your hand, move it across the pad and press the left button to select or the right one for options (and that can be configured the opposite for left-handed); inevitably the manual -or whatever- abandons this level of detail on the second paragraph to, much worse, give indecipherable instructions just three or four paragraphs later (each of them -of the indecipherable instructions-, deserving for themselves a manual).

We will try not to repeat this error; It would be delusional to think that in order to follow this blog you do not need any prior knowledge of the subjects being addressed. Whoever that wants to follow it must have a *user knowledge* (whatever this means) of computing, personal computers. As we use a Raspberry as hardware you should familiarize with it (of course, whenever the Raspberry is referred to, we are also including the Linux operating system).

Figure 3. ADS1115 tied to a Raspberry.

Nor you should get disappointed for getting your hands a little dirty: whatever the option you choose to implement the blog content you will have to connect something to the Raspberry -see Figure 3 above- and tighten some screws (terminals). We encourage you to connect things to each other: the Raspberry with the A/D converter, the probes to the converter, the Raspberry's power supply (it's like the one of a mobile phone), ... nothing too complicated.

Let’s go step by step.

1.2.1.- Electronics. A/D converters. Binary numbers.

You should know how to use a polymeter; it is worth knowing how to measure DC voltages. A review on Ohm's law won't hurt (although it is not strictly needed for the blog).

Figure 4. Polymeter.

The ADC (analog to digital conversion) issue is a bit trickier.

When it comes to acquiring values *quickly*, the A/D conversion issue is truly a challenging one. *Quickly* means thousands of times per second -or more; or much more- which is the case, for example, on an audio studio to capture music without losing quality (44,100 times per second); similar speeds are required in many problems in the industry of all kinds, in research, telecommunications, image (much more, this case).

This is not this blog case; it is enough understanding that it is all about converting real world parameters to numbers much more calmly. Although we will have to delve into how the A/D converter is controlled, this is a more *administrative* than an engineering problem, as we will see when it comes to it.

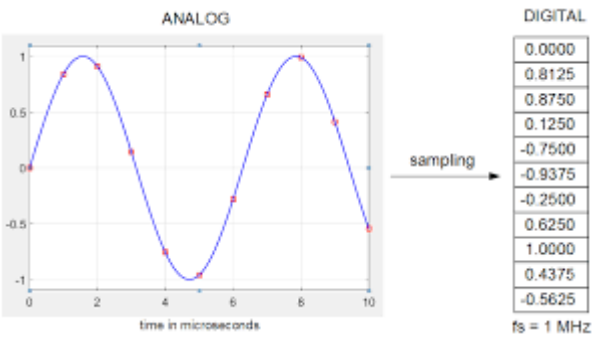


Figure 5. A/D conversion.

(As some WEB page explains, you sitting next to a mercury column thermometer taking notes every, say, 10 seconds, becomes an analog-to-digital converter: you are transforming -and recording- analog values to numbers).

The point is that the computer works with the base 2 numbering system (the -in-famous 1010001010 ...), which in this blog becomes the main scientific problem (at least different from the basic one: the analysis in OOP). If you already master this matter, congratulations, please go to the next paragraph; should you not, you have to make some investigation: entries of wikipedia for "*binary*" and "*bitwise*" are a good start. You also have to know what the hell the "two's complement" format is.

A basic knowledge of the terms *byte* and *word* are necessary too.

Incidentally, the hexadecimal numbering system is also mentioned; take a look at it on wikipedia.

1.2.2.- Raspberry.

Everything you need to know about the Raspberry hardware, at least to start with, is 1) how to connect it and 2) how to configure the µSD (microSD). There are literally thousands of WEBs that explain how to connect it; even the accompanying leaflet includes it.

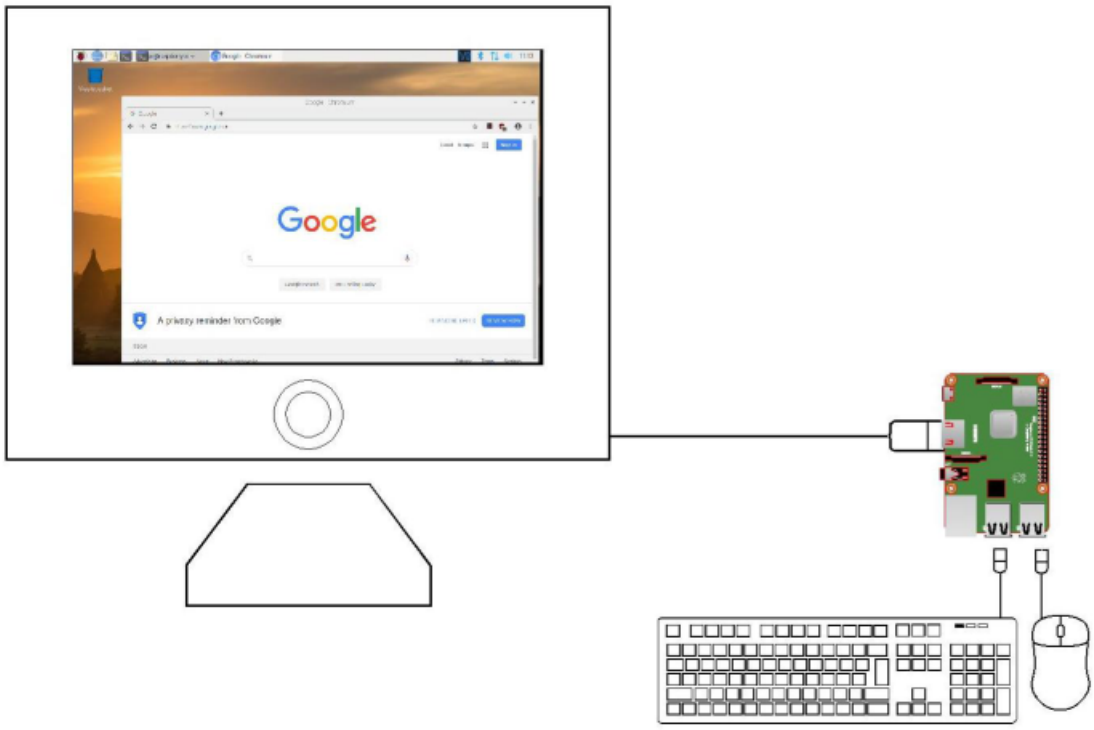


Figure 6. A Raspberry acting as a laptop.

(The easy way to get started is to connect an HDMI monitor to the video port, a keyboard and a USB mouse to two of the 4 USB ports it has, insert a properly configured µSD memory card and power it all.).

The µSD thing is a bit harder: see Annex II, “Raspbian Configuration” for more information.

Although as we present the different subjects they will be explained in detail, it does not hurt finding out what a oneWire, an i2c and a serial communication buses are. Take a look at the wikipedia entries for “oneWire”, “i2c” and “Serial Communication”.

1.2.3.- Linux.

The Raspberry has the enormous advantage of working with Linux ... should you not know Linux you miss it. But do not panic, it is not necessary to become an expert to use it, and using the graphical interface is quite simple to start with.

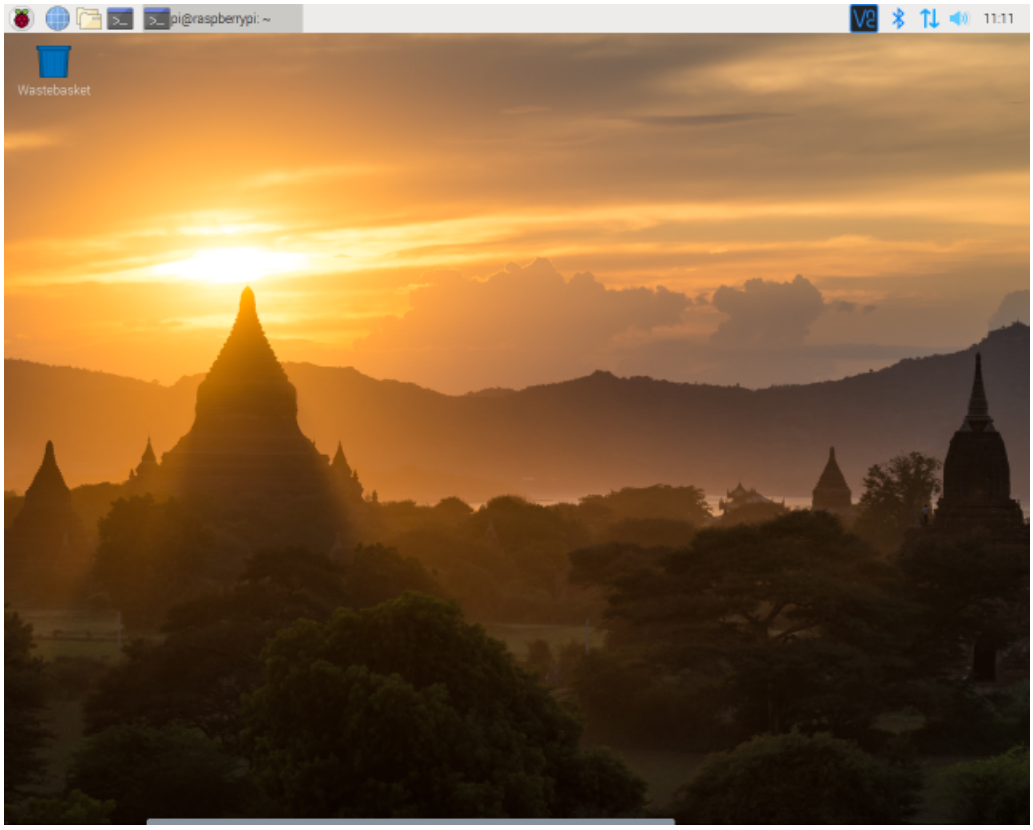


Figure 7. Linux interface (Raspbian).

To following the blog, you don't have to know much about Linux: should you have a well configured *distribution*, a user knowledge should be enough to start (see annex II, “Raspbian configuration”).

As in the precedent paragraph, if you manage with Linux, congratulations, please go to the next paragraph.

Else, we recommend you to learn a bit:

- It is in your best interest to read something to at least familiarize yourself with the terms ("Debian", "Jessie", "Raspbian", "Noobs", "Ubuntu", ... don't be alarmed, they are very much the same). Read the article from wikipedia for “Raspbian” (be patient: you won't understand many things at first, but you have to get a bit comfortable with them); Raspberry's website (“www.raspberry.org”) is not the best in the world, but having a look at it is a good idea either.

- “Terminal": the term refers to the use of the operating system (Linux, in this case, although it can be applied to any O.S.) by means of a "command line". The elders in the city would remember the black screen of MS-DOS prior to Windows: that is the

"terminal " (and, by the way, a few MS-DOS commands are copied from Linux; they are the same).

- It is quite convenient that you know some commands to configure Raspberry things (again the reader is referred to Annex II, “Raspbian Configuration”). See
  - https://maker.pro/linux/tutorial/basic-linux-commands-for-beginners
  - https://www.digitalocean.com/community/tutorials/an-introduction-to-linux-basics
  - https://www.freecodecamp.org/news/the-best-linux-tutorials/
  - https://www.tutorialspoint.com/unix/
  - The book “The Linux Command Line: A Complete Introduction. William E. Shotts Jr ” is excellent.

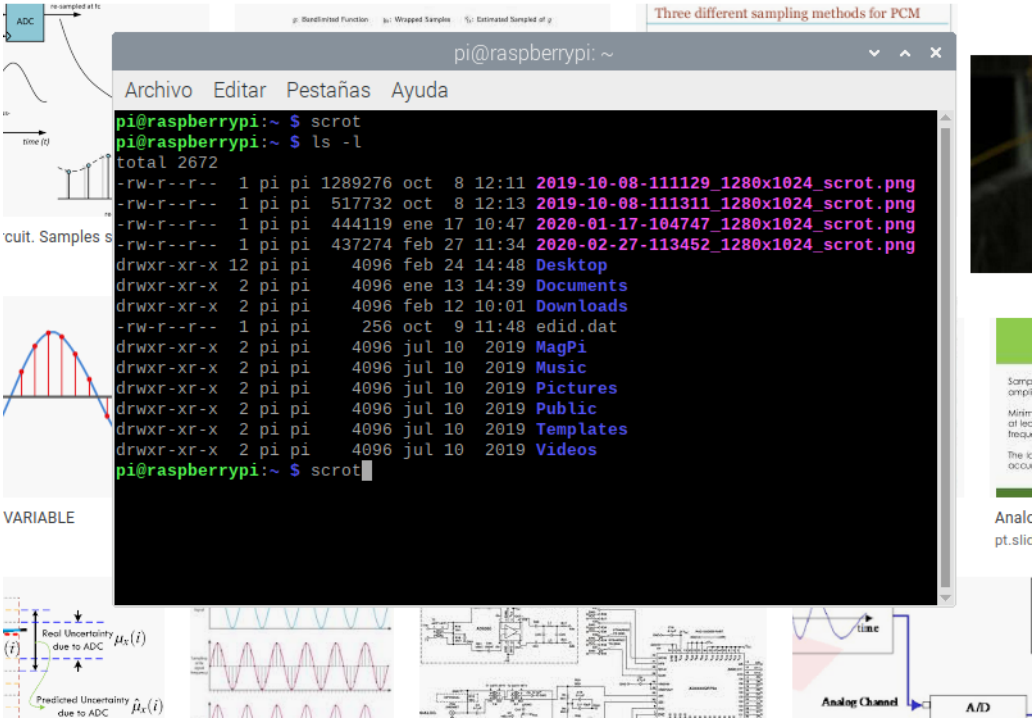


Figure 7. Linux terminal session (Raspbian).

Backup copies: One of the main advantages of Linux is the ability of recovering from the disaster of the loss of the S.O. itself. Using *another* operating system, the hard drive failure leaves you out in the open. In the case of the Raspberry, the equivalent would be the failure of the µSD (or the Raspberry itself, the worst disaster); The good -excellent- new is that it is possible to have a copy of the µSD simply by using another µSD. If you are going to mess around with the Raspberry, which we recommend, it is convenient that you have an updated backup by making periodic copies from one µSD onto the other: a few dollars in exchange of a priceless tranquility.

1.2.4.- Python. OOP.

This blog is not a Python tutorial (nor an OOP one). To follow it without too many shocks -without having to return frequently to more basic issues than those required to understand what is being developed- it is necessary to have a reasonable knowledge of Python. And you have to have written half a dozen simple Python programs.

(The term "reasonable" in the previous paragraph is used with some bitterness because, who is able to say what a *reasonable* knowledge of Python is?).

Python is one of the easiest (the easiest?) programming languages to learn, and it is extremely powerful. And it is not just that its syntax allows coding in a way that summarizes in a single line of, say, 30 or 40 characters what using other languages requires many lines: Python has implemented as native all the instructions and architecture of any other modern language, and you have to take into account what is added by the libraries included in the normal download. It also has structures -dictionaries, for instance- and/or novelties -class attributes- that at the time, and still today, makes it different from the closer environment.

Python is, in summary, a world to explore. So let's see who is the brave woman or man who defines what a *reasonable* knowledge of Python is.

So we will proceed by extension: let’s take the Python WEB reference tutorial (“https://docs.Python.org/3/tutorial”) as a guide. You need to know:

- How to create and manage numeric variables, their types -int(eger), float, ..- and formats -decimal, hexadecimal, binary, ...-. All the arithmetic and logical operations. You need to understand what “None” is.

- How to create and manage strings (the basics: what a string is).

- "Flow control". The above mentioned Python tutorial is not among the best of the foundation (“Python Software Foundation”): the first mention it makes on this concern -flow control of a program- in point 3.2 is undoubtedly unfortunate: one cannot understand why the –very first- example is implemented with a *while* without any previous anesthesia. To follow the blog smoothly go to point “4”: you have to understand all the contents in chapters 4.1 to 4.6, all.

- Lists, tuples and dictionaries: in depth. You should be familiar with all the content of tutorial points 5.1.1, 5.1.2. and 5.5. Along the library they are used a lot to iterate: please study in depth the content of point 5.6 and review the point 4.2.

- Modules: you need to understand what the "import" instruction is for and what is meant by "standard modules" (6.2).

- **Errors and exceptions**. They deserve their own comment: the implementation of a library that is worth of that name should not be made available to potential users without it having a minimum of *protection* against the mistakes that will surely be made when using it (and the ones that, undoubtedly, will be utterly hidden in the code of the library itself). The counterpart is that the use of error handling clauses

introduces a new complication in the ‘aspect’ of the code, which, at least at first glance, causes the reject of rookies.

A necessary compromise has been reached: the first version of the library (the one that is discussed in depth into this blog and can be downloaded from https://github.com/Clave-EIDEAnalog/EIDEAnalog), has no *error handling* code. A ‘Pro’ version that will have it (open source too) is in preparation. Have a look on chapter 8 of the tutorial, though.

- Classes. Objects: This is the moment of the truth. The terms *class* and *object* should have not secrets for you; and we are off to a bad start, however, if you frown as soon as the first *self* appears or you panic when you see a *__init__* coming floating in the code. Along the blog we neither take everything for granted (related to the concepts of class, object, etc.) nor can we make pedagogy about it with the basic concepts of the subject (which, on the other hand, is done -pedagogy- when the things get complicated: abstraction, encapsulation, inheritance, polymorphism, patterns). On the other hand, the didactic standard of the chapter 9 of the above mentioned tutorial (“https://docs.Python.org/3/tutorial”) leaves, again, something to be desired. And this is not an easy matter (neither the basic question of the OOP technique nor how to start with it). Please, find below a couple of WEB sites that you should study if you want to follow the blog contents without shocks.
    - https://www.programiz.com/Python-programming
    - https://realpython.com

Python download, install and configuration: If you are not familiar with Linux, it is possible that starting up Python on Raspberry involves some difficulty. Although the authors of the blog are unrepentant supporters of Linux -and, therefore, self-conscious detractors of the "other" operating system-, it would be an excess to suggest to the blog follower to study it thoroughly -Linux- as a previous step to start with the blog . The Raspberry startup and use is not complicated at all if you have a good *distribution* (the jargon to refer to the version of the S.O.). There are many vendors on the Internet who can supply you with a properly configured µSD. Linux has a graphical interface similar to that of Windows. See Annex II, “Raspbian Configuration” for complete information.

1.2.5.- Other prerequisites.

- File types: You have to be able to identify and modify a text file (windows *notepad*; Linux *leafpad*).
- Needless to say that you have to know what the ASCII code is.
- Have a look into the wikipedia entrances for NTC and thermistor.

2.- Project: ADC library. Specs & tools..

If after such a lengthy advancements, so many reading recommendations, so many warnings and even threats, if, after all this stuff, you have come this far, congratulations: it is time, finally, to get into the matter.

Let's see, now in detail, what we are going to do and what tools we have to accomplish it.

2.1.- Specs.

We face this project::

1. The goal is developing a library in Python 3 to use a Raspberry as a data acquisition system (conversion of analog to digital variables). The library must allow to select the ADC from three *native* alternatives (alternatively or simultaneously): 1/ the Raspberry Pi native oneWire bus, 2/ A “chinese” board -there is no explicit manufacturer- that mounts an ADS1115, and 3/ an arduino ProMini. The measurements taken by the ADC will be converted to a readable format.

2. The library has to solve connecting several sensor alternatives: those designed for the oneWire bus (see 2.3.1, “oneWire bus”) and *generic* sensors for temperature, pressure, distance, sound, etc. -those ones that convert the measured parameter to volts- in the case of the other two options (ADS1115 and arduino). While to add a new ADC to the library involves writing a new class, new sensor types would be added without modifying the library code. It should even be possible to do so for tabulated (non-linear) sensors.

3. Library should allow including new types of ADC converters easily. Classes that implement the control of *native* ADCs will be designed in such a way -interface- that it can be replicated for any ADC whose handling is standard. This way, the client application -the one of the user of the library- will not need to be modified even if a new type of ADC is added.

2.2.- Raspberry.

The Raspberry is a computer (microcomputer, nanocomputer, pocket computer, as you like) credit card sized and fully operational (applications compatible with -Microsoft- Office, Internet browsing, multimedia content playback, etc.). And last but not least, the latest model -perfectly suited for the purposes of this blog- costs less than 40€. Add a power supply (the official one costs 12€, there exist great ones for half the price), the µSD card and you have a full-fledged computer. A bargain for the kids and for yourself. Please visit the WEB (“https://www.raspberrypi.org/”) for complete information.

Models from the 3B upwards have bluetooth and wifi too. All of them have, what is of great importance for the purposes of this blog, a 40-pin connector to which you can connect things that you cannot connect to a desktop computer or a conventional laptop.

Raspberry was conceived for students, proposing Python as a programming language, which is already pre-installed in Raspbian.

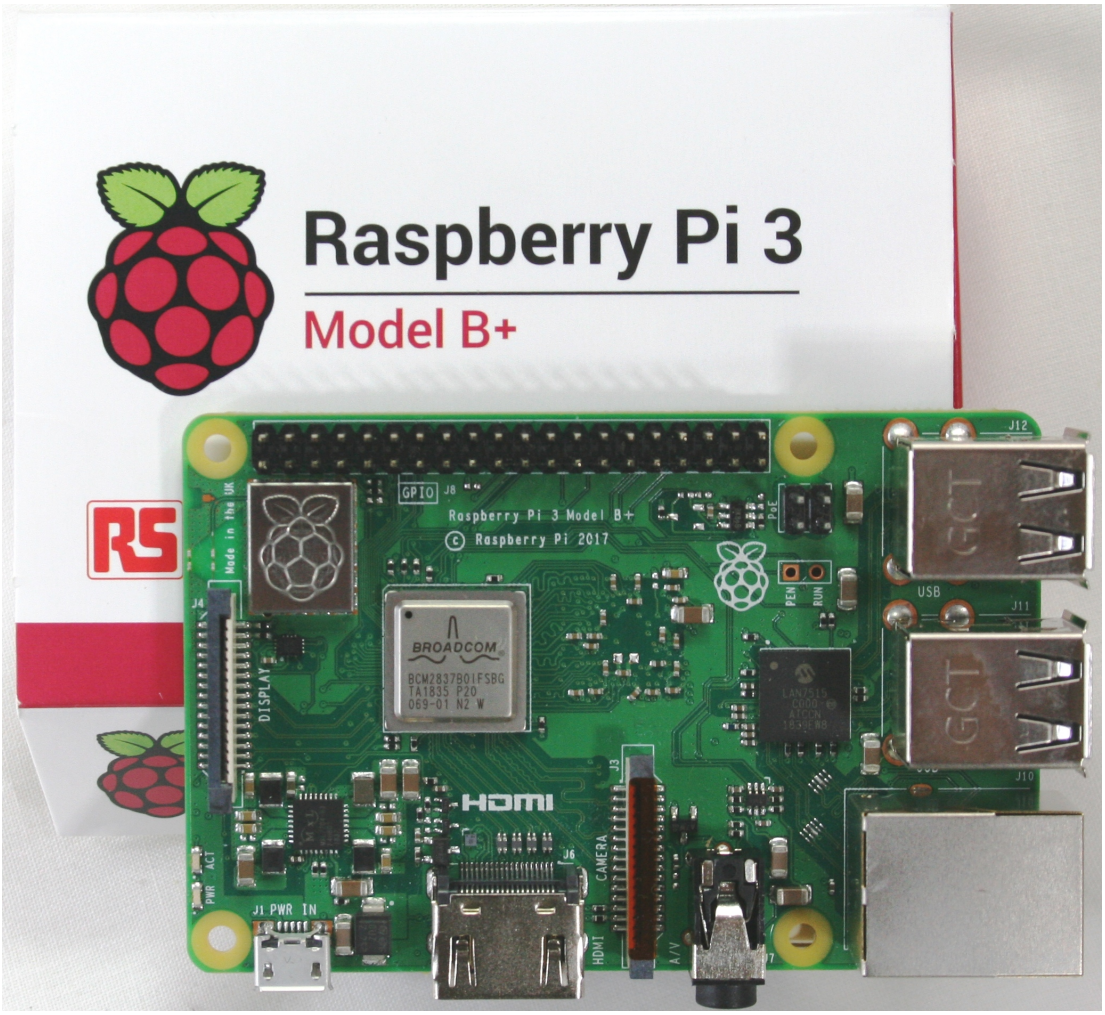


Figure 8. Raspberry Pi 3B +

Praises are over; you can find hundreds in the Internet, along with thousands and thousands of WEBs in which the product is described and all the other stuff that can be done using it. Finally, to state that any model from the "2" -Raspberry Pi 2- is more than enough for this project.

2.3.- ADC. Analog to digital converters.

An ADC converter converts -input- volts into a number you can operate with.

About the terms **"ADC" and "bus"**: These two terms are neither interchangeable nor about the same thing: while the first refers to a "chip" (or electronic board, or module), the second refers to the way two or more chips, either in a small system like the one in this project or in much more complex equipment, communicate among them. In this project -EIDEAnalog- is the case for both the *conventional* ADC chips -the ADS1115- and the small ADC converters that are integrated into the individual devices that are connected, in this case, to the Raspberry by means of a oneWire "bus" (the DS18B20 thermometers). This bus -the oneWire- thus solves the two problems simultaneously: first the sensors give the conversion already done and the bus -this is its mission- transmits it to the Raspberry; as you can see, the whole is a mixture of sensors, digital analog converter (s) and a communication bus that, from the point of view of the library we are conceiving, will be controlled by a single class - "oneWire" -.

The ADS1115 connects to the Raspberry through the i2c "bus" and the Arduino by means of a conventional "serial" bus -RS232 (buses that, on the other hand, allow you to connect many kinds of things to the Raspberry: displays, video cameras, keyboards, printers, ...).

Being the object of this project the design of classes that address all of the aforementioned possibilities in an homogeneous way, (and -being- the access to the bus and the capture of sensor data intertwined from a logical point of view), when referring to the classes that control the bus and ADC chips we will use both terms almost interchangeably (and even the -nonexistent- word “ADCBus”, which is not rigorous at all).

2.3.1.- oneWire bus.

The oneWire bus was invented by Dallas Semiconductor Corp. and is a protocol (a set of specs for both hardware and software) that eases chips connection (with each other). It has the advantage of just needing one wire for data transmission (two, in fact, the other one is "earth" or "0 volt", as you wish) for all connected devices (in number almost unlimited; of course there are restrictions due to the power supply, lengths, interference, physical space, ...). The transmission speed is not very high 16 kbits / s (about 1600 characters per second) although much more than enough for our project.

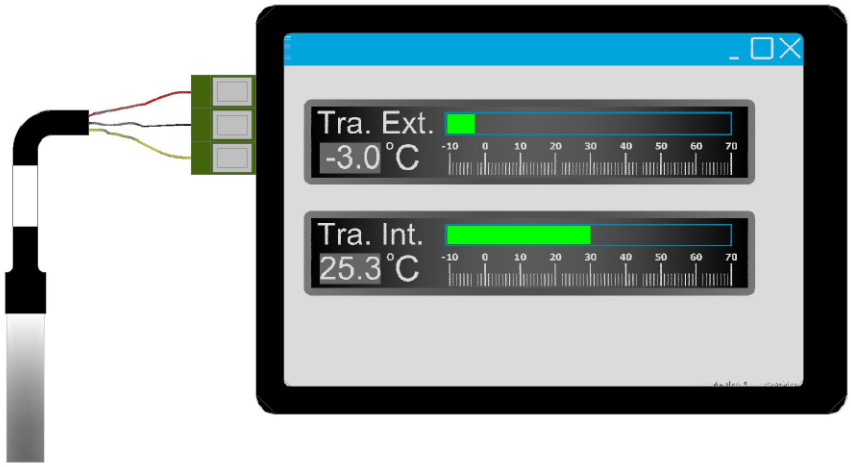


Figure 9. Commercial kit for temperature (Raspberry + oneWire)

The Raspberry acts as a oneWire bus “master” (the name says it all) without the need of any other hardware; It is, therefore, free (see Annex II, “Raspbian Configuration”). Several kinds of devices can be connected to the bus, temperature sensors among them. Along the blog we will use the DS18B20 for the examples; It is a temperature sensor with a microchip that includes an ADC converter and the necessary electronics to communicate -through the bus-, everything conveniently encapsulated. Each of the thermometers (and any other "slave" devices connected to the bus) have a single identification, that consists in a 12-character hexadecimal code. DS18B20 thermometers cost is around two dollars per unit.

Figure 10. Temperature probe DS18B20. Water proof finishing.

The DS18B20 arrangement needs a third cable and a small resistance to supply the sensors (See figure 11).

The standard Raspberry driver -software- to manage the oneWire bus at its lower level works in a quite peculiar way: when starting the Raspberry the driver makes a polling of the bus detecting every device that is connected to it; it creates a folder for each one it finds (See 6.1, “oneWire Class”). Then, to read the sensor, what you have to do is to look inside the text file "w1_slave" on the correspondent folder, this one identified by the sensor hexadecimal code.

This behavior is convenient, since without the need for any specific software you may know the temperature by just opening the file. The problem is that this is quite a slow process, at least in terms of what is usual for ADC devices, even the simple ones: as average, the time it takes to get the reading is about 0.9 seconds per sensor. Despite this, it is enough for our project, since it is about discussing different solutions; this is one of them.

(Actually the probe, including the accompanying electronics, takes a fraction of those 0.9 s. It is the subsequent reading of the file that makes the complete cycle so slow).

The oneWire bus is implemented by using three of the Raspberry Pi GPIO header.

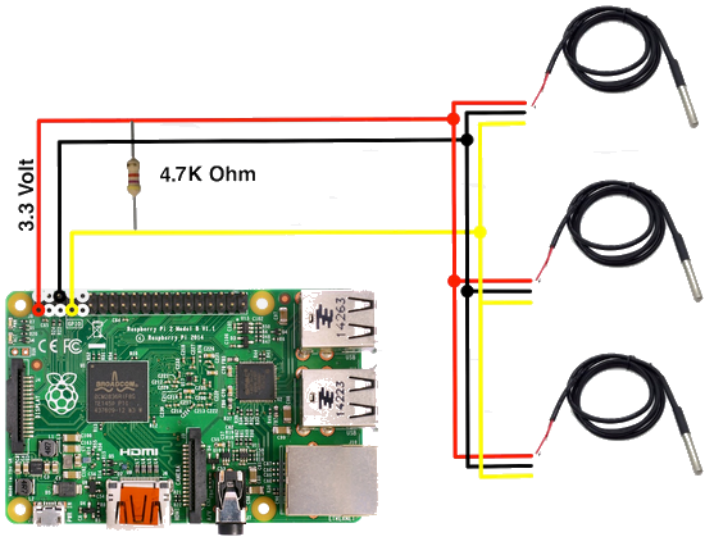


Figure 11. oneWire bus connection..

For the shake of hardware connectivity, class oneWire in this library activates a generic GPIO pin to supply 3.3 V to the oneWire bus. The most usual bus implementation -Figure 11 above- uses some pins that make the connection of the most common 3.5" LCD screens -Figure 9- impossible. A different pin may also be activated for the bus itself (again the *usual* pin is in the LCD screen header area. See annex II, "Raspbian configuration"). Discard all this information in case you are not using a LCD: you may use the conventional layout.

2.3.2.- ADS1115.

The ADS1115 is a 16-bit ADC designed by "Texas Instruments". It has four channels that may give a direct reading (4, therefore -direct readings-) or differential (2, maximum; mixed configurations can also be used -two direct + one differential-; in fact, the mode selection is made on the go, so the choice, except for the connection of the probes which is supposed to be permanent, can be changed even when working). The ADS1115 connects to Raspberry via i2c bus.

The chip has 6 programmable reference voltages (or "gains", to adapt itself to the range of the sensor) and 8 different sampling rates, also programmable (actually the sampling rate -the rate at which it samples- is always the same, a little bit more than 1,000 times per second; what happens is that the chip averages all the samples it has taken in the interval

correspondent to the rate you program: if programmed for 475 samples per second (SPS) it will average 2 samples, if asked for 250 four samples, and so on. -up to 8 SPS-. If you ask for 860 it will result in a single sample; the more samples averaged the more accurate the final result). The ADS1115 may also compare the measured value with a programmable threshold, providing alarms in case the value goes out of the predefined value (option that we will not use in this blog).

Figure 12. ADS1115

Direct use of the chip requires a quite skilled hardware job: the chip is 3 x 3 mm and has 10 pins, so the best is to look for an assembled card that mounts it. The most popular is the *Sparkfun* one, but we have chosen a more than robust card made in China that has an header to connect directly to the Raspberry and screw terminals, so that you do not have to solder anything to use it (the fact that it is chinese is an assumption: beyond the reference "HW-660" that it shows in it has no other identification). It also has a built-in thermistor (NTC) that can be connected to channel one by a jumper -again without soldering anything-, and that will be handy to check the operation of the library with non-linear probes (see 5.2.3 .- "'tabulated' calculation agent"). The HW-660 can be purchased in eBay or Amazon for 12€ (even less).

Figure 13. ADS1115 boards. Sparkfun (left) and HW-660

A couple of paragraphs above the option -need- to program the ADS1115 has been mentioned. The right way is to have a look on the chip datasheet: you can consult and download it from “www.ti.com/lit/ds/symlink/ads1115.pdf”. This is the proper way, we say. Don't worry if it looks cumbersome, in paragraph 4.1., “Class ADS1115” the data sheet is

sufficiently crumbled to justify the elaboration of the corresponding class; It is much better to start there.

A final issue: the Raspberry standard configuration of the i2c bus works at 100 kbits/s; at this speed each sample takes -seems to take- (starting at the trigger order, plus the time it actually takes the chip to convert plus the the time that it takes the reception of the information) much more than what it takes just converting (just over a couple of milliseconds for transmission and about 1.0 millisecond for conversion at 860 SPS). See exercise at the end of chapter 4.1.10 and annex II, “Raspbian configuration”.

2.3.3.- Arduino ProMini.

Arduino is an Italian company that had the great idea of taking some Atmel (A North American company now owned by “Microchip Technology Inc”, also a USA company) microcontrollers and mounting them on boards so that they could be used easily; something quite similar to what was done with the ADS1115 (see previous point) by Sparkfun -and also by the unknown Chinese manufacturer- from the point of view of hardware and, somewhat more difficult to explain -what arduino did-, from the point of view of the software. In summary, arduino managed to have their boards, and much especially the arduino "UNO", a kind of standard in the microelectronic industry. The arduino ProMini that we will use in this blog is a minimal size version of the arduino UNO.

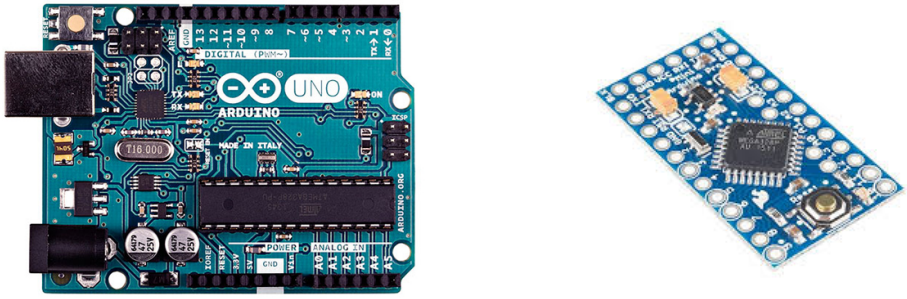


Figure 14. Arduino UNO (left) and Promini.

The Arduino boards -the Atmel microcontroller- are not -only- A/D converters, although they have one built-in (with eight conversion channels). They are, in that they resemble the Raspberry, miniature computers, although the Arduino is focused to the connection with the real world (say what is now known as IoT, although when Arduino was founded it did not exist -the name, the thing itself did-). Let's say that you cannot connect the Arduino to an HDMI display, or directly to a USB memory stick, for example (or not, at least, to basic models).

To use an Arduino it needs its own program, so you have to look for one that is already (programmed) to use the -its- ADC and you need some form of communication also programmed (all Arduinos have implemented the oneWire bus -as in the Raspberry you simply need a driver-, the i2c -the same- and several serial communication channels).

There is a Spanish manufacturer that has what is needed: a ProMini mounted on a board that, in turn, is prepared to be mounted directly on the Raspberry. Furthermore, although it has its own programming -see 4.3, “Arduino class”; this company uses it as part of a “data logger” - it behaves as intended: it converts the 8 analog channels (10 bits resolution) and sends the information packed by the serial channel on demand. We will use it as another option in the library.

Figure 15. Promini Raspberry hat.

2.4.- Programming language. Python. IDE.

Python's implementation of everything related to OOP is up to date. Add -to Python, no to OOP programming techniques, this is quite a different matter; see next paragraph- a more than generous learning curve and we have the perfect tool.

(Well, may be it is not perfect: Python is an interpreted -not compiled- language, so it is relatively slow. "Slow" means that, for example, a Python program that calculates the square of the first 1,000,000 numbers takes, on a veteran desktop computer, 0.23 seconds. That is 0.23 microseconds on average per calculation. There will be plenty of people who will cross out this ridiculous figure when compared to the speed of an iPad. By the way, on a Raspberry Pi 2 it takes 1.2 microseconds: it is five times slower!; on a Raspberry 3 B + it takes around 1.0 microseconds).

In what concerns the development environment (the IDE; *Integrated Development Environment*) to use there are discrepancies. The one that includes -it used to include, keep reading- by default the language, that is, what appears -it used to appear- when you select the option "Python 3" (or "2") in "Programming" -or a file " .py ”- on the Raspberry is -it used to be- a good one. In the latest versions -distributions- of Raspbian ("Raspbian Buster" at the end of 2019) this has been replaced -damn!- by a certain "Thonny Python IDE". This environment is very likely to improve the old one, but when one gets older becomes reluctant to change (and much more if one is obliged to).

Should you are making your *debut* -following this blog- with Python, try that *Thonny:* sure it works well. If you are disobedient and prefer the old environment you have to mess around a bit: see Annex II, “Raspbian Configuration” to install the old IDLE.

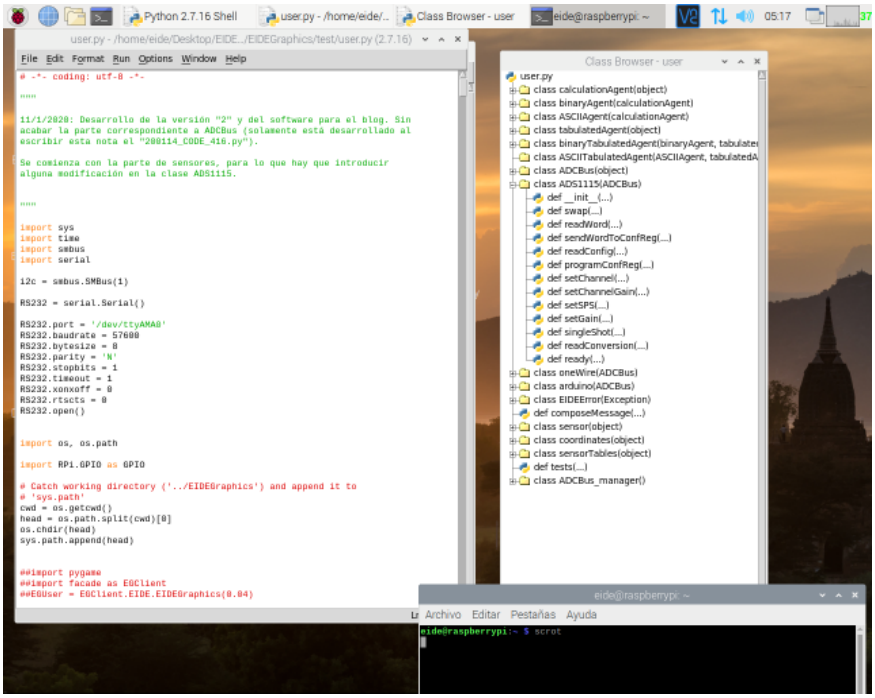


Figure 16. (Old) Python IDLE.

2.5.- Design using OOP.

2.5.1.- Abstract.

OOP is not an easy matter.

Procedural programming (for the sake of simplicity, the one used without resorting to OOP) has a generous learning curve: one knows the "for- (next)", the "if- (then) – else" (what is in parentheses is because it is not used in Python) and having an idea of what a *routine* is (*function* in Python) you can write programs to solve quite complex issues; there exist code written with these resources and a little else that would make the Apollo X computer programmers green with envy.

Yet, this type of programming has drawbacks, being the most remarkable one its poor *reusability*.

It is all about the chances of reusing the code -the whole or a part- previously written to be used for a certain environment into another. The concern was for a long time (let's say from the time first programs were written -in the 1950s - till the beginning of the 1990s) a very controversial one -and poorly solved. There was no way for the “reuse” to be painless. To a greater or lesser extent, the copy-paste did not work: best case -and always based on military discipline- routines could be reused almost without modification, but the certainty of not having maintenance problems afterwards was minimal. Any improvement -this is another matter, that of *improvements*, which deserves another blog- in the original routine always required the user -customer- some effort to readapt the code. Most occasions, the *built-in* routine ended up being completely rewritten and the intended time saving turned into a loss: there are many examples; anyone who has developed software for someone else knows perfectly well what we are talking about.

The OOP technique solves this and other problems related to software reusability and maintenance. The code with the slightest probability of being reused is implemented by *classes* whose subsequent *instantiation* enables the code use in an opaque way (encapsulation of code and attributes). The classes, in turn become part of libraries that help to solve one or more problems in a certain domain -area- (this last statement being a simplification).

2.5.2.- Example.

You are taking your first steps in Python. You want to program a basic calculator: by entering two numbers and an operation ("+", "-", "*" or "/") the operation is performed and the result is displayed.

Leave aside how you coded it (the calculation once the three elements have been entered into the computer). The point is that the basic -*native*- tools of Python to achieve a certain interactivity are really poor: the “input ()” function is quite primitive: your brand new calculator deserves something *à la mode*. Something like this:

Figure 17. Using wxPython .

Here is when the OOP becomes handy. If there is a library that covers this it will suffice (assuming that everything they say about libraries and OOP is true -it is), by using it. You mess around the Internet, social networks, ask someone you know he/she masters this stuff to finally get to the conclusion that the right thing is using "wxPython". (By the way, there is another widely used GUI library, "TkInter", which was the first for Python. There is a certain rivalry between the two -among their followers-: the authors of the blog opt for wxPython).

wxPython has a class *TextCtrl* that instantiated gives raise to an object which is the small window where the number "A" is to be written. The corresponding code would look something like this:

```
A = wx.TextCtrl(self.panel, -1, "", style = wx.PROCESS_ENTER)
```

(Do not run away: we have already told you that the OOP is not easy; by now you just want to make an idea). The other two, for "B" and the result would be:

```
B = wx.TextCtrl(self.panel, -1, "", style = wx.PROCESS_ENTER)
result = wx.TextCtrl(self.panel, -1, "", style = wx.TE_READONLY)
```

Let's say that in a certain *“self.panel”* (that is inside the window where the whole process will take place) there will be three boxes; in two of them you can write (“style = wx.PROCESS_ENTER”) and in another just look (“style = wx.TE_READONLY”; we will leave the “-1” and “” for a better day).

Now we need the labels ("A", "B" and "result"; the above is just to enter and see numbers). You have to write something like this:

```
ALabel = wx.StaticText(self.panel , -1, “A”)
BLabel = wx.StaticText(self.panel , -1, “B”)
ResultLabel = wx.StaticText(self.panel , -1, “Result”)
```

that shows up the labels (we have instantiated the *StaticText* class three times: three labels).

And so on: four instances of *wx.Button* (operations), tell where to put everything on the panel (alternatively it can be done automatically with another object of class *wx.GridBagSizer:* things get complicated), instantiate the *panel* where the whole drama will

take place (*self.panel = wx.Panel (self)*) into the window, which, of course, is an object of another class (¿*Calculator*, perhaps?) which in turn inherits from the *wxFrame* class.

Take a breath, the good new is that you do not have to program anything -anything else than the above- to have a window with a panel inside that, in turn, has buttons, labels, boxes (in which, by the way, you will have the functions of a word processor: insert, delete, etc. You can also move around the entire window with the mouse, write inside the box you have selected, press a button, etcetera). And all interactive: if you have any previous notion of programming you have a good idea of what the effort to get such an interface is. And surely you are starting to be intrigued by the above; else calm, it is a toll (the abstruse of the subject) that is worth it.

2.5.3.- OOP. Conclusions.

There are -the one you are right now reading too- a lot of blogs and WEB sites where you will find a lot of nonsense about the painlessness of the OOP: especially touching are those that introduce the subject with the class *mammal* whose subclasses are *cat* and *dog*, which, in turn, participate from the classes *tail* and *color*; its objects, "Boby" and "Sanson", move the tail (or not). Those of cars and drivers normally have also not waste.

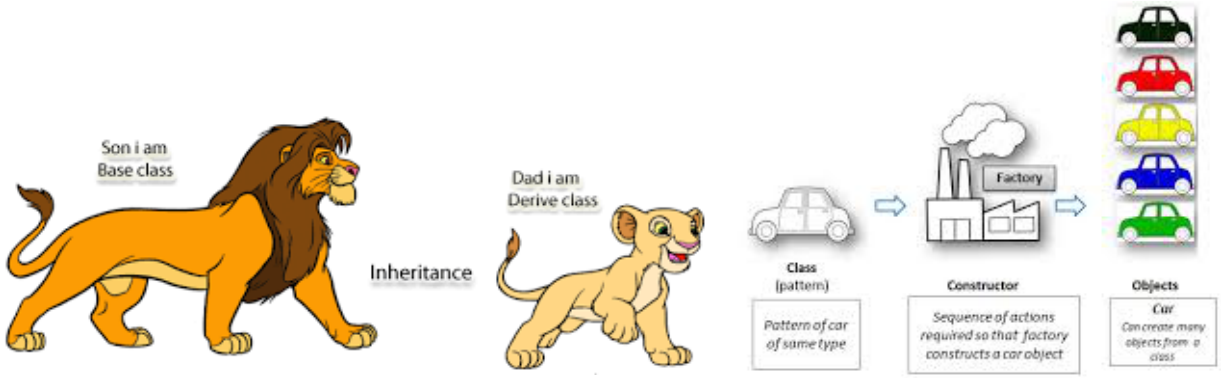


Figure 18. OOP made easy: mammals y cars.

(The point is not that this simple approach they propose being essentially false: it is not. The mistake is trying to explain a zoo with *mammal, dog, cat, Boby* and *Samson* or a Formula 1 race with just *car* and *driver*. There is much -much- more stuff than just that).

The OOP is not a one lesson subject -nor of a quarter, probably - so take it easy. This blog is intended to help you take the second steps (see 1.2.4., “[Basic knowledge to follow the blog. Prerequisites] Python. OOP” for the first ones). By the end of the blog you will be familiar to the following basics of OOP:

- Classes and objects.
  - Class instantiation.

- Class attributes. Methods. Object attributes.
- Inheritance. Multiple inheritance.
- Object composition.
- Project documentation. UML Notation (very basic notions).

- Libraries.
  - Use of. Library client.

3.- Library pattern. First approach.

There is no magic bullet for what we are going to do next paragraphs. Yet, it is the most important part of the project (at least for those who follow it to learn a bit on applications building using OOP).

Let's go for it. The first issue to solve is to fragment the complete task into subtasks that are easier to face than the whole one. We are going to do it using the "object oriented analysis" (so that OOP can be used later).

3.1.- Noun/verb.

We have to delineate the future classes and what *their* objects will be able to do (methods). As a first approach, the "noun/verb" technique is to be used. It helps clearing up the problem.

It is about:

1. Writing down the project spec (we've already done this).

2. Separate names from verbs: names would be the classes and verbs the methods.

The stated spec is:

- The goal is developing a library in Python 3 to use a Raspberry as a data acquisition system (**conversion** of analog to digital variables). The library must allow to **select** the ADC from three *native* alternatives (whether alternatively or simultaneously): 1/ the Raspberry Pi native oneWire bus, 2/ A "chinese" board -there is no explicit manufacturer- that mounts an ADS1115, and 3/ an arduino ProMini. The measurements taken by the ADC will be **converted** to a readable format.

- The library has to help connecting several sensor alternatives: those designed for the oneWire bus (see 2.3.1, "oneWire bus") and *generic* sensors for temperature, pressure, distance, sound, etc. -those ones that **convert** the measured parameter to volts- in the case of the other two options (ADS1115 and arduino). While to add a new ADC to the library involves writing a new class, new sensor types would be added without modifying the library code. It should even be possible to do so for tabulated (non-linear) sensors.

- Library should allow including new types of ADC converters easily. Classes that implement the control of *native* ADCs will be designed in such a way -interface- that it can be replicated for any ADC whose handling is standard. This way, the client application -the one of the user of the library- will not need to be modified even if a new type of ADC is added.

Some nouns (class candidates) stand out:

"Analog variable", "digital variable", "bus", "oneWire", "ADS1115", "arduino", "sensor", "ADC".

Verbs (methods candidates): “convert” (transform) is, undoubtedly, the star.

(Spoken language is not exact -is it computing?-. Some terms may be taken as nouns or verbs: conversion is a noun, but it derives directly from *to convert*; let us be a little ambiguous for the moment).

This “noun/verb” method is not quite scientific; we have to debug the result. Let's go to the next step: you have to sharp your mind because the next matter is a bit more elusive one.

3.2.- Classes as services suppliers.

It is about analyzing the -eventual- classes from the point of view of what they -the objects you get by instantiating them- can offer to the client (user). Classes have to help solving the problem, not complicating it.

Look at it this way: the programming language, Python this case, has what it must have: variables, operators, flow control instructions, error handling, class management, etc. We cannot ask the language is to have instructions for the specific problem: Python has not an instruction "convertChannel" to manage an ADC or "tableLookup" to get a value from a given table.

We are addressing one of the core aspects of the OOP: class methods have to supplement the programming language so that to the client is -almost- as easy to use a Python primitive instruction -”print”, for example- as a *new* one -*convertChannel,* for example-.

In addition, if the classes are conceived in such a way that they are valid for any similar environment -or maybe not so similar-, we are getting closer to the correct way of facing the problem. We get into matter below.

3.2.1.- Bus classes. Sensor class. Conversion.

Classes managing the *ADCbus* devices (oneWire, ADS1115, arduino; -see note at 4.3-) are quite a direct ones. They contain -encapsulate- what is essential for the *ADCbus* to work and little else; they touch the bare metal -the hardware. Taking a preliminary look at points 4 and 6, you can see that a good part of them (especially the ADS1115, which we will use as a standard throughout the blog) are almost the literal transcription of the datasheets. We will also make an effort to keep them -the ADCbus classes- as independent as possible (*loosely coupled*) from the rest of the library.

Much subtler, however, is the issue of sensors and the one of the human readable value calculation (They appear as candidates to a class -sensor- and a method -calculation-; the readout itself will end up being a sensor attribute). As we will see below, the classes related to these topics are much more abstract than the previous -ADCbus- ones.

Except for the oneWire bus, which delivers the sensor reading in an almost a readable format (it gives it directly in ASCII, in centigrade degrees millis), the conventional format for an ADC to give the reading is binary; so it is in the case of the other two ADCs that we use in the blog: in *two's complement* the ADS1115 and *direct binary* in the case of the Arduino. And we need to transform them into human readable format (for example, 00010000 = 32 for *direct binary* -Arduino- and 10000001 = - 127 in two's complement -ADS1115-).

3.2.2.- *readout* method.

To convert the value the ADC gives (remember, normally a binary one), and whoever does it and wherever it is done, the following is a must:

1. The format -*two's complement* or *binary*- is it. Put it another way, what ADC is it coming from.

2. The sensor parameters. The sensor gives a voltage -which is what the ADC converts: a voltage- proportional to what it is measuring: centigrade degrees, for example. It is highly unlikely that the ratio in between the parameter and the voltage is a "1 to 1" one, so the sensor is characterized by its *gain,* which is the ratio in between the units it measures and the volts it gives. Along the blog the LM35 is widely used as a reference sensor: it has a gain of 100 (see 5.1, "Sensors").

   Sometimes (keep seeing 5.1, "Sensors") the sensor has an offset in between its "0" (Volts) and the "0" of what it is measuring: we will call that value *zero.*

   There are sensors whose reading (or whatever they are converting) does not have a linear dependence such as that described in the previous paragraphs (keep seeing 5.1, "Sensors"). This case the conversion –in between what the ADC gives and what we want to see - is *tabulated*.

3. Finally there is a parameter which depends both on the sensor and the ADCbus: the *reference voltage*. It is not about the volts that the sensor withstands; rather it is about how much -voltage- it outputs. This is another element when computing the reading, so it becomes part of the repertoire of things that you need to convert the ADC output to the human readable format. This parameter, along with the gain, the zero and the -eventual- table, will be part of the sensor "library" (see a little later).

Summarizing, to calculate the reading you need the bus data (the data format) and the sensor info (gain, zero, reference voltage and a table -if applicable-); the question is: where -method- is the calculation made?; Which class will hold that method?.

Let's think a bit: it looks like we need a class "sensor" which, at least, encompasses the above list for each one of the sensors connected to the system: 1) assuming that the information for each *type* of sensor -LM35- is centralized -it is-, one sensor object attribute has to be the type -'LM35 ', or whatever it is- and 2) which bus is it connected to -ADS1115, for example-. (This could be done the way around: the ADCbus object has the information of the sensors it is converting; this case it would be -the object of the bus class- responsible for the calculation. It would be a perfectly valid alternative, although, as will be seen in due course, less flexible: you cannot instantiate a sensor - a *real* sensor,

not a *type* of sensor - until you know which bus it is connected to. It is possible, instead, to instantiate a bus without specifying which sensors it has connected: it may not have any).

So the sensor class has to be able of calculating the readout. As a first approach the class will have a method -*readout*- that computes, passed as an argument what the bus gives (binary; ASCII) the human readable value. To accomplish this task all the values listed above should have been passed as arguments at the time of -sensor- instantiation. Let's anticipate, it is a little early for this, that the instantiation will be something like this:

```
extTemperature = sensor('LM35', myBus, 2)
```

(We will come back to the subject in 5.4, "Class sensor. Code. Instantiation"; let's advance that *myBus* is the bus to which the sensor is connected and "2" the ADCbus channel).

Back to the analysis:

A landmark on an OOP analysis -and its subsequent implementation- is the absence of *do case* (or *switch*) clauses. These are at the heart of procedural programming: as the computing of the readout depends on many parameters (remember: type of sensor; ADCbus it is connected to), a procedural programming will select the computing *routine* by *switching* (or *do casing*) by the sensor type and then by the ADCbus type.

For tabulated sensors a table must be added to the above: Next paragraph is copied from 5.3.5. "Multiple inheritance. Classes 'binaryTabulatedAgent' and 'ASCIITabulatedAgent'".

"In a project like the one we are dealing with -and almost always- what happens if there is a tabulated sensor (the NTC of the Chinese board made with the ADS1115 for instance, see figure 19), is that it is connected to a channel of the ADCBus, the latter delivering a reading in a certain format (binary, ASCII, ...) that must be converted to decimal -like that of any other sensor-, then pass the computed value to the "tabulatedAgent". This one looks up into the table to calculate the final reading".

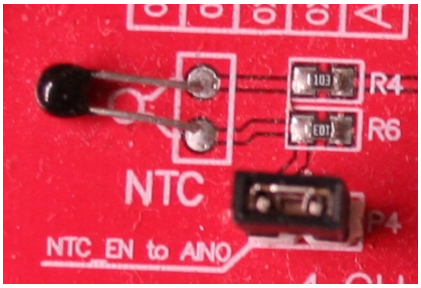


Figure 19. NTC

Should a tabulated sensor be present, the ADCbus value has to be converted as any other sensor: first the reading is converted to decimal and then the final value is extracted from the table.

Again, with procedural programming, this would lead to a "do case" or an "if-then-else" (which should not appear in the OOP either).

(It is beyond this blog scope going deeper into this matter. Let's say that the reason for what non-existence of "switch" clauses is a sign of good OOP analysis is because it is a symptom of little coupling in between classes, a desirable characteristic connected to software reusability).

3.2.3.- *Calculation agent* classes.

The issue of computing the final readout will be solved by devising a family of classes "calculation agents" (we will use the *inheritance* technique to derive some of them from the others) that implement the different calculation algorithms depending on the format (binary or ASCII), the sensor data and the fact of having or not -the sensor- an associated table (See 5.3, “Algorithms to computing readouts. Agent classes”).

Nevertheless, the point is not how this family of classes is created - inheritance-, but rather the technique that we will use to link the one needed to each concrete sensor: is the *sensor* class object what in turn will instantiate the needed class from the family of *calculation agents*: this way we will get the sensor class *readout* method, which is the one in charge of computing the reading, to do it by delegating to the calculation agent object. The code will be exactly the same whatever are the sensor and bus; a mere line:

```
return self.agent.convert(value)
```

(as will be seen in due course).

So we have an initial library pattern:

1. Classes for each one of the ADCbuses that can be connected to the Raspberry: oneWire, ADS1115 and Arduino. As will be seen in due course, an abstract class -ADCBus- will also be created, from which these three inherit; It is -the abstract class- half a matter of style and half of utility: there are a couple of basic (essential) methods that all bus classes share.

2. A class *sensor*, whose main purpose is to act as a data warehouse, will enclose a *readout* method that, by means of a previously -and conveniently- instantiated calculation agent class, computes the final human readable value. At the moment of instantiation the objects of this class perform another key task by bringing together all the objects that will collaborate with each other (See 5.4.- “Class sensor code. Instantiating”).
3. A *calculation agent* family in charge of computing readouts.

Less than a dozen classes: the stuff to solve is not complicated.

3.3.- Library pattern. Classes.

Using an OMT (https://en.wikipedia.org/wiki/Object-modeling_technique) based notation the figure below shows the intended pattern.

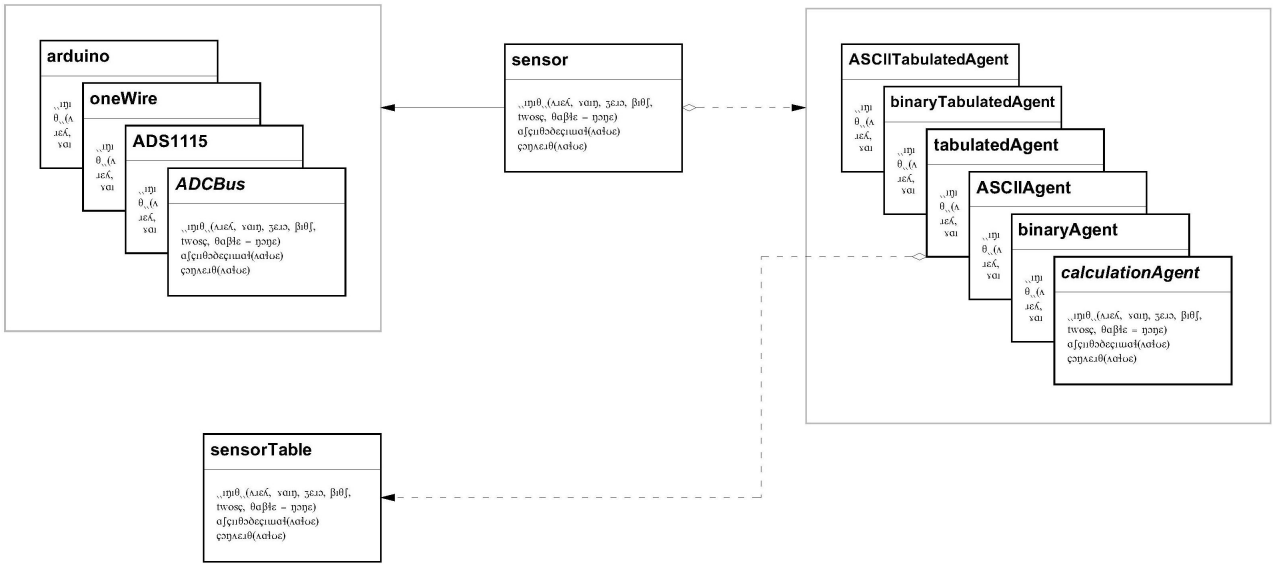


Figure 3-2. Library EIDEAnalog. Preliminary class pattern.

(Notation is derived from the excellent book “Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides”, that in turn is derived from the OMT standard).