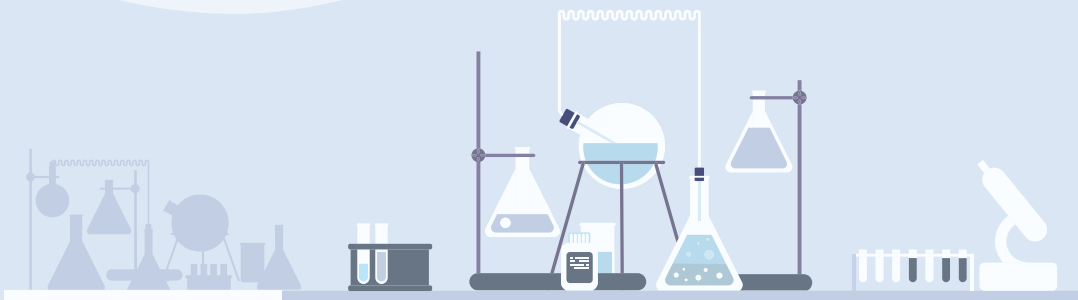
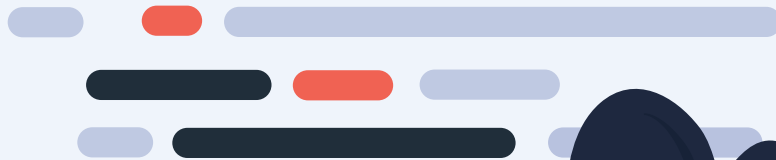


Dr. Aquiles Carattino



Python *for the lab*



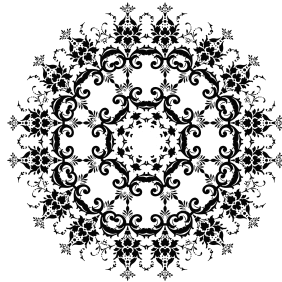
PYTHON FOR THE LAB

**AN INTRODUCTION TO SOLVING
THE MOST COMMON PROBLEMS
A SCIENTIST FACES IN THE LAB**

BY

AQUILES CARATTINO

PhD IN PHYSICS, SOFTWARE DEVELOPER



AMSTERDAM

Compiled on May 6, 2020

Contents

1	Introduction	1
1.1	What are you going to learn	2
1.2	Who Can Read this Book	2
1.3	Why building your software	3
1.4	PFTL DAQ Device	3
1.5	Why Python?	3
1.6	The Onion Principle	4
1.7	Where to get the code	5
1.8	Organizing a Python for the Lab Workshop	5
2	Setting Up The Development Environment	7
2.1	Introduction	7
2.2	Python or Anaconda	7
2.3	Installing Anaconda	8
2.3.1	Using Anaconda	8
2.3.2	Conda Environments	10
2.4	Installing Pure Python	12
2.4.1	Python Installation on Windows	12
2.4.2	Adding Python to the PATH on Windows	13
2.4.3	Installation on Linux	14
2.4.4	Installing Python Packages	14
2.4.5	Virtual Environment	16
2.5	Qt Designer	19
2.5.1	Installing on Windows	19
2.5.2	Installing on Linux	20
2.6	Editors	20
3	Writing the First Driver	23
3.1	Objectives	23
3.2	Introduction	23
3.2.1	Scope of the Chapter	24
3.3	Communicating with the Device	24
3.3.1	Organizing Files and Folders	26
3.4	Basic Python Script	27
3.5	Preparing the Experiment	30
3.6	Going Higher Level	31
3.6.1	Abstracting Repetitive Patterns	35
3.7	Doing something in the <i>Real World</i>	38

3.7.1	Analog to Digital, Digital to Analog	39
3.8	Doing an experiment	40
3.9	Using PyVISA	41
3.10	Introducing Lantz	42
3.11	Conclusions	46
3.12	Addendum 1: Unicode Encoding	46
4	Model-View-Controller for Science	49
4.1	Introduction	49
4.2	The MVCs design pattern	50
4.3	Structure of The Program	51
4.4	Importing modules in Python	52
4.5	The PATH variable	56
4.6	The Final Layout	56
4.7	Conclusions	57
5	Writing a Model for the Device	59
5.1	Introduction	59
5.2	Device Model	59
5.3	Base Model	61
5.4	Adding real units to the code	62
5.5	Testing the DAQ Model	65
5.6	Appending to the PATH at runtime	67
5.7	Real World Example	68
5.8	Conclusions	68
6	Writing The Experiment Model	71
6.1	Introduction	71
6.2	The Skeleton of an Experiment Model	72
6.3	The Configuration File	72
6.3.1	Working with YAML files	72
6.3.2	Loading the Config file	75
6.4	Loading the DAQ	77
6.4.1	The Dummy DAQ	78
6.5	Doing a Scan	79
6.6	Saving Data to a File	82
6.7	Conclusions	85
7	Run an experiment	87
7.1	Introduction	87
7.2	Running an Experiment	87
7.3	Plotting Scan Data	89
7.4	Running the scan in a nonblocking way	89
7.4.1	Threads in Python	90
7.5	Threads for the experiment model	92
7.6	Improving the Experiment Class	95
7.6.1	Threads and Jupyter Notebooks	96
7.7	Conclusions	96

8	Getting Started with Graphical User Interfaces	97
8.1	Introduction	97
8.2	Simple Window and Buttons	98
8.3	Signals and Slots	100
8.3.1	Start a Scan	101
8.4	Extending the Main Window	102
8.5	Adding Layouts	104
8.6	Plotting Data	106
8.6.1	Refresh Rate and Number of Data Points	109
8.7	Conclusions	110
9	User Input and Designing	111
9.1	Introducion	111
9.2	Getting Started with Qt Designer	111
9.2.1	Compiling or not Compiling ui files	115
9.3	Adding User Input	115
9.4	Validating User Input	118
9.4.1	Saving Data with a Shortcut	120
9.5	Conclusions	122
9.6	Where to Next	122
Appendix A	Python For The Lab DAQ Device Manual	127
A.1	Capabilities	127
A.2	Communication with a computer	127
A.3	List of Commands Available	127
Appendix B	Review of Basic Operations with Python	129
B.1	Chapter Objectives	129
B.2	The Interpreter	129
B.3	Lists	129
B.4	Dictionaries	131
Appendix C	Classes in Python	135
C.1	Defining a Class	135
C.2	Initializing classes	137
C.3	Defining class properties	138
C.4	Inheritance	139
C.5	Finer details of classes	140
C.5.1	Printing objects	140
C.5.2	Defining complex properties	141

Chapter 1

Introduction

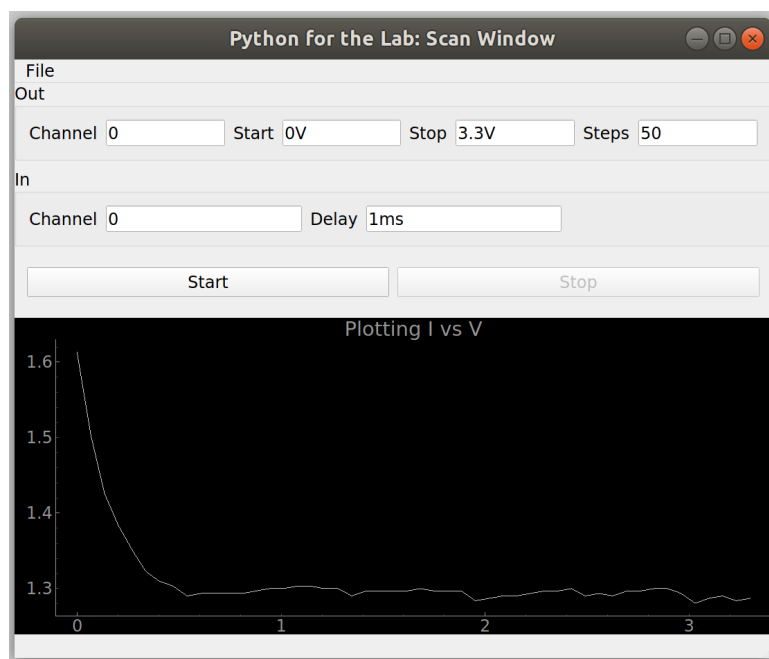
In most laboratories around the world, computers are in charge of controlling experiments. From complex systems such as particle accelerators to simpler UV-Vis spectrometers, there is always a computer responsible for asking the user for some input, performing a measurement, and displaying the results. Learning how to control devices through the computer is, therefore, of the utmost importance for every experimentalist who wants to gain a deeper degree of freedom when planning measurements.

This book is task-oriented, meaning that it focusses on how things can be done and not on much theory on how programming works in general. This approach can lead to some generalizations that may not be correct in all scenarios. I ask your forgiveness in those cases and your cooperation: if you find anything that can be improved or corrected, please contact me.

Together with the book, there is a website¹ where you can find extra information, anecdotes, and examples that didn't fit in here. Remember that the website and its forum are the proper places to communicate with fellow Python For The Lab readers. If you are stuck with the exercises or you have questions that we don't answer in the book, don't hesitate to shout in the forum. Continuous feedback is the best way to improve this book.

¹<https://www.pythonforthelab.com>

1.1 What are you going to learn



This book is the result of many years developing software for scientific applications and, more importantly, of several workshops organized in different universities and companies around Europe. In this time, we have gained experience developing new programs, and we have also collected invaluable feedback from students. With these two elements, we have designed the book in a way that allows the reader to improve their Python proficiency at the same time that we show a clear path to get started with instrumentation software.

Each chapter was carefully crafted to introduce new Python topics next to specific tools needed to control an experiment. For example, in Chapter 3, we develop a driver for a device and take a first dive into classes and objects. We introduce threads in Chapter 7 when we discuss how to be able to stop a running experiment. We also cover how to create user interfaces to accept user input and display data in real-time, such as in the image above. We believe that by following a task-oriented approach, students get the initial direction, tools, and vocabulary to ask for help if needed.

Regarding instrumentation itself, we have compiled what we believe are best practices that speed up the development of solutions and ensure a more prolonged survival of the programs. We discuss how to follow programming patterns that allow the exchange of solutions between people from the same lab or from across the world. This book is not a programmer's book, but a scientist's book written for another scientist. We try to use clear and concise language as much as possible, avoiding jargon when not necessary.

1.2 Who Can Read this Book

To follow the book, we don't assume proficiency in Python. We only require people to have a grasp of *if-statements*, *for-loops*, and *while-loops*, and when to use them. We build the knowledge when it is required, through carefully crafted examples and exercises. If the reader is already proficient in Python, we believe there is value in the best practices we show, in how we structure the code, and how we decided to solve problems. The path is never one, but the goal is the same: extend the possibilities of an experiment by controlling it with custom software.

We believe that anybody working in a lab already has some knowledge of how to perform an experiment. The book proposes to measure the I-V curve of a diode. It is not required to understand the phenomenon, we simply use it as an example of an experiment in which a voltage is varied, and another voltage is measured. This simple example is the building block of most experiments, from controlling temperatures to moving piezo-stages, to tuning the frequency of a laser. By using an LED as the diode in the experiment, we can literally see the effect of applying a voltage.

1.3 Why building your software

Computers and the software within them, should be regarded as tools and not as obstacles in a researcher's daily tasks. However, when it comes to controlling a setup, many scientists prefer to be bound by the specifications of the software provided instead of pursuing innovative ideas. Once a researcher learns how to develop their programs, these limits fall, and creativity can sprout. With automation, the throughput of the setup can increase, human errors can be reduced, or experiments that were no possible become reachable through the introduction of feedback loops.

However, there is an added consideration while building software for research labs: reproducibility. It is a primary concern for modern scientists on how to be able to reproduce results and how to enable others to perform the same measurements. We believe that open-sourcing software as much as possible lowers the entry barrier, and allows present and future colleagues to build on experience instead of reinventing it. The practices we follow in the book are ideal for sharing entire programs or at least parts of them with the community.

1.4 PFTL DAQ Device

We have developed a device nicknamed PFTL DAQ that works as a data acquisition board. The instructor provides these boards during the workshops, but if you got this book online and would like to buy one of PFTL DAQ's, please contact us². The devices are open source/open hardware, they are based on the Arduino DUE, and you can find the instructions for building one on our website. If you have access to any other acquisition card, with a bit of tinkering, you will be able to adapt the course contents to your needs.

Building software for the lab has a reality component not covered in any other books or tutorials. The fact that we are interacting with real-world devices, which can change the state of an experiment, makes the development process much more compelling. The PFTL DAQ is a toy device, easy to replace, but capable of performing quantitative measurements.

1.5 Why Python?

Python became ubiquitous in many research labs because of many different reasons. First, Python is open source, and we firmly believe that the future of research lies in openness. Even for an industrial researcher, the results and the process for generating data should be open to your colleagues (present and future). Python leverages the knowledge gathered in very different areas to deliver a better product. From high-performance computing to machine learning, to experiments, to websites, Python can be found everywhere.

²courses@pythonforthelab.com

Another factor to take into account is that Python is free, and therefore there is no overhead when implementing it. There are no limits to the number of machines in which you can install Python, nor the number of different simultaneous users. Moreover, there is a myriad of professionally developed tools such as `numpy`, `scipy`, `scikit`. Companies such as Anaconda provide customers with high-quality advice and troubleshooting, feeling an often encountered gap with open-source software.

However, for experimentalists, there is a big downside when considering Python. Searching online for instructions on how to control an experiment, few sources appear and even less if focusing on Python alone. Fortunately, this is changing thanks to an evergrowing number of people developing open source code and writing handy documentation. Python can achieve all the same functionality of LabView. The only limitation is the existence of drivers for more sophisticated instruments. With a stronger community, companies will realize the value of providing those drivers for other programming environments.

But the choice of Python is not restricted to the lab. In many cases, Python is used for data analysis, and therefore it makes sense to bring its use to the source of the data: the experiment itself. Moreover, with Python, it is possible to build websites, develop machine learning algorithms, automatize your daily tasks, and many more exciting things. Learning Python increases the employability chances in and out of academia, both for people wishing to continue working with experiments or for people who want to focus on data analysis or beyond.

1.6 The Onion Principle

When we start developing software, it is tough to think ahead. Most likely, we have a small problem that we want to solve as quickly as possible, and we just go for it. Later on, it may turn out that the small problem is something worth investigating deeper. Our software will not be able to handle the new tasks, and we will need to improve it. Having a proper set of rules in place will help us develop code that can adapt to our future needs while keeping us productive in the present. We like to call those rules the Onion Principle.

The rules we are talking about are not rules written in stone. They are not found in books (by the way, they are not here either). We are talking about a state of mind that empowers ourselves to develop better, clearer, and more expandable code. Sitting down and reflecting is the best we can do, even more than sitting down and typing. When dealing with experiments, we have many things to ask ourselves, what do we know, what do we want to prove how to do it. Only then will we sit down to write a program that responds to our needs.

If we build something that we cannot expand, it becomes useless very soon. When we don't know what may happen with our code, we should think ahead and structure it as an onion, in layers. It is not something that happens naturally, but we can develop our set of procedures to ensure that we are developing future-proof code. Once we get the handle on it, it won't take us longer than being disorganized and not having the proper structure. We can avoid variables that are not self-descriptive, lack of comments, and the list goes on and on.

It is not all about being future-proof. When we start with a simple task at hand, we want to solve it quickly and not to spend hours developing useless lines of code just thinking what if. It is also known as premature optimization. If we spend much time trying to solve a problem that may appear, we might just not see the problem that will arise. Therefore, it is better to fail quickly and improve than to fail later and run out of time. However, having a strong foundation is always important. Taking shortcuts just because we don't want to create a separate file will give us more headaches, even in the short term. We should build code that is robust enough to support for

expansion later on. In the same way that we take several steps to perform an experiment, starting with the sample preparation, we should take steps when developing software.

In this book, we go from the one-off script that can get the job done in a matter of minutes, to a fully-fledged user interface that allows us to change the parameters of the experiment and visualize them in real-time.

1.7 Where to get the code

The code that we develop through the book is freely available on Github³. The code is organized by chapters, to make it more accessible while reading. There is also an extra folder with a version of the program that goes beyond what the book covers. For example, the code includes documentation and an installation script. In this way, the readers can have an idea of the possible directions to take for their software.

If you have found any errors or would like to contact us, please send us an e-mail to aquiles@pythonforthelab.com. We will come back to you as soon as possible.

1.8 Organizing a Python for the Lab Workshop

Python for the Lab was born to bring together researchers working in a lab and the Python programming language. With that goal in mind, we developed not only this book but also a workshop in which we can train scientists. The workshops change in duration and content, and we can adapt them to the specific needs of the group.

If you would like to organize a Python for the Lab workshop at your institution, contact us by e-mail to courses@pythonforthelab.com, and we will gladly discuss with you the different options. You can also find more information about the courses on the website: <https://www.pythonforthelab.com>.

³<https://github.com/PFTL/py4lab>

Chapter 2

Setting Up The Development Environment

2.1 Introduction

To start developing software for the lab, we are going to need different programs. The process of installing programs is different depending on the operating system. It is almost impossible to keep an up-to-date detailed instruction set for every possible version of each program and every possible hardware configuration. The steps below are general and should not present issues. When in doubt, it is always best to check the instructions that the developers of the different packages provide, or ask in the forums.

2.2 Python or Anaconda

If you are already familiar with Python, you probably have encountered that different distributions are worth discussing. Python, in itself, is a text document that specifies what to expect when certain commands are encountered, giving much freedom to develop different implementations of those specifications, each one with different advantages. The *official* distribution is available at python.org and it is the distribution maintained by the Python Software Foundation. In the following sections, we discuss step by step how to install it. This distribution is also referred to as CPython because it is written in the programming language C. The official distribution follows the specification of Python to the letter and therefore is the one that comes bundled with Linux and Mac computers. Newer versions of Windows will start shipping with the official Python distribution.

However, the base implementation of Python left some room for improvement in certain areas. Some developers started to release optimized Python distributions tailored for specific tasks. For example, Intel released a specially designed version of Python to support multi-core architectures, and that leverages specific, low-level libraries developed by themselves. There are other versions of Python, such as PyPy, Jython, Iron Python, and others. Each one has its own merits and drawbacks. Some can run much faster in some contexts but at the expense of limiting the number of things that you can do. Between this wealth of options, there is one that is very popular amongst scientists and everyone doing numeric computations called *Anaconda*, and that we cover in this book.

To expand Python, we can use external packages that can be developed and made publicly available by anyone. Some time ago, the python package manager was limited; it allowed us to install only more straightforward packages. There was a clear need to have a tool that allowed to install more complex packages, including libraries not written in Python. Most numerical programs rely on libraries written in lower-level programming languages such as Fortran or C, and those libraries are not always easy to install in all operating systems, nor to keep track of

their dependencies and versions. Anaconda was born to address these issues and is still thriving nowadays. Anaconda is a distribution of Python that comes with *batteries included* for scientists. It includes many Python libraries by default and also supporting programs. It also includes a potent package manager that allows us to install highly optimized libraries for different environments, regardless of whether we are using Windows, Linux, or Mac.

The first edition of this book included instructions for using exclusively plain Python because Anaconda is overkill for the purposes we are covering. However, it is common for researchers to have Anaconda installed on their computers. Therefore, we decided to show how to work with it. If you are starting from scratch, we highly encourage you to start with Anaconda, because it makes your life as a scientist easier. However, if you are using a more limited computer or your installation options are limited, you can use plain Python. For this book, it is simple to install all the libraries required with either system.

2.3 Installing Anaconda

To install Anaconda, you just need to head to the official website: anaconda.com. Go to the download section and select the installer of the newest version of Python. It usually auto-detects your operating system and offers you either a graphical installation (recommended) or a command-line one. If you are on Linux, you have to be careful whether you want the Anaconda Python to become your default Python installation. Typically, there won't be any issues; you just need to be aware of the fact that other programs that rely on Python use the Anaconda version and not the stock version.

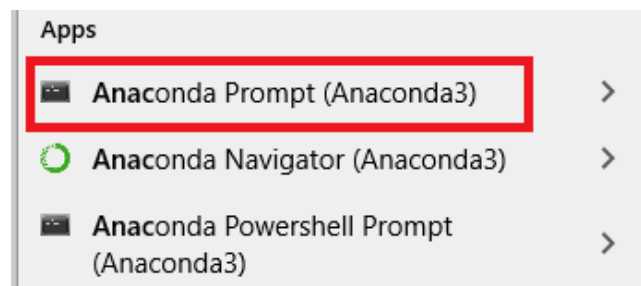


Note

Similar to the different distributions of Python, Anaconda also comes in two primary flavors: Anaconda and Miniconda. The main difference is that the latter bundles fewer programs and therefore is lighter to download. Unless you are very low in space on your computer or you have particular requirements, we strongly recommend downloading Anaconda.

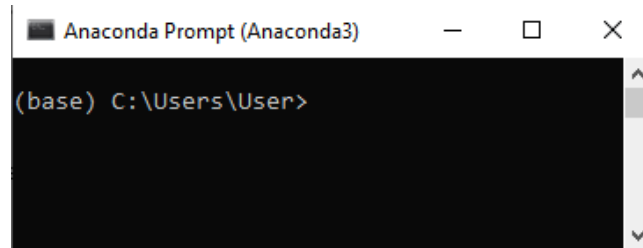
2.3.1 Using Anaconda

Even though Anaconda comes with a graphical interface to installing packages, throughout the book, we favor the command line because it is easier to transmit ideas with words. If you are on Windows, you need to start a program called *Anaconda Prompt*, such as we show in the image below:



If you are on Linux, you only need to open a terminal. On Ubuntu, you can do this by pressing Ctrl+Alt+T. What is important to note is that when you trigger Anaconda, you see that your

command line has a `(base)` prepending it. This is the best indication to know you are running on Anaconda installation, as you can see in the image below:



You can run the following command to see all the installed packages:

```
conda list
```

The output depends on what you have installed, and if you have already used Anaconda in the past. In any case, you see that at the beginning it tells you where the Anaconda installation is, and then you have 4 columns: Name, Version, Build, and Channel, something like this:

```
# packages in environment at /opt/anaconda3:
#
# Name                      Version          Build    Channel
matplotlib                  3.1.3            py37_0
numpy                       1.18.1           py37h4f9e942_0
pyyaml                      5.3              py37h7b6447c_0
yaml                        0.1.7            had09818_2
```

I have just selected some of the packages as an example, but the output should be much longer. One of the good things about Anaconda is that it keeps track of not only the package and its version but also the build. The difference is that you may be using Anaconda on a computer with an Intel processor, or a Raspberry Pi with an ARM processor. In both cases, the version of, let's say, numpy may be the same, but they were compiled differently. Also, you could be using the same version of numpy but with a different version of Python, hence the `py37` that appears in the build numbers, allowing you to keep full track of what you are doing at every moment.

The last two lines show you a package called `pyyaml` that depends on a library called `yaml`, and that we use later. With Anaconda, you can keep track of both separately, the Python package and the lower-level library that this package uses. If you come from Linux, this is not a great surprise, since this is what package managers do. If you come from Windows, however, this is something incredibly handy.

Let's say we would like to install a package that is not yet available. A package that we use later in the book is called `PySerial`. Installing it becomes as easy as running the following command:

```
conda install pyserial
```

It outputs some information, such as the version and the build, and it asks us if we want to install it. We can select 'yes', and it proceeds. If we list the installed packages again, you notice that `PySerial` is on it.

But this is not all Anaconda allows us to. We can also separate environments based on your projects.

2.3.2 Conda Environments

A conda environment is, in practical matters, a folder where all the packages that we need to run code are located, including also the underlying libraries. The environments are isolated from each other; therefore, if you update or delete a package on one, it does not affect the others. When you are working on different projects, perhaps one of them needs a specific version of a library, and you don't want to ruin the other projects. To create a new environment, you need to run the following command (change `myenv` by any name you want):

```
conda create --name myenv
```

And then we activate it:

```
conda activate myenv
```

If now you list the installed packages you will see there is nothing there:

```
conda list
# packages in environment at /opt/anaconda3/envs/myenv:
#
# Name                                Version                                Build Channel
```

Now is time to install the packages we want, starting with Python itself:

```
conda install python=3.7
```

Python Versions

The `3.7` that we added after Python specifies which version of Python we want to use. If you don't specify it, Anaconda installs the newest version, which at the time of writing is `3.8`. When Python updates, some libraries may not work correctly, or may not be available yet for that specific version. When selecting the Python version, be sure all your libraries are available.

After installing Python, you can start it by running:

```
python
```

And the output will be something like this:

```
Python 3.7.7 (default, Mar 26 2020, 15:48:22)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
```

To exit, just type:

```
exit()
```

To follow the book, you will need these packages:

- `numpy` -> For working with numerical arrays

- `pyserial` -> For communicating with serial devices
- `PyYAML` -> To work with YAML files, a specially structured text file
- `PyQt` -> Used for building Graphical User Interfaces
- `pyqtgraph` -> Used for plotting results within the User Interfaces

Which we can install by running:

```
conda install numpy pyserial pyyaml pyqt pyqtgraph
```

Don't worry too much about these packages, since we are going to see one by one later on. If you run `conda list`, you see that you got many more things installed. Each package depends either on other packages or libraries, and Anaconda took care of installing all of them for us. With a `conda install` command, we can install packages that Anaconda itself maintains. Those are official packages that come with a *certification* of quality. Many companies allow their employees to install only packages officially supported by Anaconda to avoid having malware installed within their network.

To follow the book, we need one extra package called `Pint`. This package is not on the official conda repositories. To install packages that didn't make it to the official repository yet, we can use an unofficial repository called `conda forge`. Some packages that are not mature enough, or versions that are too new and not tested enough, are located in this repository. To install a package, we just need to run the following command:

```
conda install -c conda-forge pint
```

The `-c conda-forge` specifies the `channel` from which we are installing the package. With this, we have completed installing all the packages we need to follow the rest of the book.

If you want to go outside of the environment, you can run:

```
conda deactivate
```

Quicker Environment Creation

In the steps above, we have created an empty environment, and then we installed the packages we wanted. We can perform this operation slightly faster if we already know what we need, for example, we can do the following:

```
conda create --name env python=3.7 numpy=1.18 pyserial
```

The command above creates an environment using the specified versions of Python and Numpy while using the latest version of `pyserial`.

Remove an Environment

If you want to remove a conda environment called `env`, you can run the following command:

```
conda remove --name env --all
```

In practice, you also use the `remove` command to uninstall packages. When you do `remove --name env` means you want to remove a specific package from that environment, while the `--all` tells Anaconda to remove all the packages and the environment itself. Use with care, since we can't undo it.

2.4 Installing Pure Python

If instead of installing Anaconda, you prefer to install pure Python, the procedure is straightforward, it just varies slightly on different operating systems.

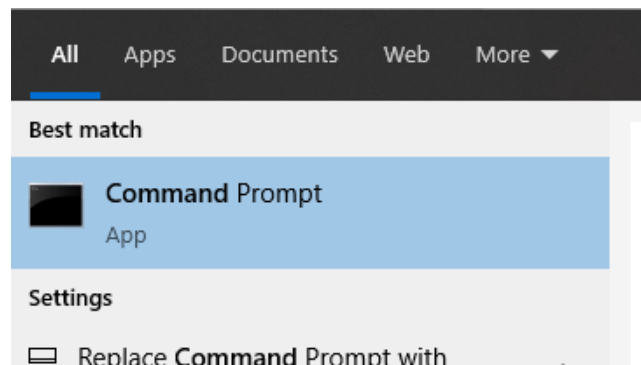
2.4.1 Python Installation on Windows

Windows doesn't come with a pre-installed version of Python. Therefore, you need to install it yourself. Fortunately, it is not a complicated process. Go to the download page at [Python.org](https://python.org), where you find a link to download the latest version of Python.

We have tested all the contents of this book with Python 3.7, but newer versions shouldn't give any problems. If you install a more recent version and find problems later on, come back to this step, uninstall Python and reinstall an older version. Once the download is complete, you should launch it and follow the steps to install Python on your computer. Be sure that **you select Add Python 3.7 to the PATH**. If there are more users on the computer, you can also select *Install Launcher* for all users. Just click on *Install Now*, and you are good to go. Pay attention to the messages that appear, in case anything goes wrong.

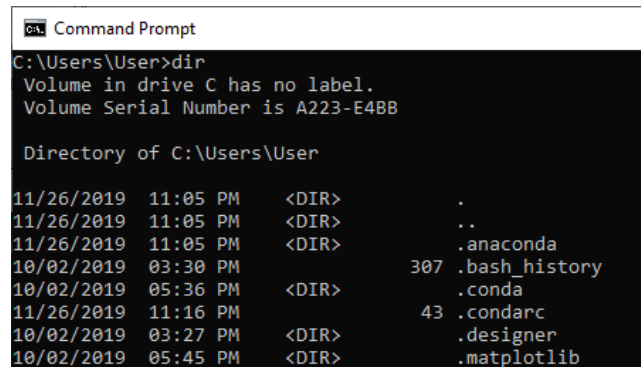
Testing Your Installation

To test whether the installation of Python worked, you need to launch the Command Prompt. The Command Prompt in Windows is the equivalent to a Terminal in the majority of the operating systems based on Unix. Throughout this book, we are going to talk about the Terminal, the Command Prompt, or the Command Line interchangeably. The Command Prompt is a program that allows you to interact with your computer by writing commands instead of using the mouse. To start it, just go to the Start Button and search for the Command Prompt (it may be within the Windows System apps), it looks like the image below shows:



In the Command Prompt, you can do almost everything that you can do with the mouse on your computer. The command prompt starts in a specific folder on your computer, something similar to `C:\Users\User`. You can type `dir`, and press enter to get a list of all the files and folders within that directory. If you want to navigate through your computer, you can use the command `cd`.

If you want to go one level up, you can type `cd ..` if you want to enter into a folder, you type `cd Folder` (where *Folder* is the name of the folder you want to change to). It is out of the scope of this book to cover all the different possibilities that the Command Prompt offers, but you shouldn't have any problems finding help online. See the image below to get an idea of how things look like on Windows:



```

C:\Users\User>dir
Volume in drive C has no label.
Volume Serial Number is A223-E48B

Directory of C:\Users\User

11/26/2019  11:05 PM    <DIR>          .
11/26/2019  11:05 PM    <DIR>          ..
11/26/2019  11:05 PM    <DIR>          .anaconda
10/02/2019  03:30 PM             307 .bash_history
10/02/2019  05:36 PM    <DIR>          .conda
11/26/2019  11:16 PM             43 .condarc
10/02/2019  03:27 PM    <DIR>          .designer
10/02/2019  05:45 PM    <DIR>          .matplotlib

```

To test that your Python installation was successful, just type `python.exe` and hit enter. You should see a message like this:

```

Python 3.7.7 (default, Oct  3 2017, 21:45:48)
[GCC 7.2.0] on Win64
Type "help", "copyright", "credits" or "license" for more information.

```

It shows which Python version you are using and some extra information. You have just started what is called the Python Interpreter, which is an interactive way of using Python. If you come from a Matlab background, you notice its similarities immediately. Go ahead and try it with some mathematical operation like adding or dividing numbers:

```

>>> 2+3
5
>>> 2/3
0.6666666666666666

```

For future reference, when you see lines that start with `>>>` it means that we are working within the Python Interpreter. The lines without `>>>` in front are the output generated by the program.

2.4.2 Adding Python to the PATH on Windows

If you receive an error message saying that the command `python.exe` was not found, it means that something went slightly wrong with the installation. Remember when you selected Add Python to the PATH? That option is what tells the Command Prompt where to find the program `python.exe`. If, for some reason, it didn't work while installing, you have to do it manually. First, you need to find out where your Python is installed. If you paid attention during the installation process, that shouldn't be a problem. Most likely you can find it in a directory like:

```

C:\Users\**YOURUSER**\AppData\Local\Programs\Python\Python36

```

Once you find the file `python.exe`, copy the full path of that directory, i.e. the location of the folder where you found `python.exe`. You have to add it to the system variable called PATH:

1. Open the System Control Panel. How to open it is slightly dependant on your Windows version, but it should be Start/Settings/Control Panel/System
2. Open the Advanced tab.
3. Click the Environment Variables button.
4. There is a section called System Variables, select Path, then click Edit. You'll see a list of folders, each one separated from the next one by a `;`.
5. Add the folder where you found the `python.exe` file at the end of the list (don't forget the `;` to separate it from the previous entry).
6. Click OK.

You have to restart the Command Prompt for it to refresh the settings. Try again to run `python.exe`, and it should be working now.

2.4.3 Installation on Linux

Most Linux distributions come with pre-installed Python. Therefore you have to check whether it is already in your system. Open up a terminal (Ubuntu users can do `Ctrl+Alt+T`). You can then type `python3`, and press enter. If it works you should see something like this appearing on the screen:

```
Python 3.6.3 (default, Oct 3 2017, 21:45:48)
[GCC 7.2.0] on Linux
Type "help", "copyright", "credits" or "license" for more information.
```

If it doesn't work, you need to install Python 3 on your system. Ubuntu users can do it by running:

```
sudo apt install python3
```

Each Linux distribution has a slightly different procedure to install Python, but all of them follow more or less the same ideas. After the installation, check again if it went well by typing `python3` and hitting enter. Future releases of the operating system will include only Python 3 by default, and you won't need to include the `3` explicitly. In case there is an error, try first running only `python` and checking whether it recognized that you want to use Python 3.

2.4.4 Installing Python Packages

One of the characteristics that make Python such a versatile language is the variety of packages that can be used in addition to the standard distribution. Python has a repository of applications called PyPI, with more than 100000 packages available. The easiest way to install and manage packages is through a command called **pip**. Pip fetches the needed packages from the repository and installs them for you. Pip is also capable of removing and upgrading packages. More importantly, Pip also handles dependencies, so you won't have to worry about them.

Pip works both with Python 3 and Python 2. To avoid mistakes, you have to be sure you are using the version of Pip that corresponds to the version of Python you want to use. If you are on Linux and you have both Python 2 and Python 3 installed, probably there are two commands, `pip2` and `3`. You should use the latter to install packages for Python 3. On Windows, you probably have

to use `pip.exe` instead of just `pip`. If, for some reason it doesn't work, you need to follow the same procedure that we explained earlier to add `python.exe` to the `PATH`, but this time with the location of your `pip.exe` file.



Info

Since the moment in which Anaconda was born to nowadays, `pip` has gone through a very long road. Today, we can install complex packages such as `numpy` or `PyQt` directly. However, there is still some discussion regarding how much we can expect from `pip` at the moment of compiling programs or performing complex tasks.

Installing a package becomes very simple. If you would like to install a package such as `numpy`, you should just type:

```
pip install numpy
```

Windows users should instead type:

```
pip.exe install numpy
```



Before Continuing

Before installing the packages listed below, it is important to read the following section on the Virtual Environment. It helps keeping clean and separated environments for software development.

`Pip` automatically grabs the latest version of the package from the repository and installs it on your computer. To follow the book, you need to install the packages listed below:

- `numpy` -> For working with numerical arrays
- `pint` -> Allows the use of units and not just numbers
- `pyserial` -> For communicating with serial devices
- `PyYAML` -> To work with `YAML` files, a specially structured text file
- `PyQt5` -> Used for building Graphical User Interfaces
- `pyqtgraph` -> Used for plotting results within the User Interfaces

You can install all the packages with `pip` without trouble. If you are in doubt, you can search for packages by typing `pip search package_name`. Usually, it is not essential the order in which you install the packages. Notice that since `pip` installs the dependencies as well, sometimes you get a message saying that a package is already installed even if you didn't do it manually.

To build user interfaces, we have decided to use `Qt Designer`, which is an external program provided by the creators of `Qt`. You don't need to have this program to develop a graphical application because you can do everything directly from within `Python`. However, this approach can be much more time consuming than dragging and dropping elements onto a window.

2.4.5 Virtual Environment

When you start developing software, it is of utmost importance to have an isolated programming environment in which you can control precisely the packages installed. You can, for example, use experimental libraries without overwriting software that other programs use on your computer. With isolated environments, you can update a package only within that specific environment, without altering the dependencies in any other development you are doing.

For people working in the lab, it is even more critical to isolate different environments. In essence, you are developing a program with a specific set of libraries, each with its version and installation method. One day you, or another researcher who works with the same setup, decides to try out a program that requires slightly different versions for some of the packages. The outcome can be a disaster: If there is an incompatibility between the new libraries and the software on the computer, you could ruin the program that controls your experiment.

Unintentional upgrades of libraries can set you back several days. Sometimes it was so long since you installed a library that you can no longer remember how to do it or where to get the same version you had. Sometimes you want just to check what would happen if you upgrade a library, or you want to reproduce the set of packages installed by a different user to troubleshoot. There is no way of overestimating the benefits of isolating environments on your computer.

Fortunately, Python provides a great tool called Virtual Environment that gives you a lot of control and flexibility. A Virtual Environment is nothing more than a folder where you find copies of the Python executable and of all the packages that you install. Once you activate the virtual environment, every time you trigger pip for installing a package, it does it within that directory; the python interpreter is going to be the one inside the virtual environment and not any other. It may sound complicated, but in practice, it is incredibly simple.

You can create isolated working environments for developing software for running specific programs or for performing tests. If you need to update or downgrade a library, you are going to do it within that specific Virtual Environment, and you are not going to alter the functioning of anything else on your computer. Acknowledging the advantages of a Virtual Environment comes with time; once you lose days or even weeks reinstalling packages because something went wrong and your experiment doesn't run anymore, you will understand it.



Warning

Virtual Environments are excellent for isolating Python packages, but many packages rely on libraries installed on the operating system itself. If you need a higher degree of isolation and reproducibility, you should check Anaconda.

Virtual Environment on Windows

Windows doesn't have the most user-friendly command line, and some of the tools you can use for Python are slightly trickier to install than on Linux or Mac. The steps below guide you through the installation and configuration. If something is failing, try to find help or examples online. There are a lot of great examples in StackOverflow.

Virtual Environment is a python package, and therefore it can be installed with pip.

```
pip.exe install virtualenv
pip.exe install virtualenvwrapper-win
```

To create a new environment called Testing you have to run:

```
mkvirtualenv Testing --python=path\to\python\python.exe
```

The last piece is crucial because it allows you to select the exact version of Python you want to run. If you have more than one installed, you can select whether you want to use, for example, Python 2 or Python 3 for that specific project. The command also creates a folder called Testing, where it keeps all the packages and needed programs. If everything went well, you should see that your command prompt now displays a (Testing) message before the path. It means that you are indeed working inside the environment.

Once you have finished working in the environment, type:

```
deactivate
```

And you return to the normal command prompt. If you want to work on Testing again, you have to type:

```
workon Testing
```

If you want to test that things are working fine, you can upgrade pip by running:

```
pip install --upgrade pip
```

If there is a new version available, it installs it. One of the most useful commands to run within a virtual environment is:

```
pip freeze
```

It gives you a list of all the packages that you have installed within that working environment and their exact versions. So, you know what you are using, and you can revert if anything goes wrong. Moreover, for people who are worried about the reproducibility of the results, keeping track of specific packages is a great way to be sure that you can repeat everything at a later time.

You can try to install the packages listed before, such as numpy, PyQt5, and see that they get installed only within your Test environment. If you activate/deactivate the virtual environment, the packages you installed within it are not going to be available, and you can see this with `pip freeze`



If you are using Windows Power Shell instead of the Command Prompt, there are some things that you have to change.

```
pip install virtualenvwrapper-powershell
```

And most likely you need to change the execution policy of scripts on Windows. Open a Power Shell with administrative rights (right-click on the Power Shell icon and then select Run as Administrator). Then run the following command:


```
Set-ExecutionPolicy RemoteSigned
```

Follow the instructions that appear on the screen to allow the changes on your computer. It should allow the wrapper to work. You can repeat the same commands that we explained just before and see if you can create a virtual environment.

If it still doesn't work, don't worry too much. Sometimes there is a problem with the wrapper, but you can still create a virtual environment by running:

```
virtualenv.exe Testing --python=path\to\python\python.exe
```

Which creates the virtual environment within the Testing folder. Go to the folder Testing/Scripts and run:

```
.\activate
```

Now you are running within a Virtual Environment in the Power Shell.

Virtual Environment on Linux

On Linux, it is straightforward to install the Virtual Environment package. Depending on where you installed Python, you may need root access to follow the installation. If you are unsure, first try to run the commands without sudo, and if they fail, run them with sudo as shown below:

```
sudo -H pip3 install virtualenv
sudo -H pip3 install virtualenvwrapper
```

If you are on Ubuntu, you can also install the package through apt, although we don't recommend it:

```
sudo apt install python3-virtualenv
```

To create a virtual environment, you need to know where is located the version of Python that you would like to use. The easiest is to note the output of the following command:

```
which python3
```

It tells you what program triggers when you run python3 on a terminal. Replace the location of Python in the following command:

```
mkvirtualenv Testing --python=/location/of/python3
```

It creates a folder, usually `~/.virtualenvs/Testing`, with a copy of the Python interpreter and all the packages that you need, including pip. That folder is the place where new modules are installed. If everything went well, you see the `(Testing)` string at the beginning of the line in the terminal. If you see it, you know that you are working within a Virtual Environment.

To close the Virtual Environment you have to type:

```
deactivate
```

To work in the virtual environment again, just do:


```
workon Testing
```

If for some reason the wrapper is not working, you can create a Virtual Environment by executing:

```
virtualenv Testing --python=/path/to/python3
```

And then you can activate it by executing the following command:

```
source Testing/bin/activate
```

Bear in mind that in this way, you create the Virtual Environment wherever you are on your computer and not in the default folder. It can be handy if you want, for example, to share the virtual environment with somebody, or place it in a precise location on your computer.

Once you have activated the virtual environment, you can go ahead and install the packages listed before, such as `numpy`. You can compare what happens when you are in the working environment or outside, and check that you are isolated from the central installation. The packages that you install inside of Test are not going to be available outside of it.

One of the most useful commands to run within a virtual environment is:

```
pip freeze
```

It gives you a list of all the packages that you have installed within that working environment and their exact versions. In this way, you know what you are using, and you can revert if anything goes wrong. Moreover, for people who are worried about the reproducibility of the results, keeping track of specific packages is a great way to be sure that anyone can repeat it at a later time.

2.5 Qt Designer

Qt Designer is a great tool to quickly build user interfaces by dragging and dropping elements to a canvas. It allows you to quickly develop elaborate windows and dialogs, styling them, and defining some basic features without writing actual code. We use this program to develop an elaborate window in which the user can tune the parameters of the experiment and display data in real-time.

If you are using **Anaconda**, the Designer comes already bundled, so you don't need to follow the steps below.

2.5.1 Installing on Windows

Installing Qt Designer on Windows only takes one Python package: `pyqt5-tools`. Run the following command:

```
pip install pyqt5-tools
```

And the designer should be located in a folder called `pyqt5-tools`. The location of the folder depends on how you installed Python and whether you are using a virtual environment. If you are not sure, use the tool to find folders and files in your computer and search for `designer.exe`.

2.5.2 Installing on Linux

Linux users can install Qt Designer directly from within the terminal by running:

```
sudo apt install qttools5-dev-tools
```

To start the Designer, just look for it within your installed programs, or type `designer` and press enter on a terminal.

The package `pyqt5-tools` is an independent package just aimed at making the installation of the Qt Designer easier. However, it takes a bit of time for it to update to the latest version of Python. At the time of writing, we know that it works with Python 3.7 and not with Python 3.8.

2.6 Editors

To complete the Python For The Lab book, you need a text editor. As with many decisions in this book, you are entirely free to choose whatever you like. However, it is essential to point out some resources that can be useful to you. For editing code, you don't need anything more sophisticated than a plain text editor, such as Notepad++. It is available only for Windows, is very basic and straightforward. You can have several tabs open with different files; you can perform a search for a specific string in your opened documents or within an entire folder. Notepad++ is very good for small changes to the code, perhaps directly in the lab. The equivalent to Notepad++ on Linux is text editors such as Gedit or Kate. Every Linux distribution comes with a pre-installed text editor.

Developing software for the lab requires working with different files at the same time, being able to check that your code is correct before running it, and ideally being able to interface directly with Virtual (or Conda) Environments. For all this, there is a range of programs called IDE's or Integrated Development Environments. We strongly suggest you check **Pycharm**, which offers a free and open-source Community Edition and a Professional Edition, which you can get for free if you are a student or teacher affiliated with a University. Pycharm integrates itself with environments, allows you to install a package if it is missing, but you need it and many more things. It is a sophisticated program, but there are many great tutorials on how to get started. Familiarizing yourself with PyCharm pays off quickly.

Another very powerful IDE for Python is **Microsoft's Visual Studio**, which is very similar to Pycharm in capacities. If you have previous experience with Visual Studio, I strongly suggest you keep using it. It integrates very nicely with your workflow. Visual Studio is available not only for Windows but also for Linux and Mac. It has some excellent features for inspecting elements and help you debug your code. The community edition is free of charge. Support for Python is complete, and Microsoft has released several video-tutorials showing you how to get the best out of their program.

There are other options around, such as Atom or Sublime. However, they don't specifically target Python as the previous two. Remember that always, the choice is yours. Editors should be a tool and not an obstacle. If you have never used an IDE before, I suggest you just install PyCharm. That is what we use during the workshops, and everyone has always been very pleased with it. If you already have an IDE or a workflow with which you are happy, then keep it. If at some point, it starts failing you, you can reevaluate the situation.

! Tabs or Spaces

Python is sensitive to the use of tabs and spaces. You shouldn't mix them. A standard is to use 4 spaces to indent your code. If you decide to go for a text editor, be sure to configure it such that it respects Python's stylistic choices. Notably, Notepad++ comes configured by default to use tabs instead of spaces, which is a problem if you ever copy-paste code from other sources.

Chapter 3

Writing the First Driver

3.1 Objectives

Communicating with real-world devices is the cornerstone of every experiment. However, devices are very different from each other. Not only is their behavior different (you can't compare a camera to an oscilloscope), but they also communicate in different ways with the computer. In this chapter, we are going to build the first driver for communicating with a real-world device. You are going to learn about low-level communication with a serial device and, from that experience, build a reusable class that you can share with other developers.

3.2 Introduction

We can split devices into different categories depending on how they communicate with a computer. One of the most common ways to communicate is through the exchange of text messages. The idea is that the user sends a specific command, i.e., a message, and the device answers with specific information, another message. Sometimes there is no answer because it is just a command to perform an action such as an auto setting or switching off. Sometimes the message we get back contains the information we requested.

To have an idea of how commands look like, you can check the manuals of devices such as oscilloscopes or function generators. Both Tektronics¹ and Agilent have complete sets of instructions. If you search through their websites, you find plenty of examples. A command that you can send to a device may look like this:

```
*IDN?
```

Which is asking the device to identify itself. An answer to that request would look like `Oscilloscope ID#####`. In this chapter, we are going to see how you can exchange messages with devices using Python.

The devices that exchange information with the computer in this way are called **message-based** devices. Some of this type of device are oscilloscopes, lasers, function generators, lock-ins, and many more. The PFTL DAQ device to work with this book also enters into this category. If you got the book online and not as part of a workshop, you can build your device or contact us, and we may be able to offer you one already programmed².

¹You can check the manual of an oscilloscope here: <https://www.tek.com/oscilloscope/tds1000-manual>

²courses@pythonforthelab.com



Device Drivers

There is an entire world of devices that do not communicate through messages, but that specify their own drivers. These devices are typically cameras, fast data acquisition cards, motorized mirrors and stages, and more. They depend on specific drivers and are harder to work with at this stage. If you are already confident programming message-based devices and need to move to non-message based ones, you can check the Advanced Python for the Lab materials.

Remember, *message-based* refers only to how the device exchanges information with the computer, and not to the actual connection between them. It is possible to connect a message-based device via RS-232, USB, GPIB, or TCP/IP. Be aware, however, that it is not a reciprocal relation: not all devices connected through RS-232 or USB are message-based. If you want to be sure, check the manual of the device and see how it is controlled. In this chapter, we are going to build a driver for a message based device.

3.2.1 Scope of the Chapter

In the introduction, we discussed that the objective is to acquire the I-V curve of a diode. You need, therefore, to set an analog output (the V) and read an analog input (the I) with the device. In this chapter, we focus on everything we need to perform our first measurement. However, keep in mind the onion principle, which tells you that you should always be prepared to expand your code later on if the need arises.

3.3 Communicating with the Device

To communicate with the PFTL DAQ³ device, we are going to use a package called `PySerial`, which you should have already installed if you followed chapter 2. The first thing we can do is to list all the devices connected to the computer to identify the one in which we are interested. Plug your device via the USB port of your computer. We need to connect the PFTL DAQ device through the micro USB port closest to the power jack, also known as *programming port*. Then, the following command in a terminal (be sure you are within the environment in which you have installed the required packages):

```
python -m serial.tools.list_ports
```



Warning

From now on, instead of telling you to open a terminal to run a command, if you see code which starts with a `$` symbol, it means that you should run it in the terminal

Depending on the operating system, the output can be slightly different. On Windows, we get something like this:

³PFTL is the shorthand notation for Python For the Lab

```
COM3
```

While if you are on Linux you will see something like this:

```
/dev/ttyACM0
```

The most important thing is to remember the number at the end. If you happen to see more than one device listed (this is very common on Mac), unplug the PFTL DAQ, run the command to list the ports again, note which ones appear. Plug it back and list the devices. The new one is the device in which we are interested.

Now is time to start working with the device. Start Python by running:

```
python
```

And then we can start working directly from the command line. First, we are going to import the package we need for communication:

```
>>> import serial
```



Interpreter Characters

Earlier we explained that we must run in a terminal everything prepended with a \$. Lines prepended by `>>>` are lines that run in a Python interpreter. Note that there is no need to type the `>>>`

And then we can open the communication with the device. Bear in mind that you must change the port number by the one you got earlier:

```
>>> device = serial.Serial('/dev/ttyACM0') # <---- CHANGE THE PORT!
```

Now we are ready to get started exchanging messages with the device. Before we discuss each line, let's see what you can do. The lines without `>>>` are the output generated by the code.

```
>>> device.write(b'IDN\n')
4
>>> answer = device.readline()
>>> print(f'The answer is: {answer}')
The answer is: b'PFTL DAQ Device built by Python for the Lab v.1.2020\n'
>>> device.close()
```

Even if short, many things are going on in the code above. First, we import the `PySerial` package, noting that we are actually importing `serial` and not `PySerial`. Then we open the specific serial port that identifies the device. Bear in mind that serial devices can maintain only one connection at a time. If you try to run the line twice, it gives you an error letting you know that the device is busy. It happens if, for example, we try to run two programs at the same time, or if we start Python from two different terminals.

Once we established the connection, we send the **IDN** command to the device. There are some caveats in the process. First, the `\n` at the end, is a special character known as `newline`. It is a

way to tell the device that we are not going to send more information afterward. When a device is receiving a command, it reads the input until it knows that no more data is arriving. If we were sending a value to a device such as a wavelength, the command could look like `SET:WL:1200`. However, the device needs to know when it has received the last number. It is not the same setting the laser wavelength to 120 nm or to 1200 nm.

The other particular detail is the `b` before the command string. Adding the `b` in front of a string is one way of telling Python to encode a string as a binary string. Devices don't understand what an `A` is. The serial communication can only send a stream of 1's and 0's. Therefore, we need to transform any information we are trying to send, such as `'IDN'`, to bytes before we can send it to the device. We give a lengthier discussion about encoding strings and what it means at the end of the chapter, in section 3.12.

After we write to the device, we get a `4` as output. It is the number of bytes we sent, taking into account that `\n` is only one byte because it is only one character. To get the answer that the device is generating for us, we have to read from it. We use the method `readline()` for this. Then we print the answer to the screen. The answer we get also has a `\n` and a `b`. Finally, we close the connection to the device.

We decided to use the `IDN` command because we knew it existed. But if we are starting with a new device, it is always fundamental to start by reading the manual. Manuals are our best and, perhaps, the only friend we have when developing software for controlling instruments. The PFTL DAQ is no exception. The manual is part of the book, and we can find it in chapter A. It is a simple manual, but with enough information to get started, and it follows similar conventions to those we can find on more complex devices.

In the manual, the first thing we have to find is the line termination. We used the newline character because we knew it, but each device can specify something different. Some devices use the newline character as part of the commands you can send and specify that the line ending must be something else. Once we know how to terminate commands, we can go ahead and see the list of options.

Many devices (but not all) follow a standard called SCPI⁴. The standard makes devices easy to exchange, because the commands for all oscilloscopes are the same, for all function generators are the same, and so forth. Moreover, the SCPI standard follows a structure that makes messages easy to understand and modular. A more powerful oscilloscope, for example, has commands not available to a more basic device, but the common features are controlled by the same messages.

Now that we know how to get started with serial communication, it is time to move to more complex programs. Typing everything on the Python interpreter is not handy and takes much time. So now we can start working with files.

3.3.1 Organizing Files and Folders

When we start a new project, it is always a good idea to decide how we are going to organize the work. In the previous chapter, we have set up an environment for developing a program. That is the first step to be organized. The second step is deciding where we are going to save the files we need to write our program. The general advice is to have a folder for all the programs, for example, `Programs`. Inside, each project we work on has its folder, such as `PythonForTheLab`. Each person has their way of organizing themselves, but from now on, every time we talk about creating a file, we are referring to that base folder for the project.

⁴https://en.wikipedia.org/wiki/Standard_Commands_for_Programmable_Instruments

3.4 Basic Python Script

The code we have developed above can also be written as a Python script, that we can run from the command line without the need to re-write everything. Create an empty file called **communicate_with_device.py**, and add the same code we had earlier to it:

```
import serial

device = serial.Serial('/dev/ttyACM0')
device.write(b'IDN\n')

answer = device.readline()
print(f'The answer is: {answer}')

device.close()
```

Now we can run the file:

```
python communicate_with_device.py
```



To be able to run the file, you need to be in the same folder where the file is. To change the folder in the terminal, you can use `cd`

What happens when you run the file?

The program hangs, there is no error message and no IDN information printed to the screen. It means the program is waiting for something to let it continue. To force the stop of the program, you can press Ctrl+C, and if this does not work on Windows, you can press Ctrl+Pause/Break. Now can see the easiest way to debug when such a situation appears.

We want to know first when the program hangs, and then we can see how to fix it. So we can edit the file and add some print statements to check until which point is running:

```
import serial
print('Imported Serial')

device = serial.Serial('/dev/ttyACM0')
print('Opened Serial')

device.write(b'IDN\n')
print('Wrote command IDN')

answer = device.readline()
print(f'The answer is: {answer}')

device.close()
print('Device closed')
```

Rerun the script. **Where is it hanging?** Surprisingly, it is hanging during the `readline` execution. Can you understand what is going on?

There is something very different between writing on the Python interpreter and running a script: the time it takes to go from one line to the other. While you type, everything happens

slowly, while when you run a script, everything happens incredibly fast. Now, the `readline` is waiting to get some information from the device, but the devices are not generating it. It means that the problem should be earlier when we sent the `IDN` message. The command is not wrong in itself, but what is happening is that between opening the communication with the device and sending the first message, we give no space.

When you establish communication with most devices, there is a small delay until you can start using it. In our case, we must add a delay between starting the communication and sending the first message. We can achieve this by doing the following:

```
import serial
from time import sleep

device = serial.Serial('/dev/ttyACM0')
sleep(1)

device.write(b'IDN\n')
answer = device.readline()
print(f'The answer is: {answer}')
device.close()
print('Device closed')
```

If you rerun the program, you can see that it takes a bit of time to run, but it outputs the proper message. The `sleep` function makes the program wait for a given number of seconds (also fractions) before continuing. You can try lowering the number until you get the minimum possible value. Still, in typical cases, you start the communication only once, therefore waiting 1 second or .5 seconds won't have a significant impact on the overall execution time.

Reading an Analog Value

Before we continue, it would be great to also read a value from the device, not just the serial number. If we refer again to the manual, we see that the way of getting an analog value is using the `IN` command. We can modify the code on our previous program to read a value with the device:

```
import serial
from time import sleep

device = serial.Serial('/dev/ttyACM0')
sleep(1)

device.write(b'IDN\n')
answer = device.readline()
print(f'The answer is: {answer}')

device.write(b'IN:CH0\n')
value = device.readline()
print(f'The value is: {value}')
device.close()
print('Device closed')
```

The value you are reading doesn't make much sense, especially if there is nothing connected to the input number 0; it is just noise. But it is an excellent first step. We can acquire a value from the real world using the device. We take care of all the things that we need to address in the following chapters.

? Exercise

What happens if you use `read()` instead of `readline()` ?

? Exercise

What happens if you use `read()` once, and then `readline()` ?

? Exercise

What happens if you call `readline()` before writing the IDN command?

? Exercise

What happens if you try to write to the device after you have closed it?

! Important note about ports

If you are using the old RS-232 (also simply known as *serial*), the number refers to the physical number of the connection, on Windows, it is something like COM1, on Linux and Mac, it is something like `/dev/ttyACM1`. In modern computers, there are no RS-232 connections, and most likely, we have to use a USB hub for them. It means that there is no physical connection straight from the device into the motherboard. The numbering can change if we plug/unplug the cables. The PFTL DAQ device, since it acts as a hub for a serial connection, can show the same behavior. If you plug/unplug the device while it is being used, the port we get the second time is likely different. The second time we run the program, we need to update the information.

? Exercise

Read the manual of the PFTL DAQ and find a way to set an analog output to 1 Volt.

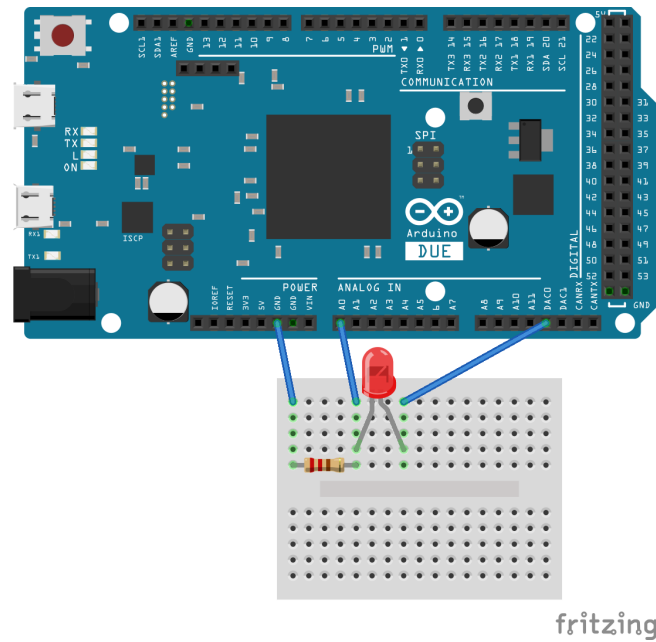


Figure 3.1: Schematic of the connections to perform the experiment

3.5 Preparing the Experiment

Before moving forward with programming, it is time to set up the measurement we want to perform and discuss what we need to achieve it. This book revolves around the idea of measuring the I-V curve of a diode. If you are not too familiar with electronics, don't worry, it is not essential to follow the book, you can just copy the connections as shown below. If you are a bit more familiar with electronics, it is worth explaining what we are going to do.

Diodes are elements that let current flow only in one direction, but their behavior is highly non-linear. The current flowing is not proportional to the voltage applied. We chose to use an LED for the experiments because it is easy to have visual feedback on what is going on. On the other hand, we can't measure current directly. First, we need to transform it into a voltage. If you are familiar with Ohm's law, you remember the relationship:

$$V = I \cdot R \quad (3.1)$$

Voltage is current times resistance. Therefore, if we want to transform a current to a voltage, we just need to add a resistance to the circuit.

To perform the experiment, we need to apply a given voltage and read a voltage. This pattern is common to a wealth of experiments. The underlying meaning is what matters.

With the PFTL DAQ device, the connections that will allow us to apply a voltage and read a voltage are as follows:

Only three cables, an LED and a resistance, are all you need to follow the rest of the book. We apply the voltage to the LED through `DAC0`. The current flows through the diode and the resistance. The voltage that we acquire at the Analog In `A0` is proportional to the current flowing through the resistance.



Exercise

Now that you have set up the experiment know how to set and read values. Acquire the I-V curve of the diode. It is a challenging exercise, aimed at showing you that it doesn't take a long time to be able to achieve an essential goal.

3.6 Going Higher Level

We saw that communicating with a device implies taking into account parameters such as the line ending, or adding the `b` in front of messages for encoding. If the number of commands is large, it becomes very unhandy. The PFTL DAQ device is an exception because it is minimal, but there are still a lot of possible improvements.

If you still remember the *Onion Principle* (Section 1.6), it is now the time to start applying it. If you completed the last exercise, you probably have written much code to do the measurement. Perhaps you used a for loop, and acquired values in a sequence. However, if you want to change any of the parameters, you need to alter the code itself. This approach is not very sustainable for the future, especially if you are going to share the code with someone else.

Since we know how to communicate with the device, we can transform that knowledge into a reusable Python code by defining a class. Classes have the advantage of being easy to import into other projects, which are easy to document and to understand. Moreover, they are easy to expand later on. If you are not familiar with what classes are, check the appendix C for a quick overview. With a bit of patience and critical thinking, you can follow the rest of the chapter and understand what is going on as you keep reading.

Create a new, empty file, called **pftl_daq.py**, and write the following into it:

```
import serial
from time import sleep

class Device:
    def __init__(self, port):
        self.rsc = serial.Serial(port)
        sleep(1)

    def idn(self):
        self.rsc.write(b'IDN\n')
        return self.rsc.readline()
```

The code above shows you how to start a class for communicating with a device and get its serial number. However, if you run the file, nothing happens. At the end of the file, add the following code:

```
dev = Device('/dev/ttyACM0') #<---- Remember to change the port
serial_number = dev.idn()
print(f'The device serial number is: {serial_number}')
```

If we rerun the code, we get the serial number of the device. Let's go line by line to understand what is going on. First, we create a class, and we define what do we want to happen when we do `Device()`. In the `__init__` method, we specify that the class needs a port, and we use that port to start the serial communication. The serial communication is stored as `self.rsc` in the class itself, where `rsc` is just a shorthand notation for *resource*. Then we sleep for one second to give time to the communication to be established.

The second method, `idn`, just repeats what we have done earlier: we write a command, we read the line and return the output. If we look at the few lines at the bottom of the file, we now see that way of working with this class is simpler. We just use `idn()` instead of having to write and read every time.

? Exercise

Once you read the serial number from the device, it does not change. Instead of just returning the value to the user, store it in the class in an attribute `self.serial_number`.

? Exercise

When you use the method `idn`, instead of writing to the device, check if the command was already used and return the value stored. This behavior is called caching, and is very useful not to overflow your devices with useless requests for data.

Reading and Setting Values

We have just developed the most basic class, one that allows us to start the communication with the device and read its identification number. We can also write methods for reading an analog input or generating an output. The most important thing is to decide what argument each method needs. For example, reading a value only needs the channel that we want to read. Setting a value needs not only a channel but also the value itself. Also, reading a means that the method returns something. When you set an output, there is not much to return to the user.

? Exercise

Write a method `get_analog_input` which takes two arguments: `self` and `channel` and which returns the value read from the specified channel.

? Exercise

Write a method `set_analog_value` which takes three arguments: `self`, `channel` and `value` and that sets the output value to the specified port.

Even though the exercises are important for you to start thinking by yourself, they have some caveats that are very hard to iron out if you don't have a bit of experience. First, let's look at the way of reading an analog input. We can try to develop a method that looks like this:

```
def get_analog_input(self, channel):
    message = f'IN:CH{channel}\n'
    self.rsc.write(message)
    return self.rsc.readline()
```

But it won't work, because even though we are adding the line ending, we are missing the `b` that we were using in the other examples. On the other hand, if we try to do something like:

```
message = b'IN:CH{}\n'.format(channel)
```

It will fail, because `format` only works with strings, and as soon as the `b` is added in front of a string, it is encoded to bytes. This means that we have to do it in two steps:

```
def get_analog_input(self, channel):
    message = f'IN:CH{channel}\n'
    message = message.encode('ascii')
    self.rsc.write(message)
    return self.rsc.readline()
```

First we form the message we want to send to the device, then we *encode* it, which is the same as adding the `b` in front of a string. And then we write it to the device. After writing, we return the line with the value. We could use it as follows:

```
dev = Device('/dev/ttyACM0') #<---- Remember to change the port
serial_number = dev.idn()
print(f'The device serial number is: {serial_number}')
volts = dev.get_analog_input(0)
print(volts)
```

If you run the code above, you will notice that the output still has the `b` and the `\n`, we will work on this later. The next step is to generate an output:

```
def set_analog_output(self, channel, output_value):
    message = f'OUT:CH{channel}:{output_value}\n'
    message = message.encode('ascii')
    self.rsc.write(message)
```

And we can use it as follows:

```
dev = Device('/dev/ttyACM0') #<---- Remember to change the port
serial_number = dev.idn()
print(f'The device serial number is: {serial_number}')
volts = dev.get_analog_input(0)
print(volts)
dev.set_analog_output(0, 1000)
```

This would be all, unless we do add something extra, like reading the input after setting the output:

```
dev = Device('/dev/ttyACM0') #<---- Remember to change the port
serial_number = dev.idn()
print(f'The device serial number is: {serial_number}')
volts = dev.get_analog_input(0)
print(volts)
```

```
dev.set_analog_output(0, 1000)
volts = dev.get_analog_input(0)
print(volts)
```

The second time we read the analog input, we get the same value we passed to the analog output. It does not matter if it is 1000 or 999; it does not matter if the cables are connected or not. The value is always the same.

? Exercise

Explain why when getting the analog input we get the same value that we set earlier

This question is very tricky and requires that we read the manual of the device. In the documentation for the `OUT` command, you can see that it returns something: the same value that was passed to it. However, in our method, we are just writing to the device and not reading from it. The message waits in the queue until next time we read from it, and this happens when we try to read an analog input.

As you can see, the number of possible mistakes that we can do when developing this kind of programs is huge. On top of that, many mistakes do not generate an error, and can easily go unnoticed. When performing measurements, perhaps you don't realize the mistake on the `set_analog_output` until you are analyzing the data you acquired.

To solve the problem while setting the output, we just need to read from the device after setting the output:

```
def set_analog_output(self, channel, output_value):
    message = f'OUT:CH{channel}:{output_value}\n'
    message = message.encode('ascii')
    self.rsc.write(message)
    self.rsc.readline()
```

We are not doing anything with the information we get. We just clear it from the device.

Proper Values Instead of Bytes

To have a bit more functional class, it would be great if we could get rid of the extra `b` and `\n` that we get every time we use the `readline()` function. First, we need to transform bytes to strings. In the previous section we transformed strings to bytes by using `.encode('ascii')`, and to no surprise, if we want to transform bytes to a string, we can do the opposite, in the `idn` method, for example:

```
def idn(self):
    self.rsc.write(b'IDN\n')
    answer = self.rsc.readline()
    answer = answer.decode('ascii')
    return answer
```

If you try this out, you will see that it took care of the initial `b`, but the `\n` is still there. We need one more step to get rid of it:


```
def idn(self):
    [...]
    answer = answer.strip()
    return answer
```

Note that we have used `[...]` to hide the code that didn't change. Now you can go ahead and see that the output is formatted correctly.

? Exercise

By using what you've learned for the `idn` method, improve the `get_analog_input` method so that it returns an integer. **Hint:** To transform a string to an integer, you can use `int()`, for example: `int('12')`.

3.6.1 Abstracting Repetitive Patterns

When you start to develop programs, there is a principle called **DRY**, which stands for *don't repeat yourself*. Sometimes it is clear that code is repeating itself, for example, if we copy-pasted some lines. Sometimes, however, the repetition is not about code itself but a pattern. DRY is not a matter of just typing fewer lines of code. It is a way of reducing errors and making the code more maintainable. Imagine that after an upgrade, the device requires a different line ending. We would need to go through all your code to find out where the line ending is used and change it. If we would specify the line ending in only one location, changing it would require just to change one line.

First, we can specify the default parameters for our device. They will be all the constants that we need in order to communicate with it, such as line endings. We can define them just before the `__init__` method, like this:

```
class Device:
    DEFAULTS = {'write_termination': '\n',
                'read_termination': '\n',
                'encoding': 'ascii',
                'baudrate': 9600,
                'read_timeout': 1,
                'write_timeout': 1,
                }
    def __init__(self, port):
        [...]
```

You can see that there is much new information in the class. We have established a clear place where both the read and write line endings are specified (in principle they don't need to be the same), we also specify that we want to use `ascii` to encode the strings and that the baud rate is 9600. This value is the default of `PySerial`, but it is worth making it explicit in case newer devices need a different option. We also specify timeouts, which are allowed by `PySerial` and would prevent the program from freezing if writing or reading takes too long.

It is normally good practice to separate the instantiation of the class with the initialization of the communication. One thing is creating an object in Python, and the other is to establish communication with a real device. Therefore, we can rewrite the class like this:

```
def __init__(self, port):
    self.port = port
    self.rsc = None

def initialize(self):
    self.rsc = serial.Serial(port=self.port,
                             baudrate=self.DEFAULTS['baudrate'],
                             timeout=self.DEFAULTS['read_timeout'],
                             write_timeout=self.DEFAULTS['write_timeout'])

    sleep(1)
```

You can see that there are some major changes to the code, but the arguments of the `__init__` method are the same. In this way, code already written does not fail if we change the number of arguments of a method. When we do this kind of change, it is called *refactoring*. It is a complex topic, but one of the best strategies you can adopt is not to change the number of arguments functions take, and the output should remain the same. In the class, the `__init__` definition looks the same, but its behavior is different. Now, it just stores the `port` as the attribute `self.port`. Therefore, to start the communication with the device, we need to do `dev.initialize()`. You can also see that we have used almost all the settings from the `DEFAULTS` dictionary to start the serial communication.

After we do these changes, we should also update the code we use to test the device:

```
dev = Device('/dev/ttyACM0') #<---- Remember to change the port
dev.initialize()
serial_number = dev.idn()
print(f'The device serial number is: {serial_number}')
volts = dev.get_analog_input(0)
print(volts)
dev.set_analog_output(0, 1000)
volts = dev.get_analog_input(0)
print(volts)
```

So far the only difference with the previous code is the `__init__` method. We have to improve the rest of the class. We already know that for message-based devices there are two operations: **read** and **write**. However, we will only read after a write (remember, we should ask something from the device first.) It is possible to update the methods of the class to reflect this behavior. Since all the commands of the device return a value, we can develop a method called `query`:

```
def query(self, message):
    message = message + self.DEFAULTS['write_termination']
    message = message.encode(self.DEFAULTS['encoding'])
    self.rsc.write(message)
    ans = self.rsc.readline()
    ans = ans.decode(self.DEFAULTS['encoding']).strip()
    return ans
```

In this way, we take the message, append the proper termination, and encodes it as specified in the `DEFAULTS`. Then, it writes the message to the device exactly as we did before. Then we read the line, we decode it using the defaults and strip the line ending. Now it is time to update the other methods of the class to use the `query` method we have just developed. Let's start with `idn`, which now looks like this:

```
def idn(self):
    return self.query('IDN')
```

And the same we can do for the other methods:

```
def get_analog_input(self, channel):
    message = 'IN:CH{}'.format(channel)
    ans = self.query(message)
    ans = int(ans)
    return ans

def set_analog_output(self, channel, output_value):
    message = 'OUT:CH{}:{}'.format(channel, output_value)
    self.query(message)
```

For such a simple device, perhaps the advantages of abstracting patterns are not evident. It is something that happens very often in more extensive programs, and being able to identify those patterns can make the difference between a successful program and something only one person can understand. Note that even if we have changed the methods for identifying, reading, and setting analog values, there is no need to update the example code.

It is important to see that we achieved the communication with the device through the resource `self.rsc` that is created with the method `initialize`. There is a common pitfall with this command. If we try to interact with the device before we initialize it, we get an error like the following:

```
AttributeError: 'NoneType' object has no attribute 'write'
```

We now remember why this happened, but it is very likely that in the future, either we forget or someone else is using our code, and the error message that appears is incredibly cryptical. Therefore, we suggest you do the following:

Exercise

Improve the `query` method to check whether the communication with the device has initialized. If it hasn't, you can print a message to the screen and prevent the rest of the program from running.

When we develop code, we must always keep an eye on two people: the future us and other users. It may seem obvious now that we initialize the communication before attempting anything with the device, but in a month, or a year, when we dig up the code and try to do something new, we are going to be another person. We won't have the same ideas in our mind as right now. Adding safeguards are, on the one hand, a great way of preserving the integrity of your equipment; on the other, it cuts down the time it takes to find out what the error was.

There is only one last thing that we are missing. We have completely forgotten to add a proper way of closing the communication with the device. We can call that method `finalize`:

```
def finalize(self):
    if self.rsc is not None:
        self.rsc.close()
```

We first check that we have actually created the communication by verifying that the `rsc` is not `None`. Then, we can update our example code at the bottom of the file to actually use the `finalize` method:

```
dev = Device('/dev/ttyACM0') #<---- Remember to change the port
dev.initialize()
serial_number = dev.idn()
print(f'The device serial number is: {serial_number}')
volts = dev.get_analog_input(0)
print(volts)
dev.set_analog_output(0, 1000)
volts = dev.get_analog_input(0)
print(volts)
dev.finalize()
```

We may wonder why things work out fine even though we didn't have the `finalize` method in place. The answer is that PySerial is smart enough to close the communication with the device when it realizes we will no longer use it. However, it is not always the case if the program crashes. Sometimes the communication stays open, and the only way to regain control of the device is by manually shutting it off and on again. If this happens, we must always check whether the port changed.

3.7 Doing something in the Real World

Until now, everything looked like a big exercise of programming but now it is time to start interacting with the real world. As we know from reading the manual, the PFTL DAQ device can generate analog outputs, and the values we can use go from 0 to 4095. We can expand slightly the code below the class in order to make the LED blink for a given number of times, and report the measured voltage when it is either on or off:

```
dev = Device('/dev/ttyACM0') #<---- Remember to change the port
dev.initialize()
serial_number = dev.idn()
print(f'The device serial number is: {serial_number}')
for i in range(10):
    dev.set_analog_output(0, 4000)
    volts = dev.get_analog_input(0)
    print(f'Measured {volts}')
    sleep(.5)
    dev.set_analog_output(0, 0)
    volts = dev.get_analog_input(0)
    print(f'Measured {volts}')
    sleep(.5)
```

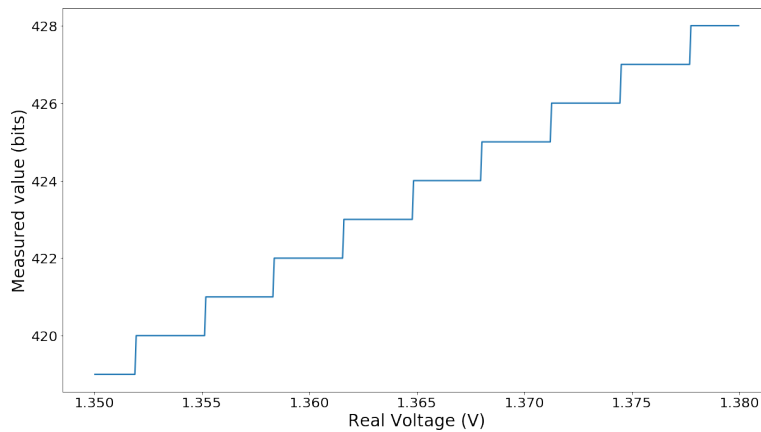
With this simple code, we can switch on and off the LED 10 times, and we print to screen the values that we are reading when it is on or off. There are two things to note: first, we are switching it ON by using a value of 4000. We have selected it because it is high enough to switch the LED on, but it has no units, it is not a voltage. The same with the value reported by the `get_analog_input`, which is just an integer, but we have no idea, yet, of what it means.

Before we can proceed, we must understand how to transform Analog signals to digital values and the opposite.

3.7.1 Analog to Digital, Digital to Analog

Almost every device that we find in the lab transforms a continuous signal to a value that can be understood by the computer. The first step is to transform the quantity you are interested in a voltage. Then, we need to transform the voltage (an analog signal) to something with which the computer can work. Going from the real world to the computer space is normally called *digitizing* a signal. The main limitation of this step is that the space of possible values is limited, and therefore we have discrete steps in our data.

For example, the PFTL DAQ device establishes that when reading a value, it uses 10 bits to digitize the range of values between 0 V and 3.3 V. In the real world, the voltage is a real number that can take any value between 0 V and 3.3 V. In the digital world, the values are going to be integers between 0 and 1023 ($2^{10} - 1$). It means that if the device gives us a value of 0, we can transform it to 0 V. A value of 1023 corresponds to 3.3 V, and there is a linear relationship with the values in between.



The figure above shows a detail of how the digitalization looks like for a range of voltages. You see the discrete steps that the digital value takes for different voltages. Digitizing signals is a critical topic for anybody working in the lab. There is a whole set of ramifications regarding visualization, data storage, and more.

Particularly, the PFTL DAQ has a different behavior for reading than for setting values. The output channels take 4095 ($2^{12} - 1$) different values, i.e. they work with 12 bits instead of 10. Knowing the number of bits, also allows us to calculate the minimum difference between two output values:

$$\frac{3.3 \text{ V} - 0 \text{ V}}{4096} \approx 0.0008 \text{ V} = 0.8 \text{ mV} \quad (3.2)$$

The equation above shows how the resolution of the experiment is affected by the digitalization of the signals. We can't create voltages with a difference between them below 0.8 mV, and we are not able to detect changes below 3 mV. Later in the book, we come back to this discussion when we need to decide some parameters for visualizing our data.

Digitizing is everywhere. Digital cameras have a certain *bit depth*, which tells us which range of values they can cover or, in other words, their dynamic range. Oscilloscopes, function generators, acquisition cards, they all have a precise digital resolution. When planning experiments, we always need to keep an eye on these values to understand if the devices are appropriate for the measurement we want to perform.

? Exercise

We have used a for-loop to switch on and off the LED, and we have also displayed the voltage measured, but without units. Update the code so that instead of printing integers it prints the read value in volts.

3.8 Doing an experiment

At this stage, we can easily communicate with the device; we can set an output and measure a voltage. It means that we have developed everything that we needed to measure the current that goes through the LED. We only need to combine setting an analog output and then reading an analog input. Since we are going to develop this with a more consistent approach, we leave it as an exercise:

? Exercise

Write a method that allows you to linearly increase an analog output in a given range of values for a given step. **Hint:** the function `range` allows you to do this:

```
range(start, stop, step)
```

If you use this method, you should be able to see the LED switching gradually on.

? Exercise

Improve the method so that we can read analog values and store them once the measurement is complete. Returning the values can be a good idea so that we can use them outside of the object itself.

? Exercise

If you already have some experience with Python, you can also make a plot of the results. We cover this topic, later on, so don't stress too much about it now.

If you tried to solve the exercises, you probably noticed that by having classes, our code is straightforward to use. It would be simple to share it with a colleague that has the same device, and they can adapt and expand it according to their needs. We should also keep in mind that when working with devices, it may very well be that someone else has already developed a Python driver for it, and we can just use it. One of the keys to developing sustainable code is to compartmentalize different aspects of it. Don't mix the logic of a particular experiment with the capabilities of a device, for example, is precisely the topic of the following chapter.

Remember the Onion: We have discussed in the Introduction, that we should always remember the onion principle when developing software. If we see the outcome of these last few exercises, we notice that we are failing to follow the principle. We have added much functionality to the driver class that does not reflect what the device itself can do. The PFTL DAQ doesn't have a way of linearly increasing an output, and we have achieved that new behavior with a loop in a program. Therefore, the proper way of adding extra functionality would be by adding another layer to the program, as we see in the next chapter.

Before moving forward, it is also important to discuss other libraries that may come in handy. We are not the first ones who try to develop a driver for a device. The pattern of writing and reading, initializing, and many more that we haven't covered, were faced by many developers before ourselves. It means that there are libraries already available that can speed up a lot the development of drivers. Let's see some of them.

3.9 Using PyVISA

Some decades ago, prominent manufacturers of measurement instruments sat together and developed a standard called Virtual instrument software architecture, or VISA for short. This standard allows communicating with devices independently from the communication channel selected, and from the backend chosen. Different companies have developed different backends, such as NI-VISA, or TekVISA, but they *should* be interchangeable. The backends are generally hard to install and do not work on every operating system. But they do allow to switch from a device connected via Serial to a device connected via USB or GPIB without changing the code.

To work with VISA instruments, we can use a library available for Python called `pyvisa`. There is also a pure Python implementation of the VISA backend called `pyvisa-py`, which is relatively stable even if it is still work in progress. It does not cover 100% of the VISA standard, but for simple devices like the PFTL DAQ it should be more than enough. For complex projects, the solutions provided by vendors such as Tektronix or National Instruments may be more appropriate. In the next few paragraphs, we see how to get started with `pyvisa-py`, but it is not a requirement of the book. We decided to show it here to have it as a reference for other projects.

First, we need to install `pyvisa`, which is a wrapper around the VISA standard. Either with `pip`:

```
pip install pyvisa
```

Or with `conda`:

```
conda install -c conda-forge pyvisa
```

In case we don't have a VISA backend on our computer, we need to install one, and the easiest is the python implementation:

```
pip install pyvisa-py
```

or with `conda`:

```
conda install -c conda-forge pyvisa-py
```

There is also an interesting dependency missing: `PySerial`. Neither `Pyvisa` nor `Pyvisa-py` depend on `PySerial`. If we are going to communicate with serial devices, we should install that package

ourselves (and the same is true for USB, GPIB, or any other communication standard.) The documentation of pyvisa-py⁵ has handy information.

To quickly see how to work with PyVISA, we can start in a python interpreter, before going to more complex code. VISA allows you to list your devices:

```
>>> import visa
>>> rm = visa.ResourceManager('@py')
>>> rm.list_resources()
('ASRL/dev/ttyACM0::INSTR',)
>>> dev = rm.open_resource('ASRL/dev/ttyACM0::INSTR')
>>> dev.query('IDN')
'PFTL DAQ Device built by Python for the Lab v.1.2020\n'
```

We make explicit the backend we want to use by calling `ResourceManager`. In some cases, visa can automatically identify the backend on the computer. Then, we list all the devices connected to the computer. Bear in mind that this depends on the other packages that we installed. For example, we have only PySerial, and therefore pyvisa-py only lists serial devices. We can install PyUSB to work with USB devices, or GPIB, and so forth. The rest of the code is very similar to what we have done before. It becomes clear why we decided to call *resource* the communication with the device.

Pay attention to the `query` method that we use to get the serial number from the device. We didn't develop it. PyVISA already took care of defining query for us. Not only PyVISA takes care of the query method, but they have plenty of options that we can use, such as transforming the output according to some rules or establishing the write termination. If we were to follow the pyVISA path, we could start by reading their documentation⁶.

Why didn't we start with pyVISA? There are several reasons. One is pedagogical. It is better to start with as few dependencies as possible, so we can understand what is going on. We had to understand not only what commands are available, but we also had to be aware of the encoding and line termination. We made explicit the fact that to read from a device, we first have to write something to it. Once you gain confidence with the topics covered in this chapter, you can explore other solutions and alternatives. PyVISA is only the tip of the iceberg.

3.10 Introducing Lantz

Defining a class for your device was a massive step in terms of usability. You can easily share your code with your colleagues, and they can immediately start using what you have developed with really few extra lines of code. However, there are many features that we may want but that someone needs to develop. For example, imagine that we want to limit the number of times the output voltage can change, or we don't want to write to the device always the same value, the first time was enough.

We may want to establish some limits, for example, to the analog output values. Imagine that we have a device that can handle up to 2.5 V. If we set the analog output to 3 V, we would burn it. Fortunately, there are some packages written especially to address this kind of problem. We are going to mention only one because it is a project with which we collaborate: Lantz⁷. You can install it by running:

⁵<https://pyvisa-py.readthedocs.io>

⁶<https://pyvisa.readthedocs.io>

⁷<https://github.com/lantzproject>


```
pip install lantzdev
```

About Lantz

We introduce Lantz here for you to see that there is much room for improvement. However, through this book, we are not going to use it, and that is why it was not a requirement when you were setting up the environment. Lantz is under development, and therefore some of the fine-tuned options may not work correctly on different platforms. Using Lantz also shifts a lot of the things you need to understand under-the-hood, and it is not what we want for an introductory course. If you are interested in learning more about Lantz and other packages, you should check for the Advanced Python for the Lab book when you finish with this one.

Lantz is a Python package that focuses exclusively on instrumentation. We suggest you check their documentation and tutorials since they can be very inspiring. Here we just show you how to write your driver for the PFTL DAQ device using Lantz, and how to take advantage of some of its options. Lantz can do much more than what we show you here, but with these basics, you can start in the proper direction. You can also notice that some of the decisions we made earlier were directly inspired by how Lantz works.

Let's first re-write our driver class to make it Lantz-compatible, we start by importing what we need and define some of the constants of our device. We also add a simple method to get the identification of the device. Note that the first import is `MessageBasedDriver`, precisely what we have discussed at the beginning of the chapter.

```
from time import sleep

from lantz import MessageBasedDriver, Feat

class MyDevice(MessageBasedDriver):

    DEFAULTS = {'ASRL': {'write_termination': '\n',
                        'read_termination': '\n',
                        'encoding': 'ascii',
                        }}

    @Feat()
    def idn(self):
        return self.query('IDN')

dev = MyDevice.via_serial('/dev/ttyACM0')
dev.initialize()
sleep(1)
print(dev.idn)
```

There are several things to point out in this example. First, we have to note that we are importing a special module from Lantz, the `MessageBasedDriver`. Our class `MyDevice` inherits from the `MessageBasedDriver`. There is no `__init__` method in the snippet above. The reason for this is that the instantiation of the class is different, as we see later. The first thing we do in the class is to

define the `DEFAULTS`. At first sight, they look the same as the ones we have defined for our driver. The `ASRL` option is for serial devices. In principle, we can specify different defaults for the same device, depending on the connection type. If we were using a USB connection, we would have used `USB`, or `GPIB` instead of, or in addition to `ASRL`.

The only method that we have included in the example is `idn` because, even if simple, it already shows some of the most interesting capabilities of Lantz. First, we can see that we have used `query` instead of `write` and `read`. Indeed, Lantz depends on pyVISA, so what is happening here is that under the hood, you are using the same command that we saw in the previous section. Bear in mind that Lantz automatically uses the write and read termination.

An extra syntactic thing to note is the `@Feat()` before the function. It is a `decorator`, one of the most useful ways of systematically altering the behavior of functions without rewriting. Without entering too much into details, a decorator is a function that takes as an argument another function. In Lantz, when using a `Feat`, it checks the arguments that you are passing to the method before actually executing it. Another advantage is that you can treat the method as an attribute. For example, you can do something like this `print(dev.idn)` instead of `print(dev.idn())` as we did in the previous section.

? Exercise

Write another method for getting the value of an analog input. Remember that the function should take one argument: the channel.

To read or write to the device, we need to define new methods. If you are stuck with the exercise, you can find inspiration from the example on how to write to an analog output below.

```
output0 = None

[...]

@Feat(limits=(0,4095,1))
def set_output0(self):
    return self.output0

@set_output0.setter
def set_output0(self, value):
    command = "OUT:CH0:{}".format(value)
    self.write(command)
    self.output0 = value
```

What we have done may end up being a bit confusing for people working with Lantz and with instrumentation for the first time. When we use `Features` in Lantz, we have to split the methods in two: first, a method for getting the value of a feature, and then a method for setting the value. We have to trick Lantz because our device doesn't have a way of knowing the value of an output. When we initialize the class, we create an attribute called `output0`, with a `None` value. Every time we update the value of the output on channel 0, we are going to store the latest value in this variable.

The first method reads the value, pretty much in the same way than with the `idn` method. The main difference here is that we are specifying some limits to the options, exactly as the manual specifies for the PFTL DAQ device. The method `set_output0` returns the last value that has been

set to the channel 0, or `None` if it has never been set to a value. The `@Feat` in Lantz, forces us to define the first method, also called a `getter`. It is the reason why we have to trick Lantz, and we couldn't simply define the `setter`. On the other hand, if the setter is not defined, it means that you have a read-only feature, such as with `idn`. The second method determines how to set the output and has no return value. The command is very similar to how the driver you developed earlier works. Once we instantiate the class, we can use the two commands like this:

```
print(dev.set_output0)
dev.set_output0 = 500
print(dev.set_output0)
```

Even if the programming of the driver is slightly more involved, we can see that the results are clear. A property of the real device also appears as a property of the Python object. Remember that when you execute `dev.set_output0 = 500`, you are changing an output in your device. The line looks very innocent, but it isn't. Many things are happening under the hood both in Python and on your device. I encourage you to see what happens if you try to set a value outside of the limits of the device, i.e., try something like `dev.set_output0 = 5000`.

The method we developed works only with the analog output 0. It means that if we want to change the value of another channel, we have to write a new method. It is both unhandy and starts to violate the law of the copy/paste. If we have a device with 64 different outputs, it becomes incredibly complicated to achieve a simple task. Fortunately, Lantz allows us to program such a feature without too much effort:

```
_output = [None, None]

[...]

@DictFeat(keys=[0, 1], limits=(0, 4095, 1))
def output(self, key):
    return self._output[key]

@output.setter
def output(self, key, value):
    self.write(f'OUT:CH{key}:{value}')
    self._output[key] = value
```

Because the PFTL DAQ device has only two outputs, we initialize a variable `output` with only two elements. The main difference here is that we don't use a `Feat` but a `DicFeat`, which will take two arguments instead of one: the channel number and the value. The `keys` are a list containing all the possible options for the channel. The values, such as before, are the limits of what we can send to the device. The last `1` is there just to make it explicit that we take values in steps of 1. We can use the code in this way:

```
dev.output[0] = 500
dev.output[1] = 1000
print(dev.output[0])
print(dev.output[1])
```

And now it makes much more sense, and it is cleaner than before. We can also check what happens if we set a value outside of what we have established as limits. The examples above only scratch the surface of what Lantz can do. Sadly, at the moment of writing, the documentation for the latest version of Lantz is missing. The best starting point is the repository with the code: <https://github.com/lantzproject>.

With the examples above, there is a small step to understand how to solve the following:

Exercise

Write a `@DictFeat` that reads a value of any given analog input channel.

3.11 Conclusions

We have covered many details regarding the communication with devices. We have seen how to start writing and reading from a device at a low level, straight from Python packages such as *PySerial*. We have also seen that it is handy to develop classes and not only plain functions or scripts. We have briefly covered *pyVISA* and *Lantz*, two Python packages that allow you to build drivers in a systematic, clear, and easy way. The rest of the book doesn't depend on them, but you must know of their existence.

It is impossible in a book to cover all the possible scenarios that you are going to observe over time in the lab. You may have devices that communicate in different ways. You may have devices that are not message-based. The important point, not only in this chapter but also throughout the book, is that once you build a general framework in your mind, it is going to be much easier to find answers online and to adapt others' code.

Remember, documentation is your best friend in the lab. You always have to start by checking the manual of the devices you are using. Sometimes some manufacturers already provide drivers for Python. Such is the case of National Instruments and Basler, but they are not the only ones. Checking the manuals is also crucial because you have to be careful with the limits of your devices. Not only to prevent damages to devices but also because if you employ an instrument outside of the range for which it was designed, you can start generating artifacts in your data. When in doubt, always check the documentation of the packages you are using. *PySerial*, *PyVISA*, *PyUSB*, *Lantz*, they are quite complex packages, and they have many options. In their documentation pages, you can find a lot of information and examples. Moreover, you can also check how to communicate with the developers because they are very often able to give you a hand with your problems.

3.12 Addendum 1: Unicode Encoding

We have seen in the previous sections that when we want to send a message to the device, we need to transform a string to binary. This process is called encoding a string. Computers do not understand what a letter is, they just understand binary information, 1s and 0s. It means that if we want to display an `a`, or a `b`, we need to find a way of converting bytes into a character, or the other way around if we want to do something with that character.

A standard that appeared several years ago is called ASCII. ASCII contemplates transforming 128 different characters to binary. Characters also include punctuation marks such as `.`, `!`, or `:`, and numbers. 128 is not a random number, but it is 2^7 . For the English language, 128 characters are enough. But for languages such as Spanish, which have characters such as `ñ`, French with its different accents, and without even mentioning languages that use a non-Latin script, forced the appearance of new standards.

Having more than one *standard* is incompatible with the definition of a standard. Imagine that we write a text in French, using a particular encoding, and then we share it with someone else.

That other person does not know which encoding we used and decides to decode it using a Spanish standard. What will the output be? Very hard to know, and probably very hard to read by that person. It is without considering what would happen if someone writes in Thai and shares it with a Japanese, for example.

It still happens with some websites which handle special characters very poorly. Depending on how people configure databases, some characters which do not conform to the English script are just trimmed. This unbearable situation gave rise to a new encoding standard called, as the title of this section suggests: **Unicode**.

Unicode uses the same definition as `ascii` for the first 128 characters. It means that any `ascii` document looks the same if decoded with Unicode. The advantage is that Unicode defines the encoding for millions of extra characters, including all the modern scripts, but also ancient ones such as Egyptian hieroglyphs. Unicode allows people to exchange information without problems.

Thus, when we want to send a command to a device such as the PFTL DAQ, we need to determine how to encode it. Most devices work with ASCII values, but since they overlap with the Unicode standard, there is no conflict. Sometimes devices manufactured outside of the US may also use characters beyond the first 128, and thus choosing Unicode over `ascii` is always an advantage. In Python, if we want to choose how to encode a string, we can do the following:

```
var = 'This is a string'.encode('ascii')
var1 = 'This is a string'.encode('utf-8')
var2 = 'This is a string with a special character ñ'.encode('utf-8')
```

Utf-8 is the way of calling the 8-bit Unicode standard. The example above is quite self-explanatory. You may want to check what happens if, on the last line, you change `utf-8` by `ascii`. You can also see what happens if you decode with `ascii` a string encoded as `utf-8`.

One of the changes between Python 2 and Python 3 that generated some headaches to unaware developers was the out-of-the-box support for Unicode. In Python 3, you are free to use any utf-8 character not only in strings but also as variable names, while in Python 2, this is not the case. For example, this is valid in Python 3:

```
var_ñ = 1
```

If you are curious to see how Unicode works, the Wikipedia article is very descriptive. Plus, the Unicode consortium keeps adding new characters based on the input not only from industry leaders but also from individuals. You can see the latest emojis added and notice that some were proposed by local organizations that wanted to have a way of expressing their idiosyncrasies.