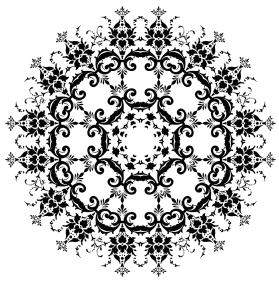
# PYTHON FOR THE LAB

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

**BY**

## AQUILES CARATTINO

**PhD IN PHYSICS, SOFTWARE DEVELOPER**



**AMSTERDAM**
**2018**

# Contents

# 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 is focused on showing you how things can be done and not into a lot of theory on how programming works in general. This, of course, leads 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 website[1] 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 are not answered in the book, don't hesitate to shout in the forum. Continuous feedback is the best way to improve this book.

## 1.1 Why building your own 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 what

[1] https://www.pythonforthelab.com

the software can perform while not pursuing innovative ideas. Once you learn how to develop your own programs, you will be limited only by your imagination.

Let's assume for a moment that you work with a microscope. Your task is to acquire spectra of some bright nanoparticles. Your job is to focus on each particle and trigger a spectrometer. Change some parameters and repeat. This is tedious, slow, and error-prone. Focusing on a bright spot can be easily done by a computer, as well as triggering a spectrometer.

Another example, imagine that you are inspecting a sample, looking for a special pattern. But once you find it, you have a few seconds to trigger a measurement. Unfortunately, you will never be able to do this by hand. Even in optimized conditions, you need more than a few seconds between acquiring, analyzing and making a decision.

The problems listed above are just some of the realities that researchers face in the lab. In some cases, better software will increase the throughput of a setup. You can leave it running all night, generating better statistics. Sometimes it will open the door to experiments that would have been impossible otherways. Custom software is the way in which you can bring your instruments to a new life, and your creativity to a completely new stage.

## 1.2 What are you going to learn

With this book we want you to gain a first insight into the world of developing software for controlling experiments. We will start by discussing how to set up a proper development environment and quickly focus on building a driver for a device. Drivers are the fundamental building blocks of any program for controlling experiments and therefore it is a topic in which you need a solid foundation.

Once you have the driver in place, you will be tempted to start performing measurements. However, there are some programming patterns that will guarantee the success of your software in the long run. You will be introduced to the Model-View-Controller paradigm and its implications for instrumentation software. You will learn about extracting logic from functionality and laid the foundations for reusable code.

Once the code is working, you will learn how to perform measurements and save the data, including metadata, to be able to repeat a measurement if needed. You will plot your results and try to interpret them. Up to this point, your program will be running through the

command line, with messages being printed to screen. This book is going even one step further.

Thanks to the solid structure that you have achieved up to this point, building a Graphical User Interface will be relatively easy. You will learn how to design a window that accepts input from the user and displays back a plot with the results of the measurement. You will be able to save data from the window and learn how to verify the input parameters before passing them to an instrument.

More importantly, you will learn some strategies that will make your programs more reusable, not only by you in the future, but also by others who have access to the same setup. This book is the starting point for laying out a program that is guaranteed to last. The most difficult task when developing software for the lab is knowing where to start. This book is exactly that, a starting point for your future endeavors.

## 1.3 Who is this book intended for

When I started this book, I wanted to make it possible for people with very different programming levels to achieve a common goal. I found that the only way to do this is to be a bit dogmatic with the explanations. For example, if you want to do this you should type that. By starting the development through a real need, a question arising from the real world, you can anticipate what is going to happen in the future and therefore the motivation level will remain high enough as to continue thinking in a solution.

Generally speaking, you will need a basic level of programming. You should understand what is an *if-statement*, why would you use a *for-loop* or a *while-loop*. We are going to make extensive use of classes in Python, but without complicating it too much. In the appendixes of the book, you can find a quick walkthrough about how to work with classes and other syntactic examples of Python. You should start by reading that chapter if you are not familiar with Python. If you find that the contents are too complicated for your level, you should consider starting with an easier book, perhaps on how to use Python for analyzing data.

Even if not strictly required, you will also need a basic understanding of how an experiment works and a bit of physics. Namely, you need to understand that an experiment starts by designing a way in which you can alter your object of study in a controlled manner and monitor some signals while you do so. The experiment we are going to perform throughout this book is a common example in many, many different fields: we are going to change an analog output while we measure an

analog input. If you are a student of the Python for the Lab workshop, the experiment is going to be measuring the I-V curve of a diode.

Varying an output voltage and recording an input is a common task in innumerable experiments from different fields. Changing a voltage can be related to displacement through a piezo stage, you can change the intensity of light, you can alter temperature, and the list goes on and on. Measuring a voltage can be equivalent to determining a current, the intensity of light, a displacement, a force, etc. Look around in your own lab and probably will find examples.

## 1.4 PFTL DAQ Device

In order to follow the book, you will need a device that acts as a data acquisition board. If you joined one of the workshops, the device will be provided by the instructor. If you got this book online and would like to buy one of PFTL DAQ's, please contact us. The devices are open source/open hardware, and you can find the instructions for building one on our website. The core of the course is that you will develop code that makes it easy to exchange devices. Therefore, 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.

The requisites for the device are to be able to acquire analog signals, and also to generate analog outputs. The device will be connected to the computer via the USB port. Having a real device is paramount for students to translate the knowledge acquired to their own realities.

Building software for the lab has a reality component not covered in any other books or tutorials. The fact that you are interacting with real-world devices, which are able to 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 About this book

This book is a compendium of the best strategies that I have found while developing software for different laboratories. I have tried to justify every decision made, but some are rooted more in an aesthetic choice than in an architectural one. You are free to change and improve whatever you think will lead you to better (or faster) results. I have tested what I propose and I am very confident with the outcome. If you

are not very experienced, I suggest you start in the same way as the book does and later you can find your own way.

When developing software, you will probably realize that it is going to be used by another person either at the same time or in the future. In many labs, people come and go and therefore you cannot count on being around for answering questions regarding your programs, or vice versa, you will have to understand someone else's code. If you adhere to some common standards, everybody's life will be much easier. In Python For The Lab, you will find a lot of recommendations regarding how to make your code clear, to you and to others.

You can find many tools that will help you in keeping your code organized, such as tools for version control and documenting. They are, however, a topic completely different from what we intend to cover in this book. There will be some hints as to where to start. Remember that many of the problems can be solved by clear policies established in each lab, such as where to put the code for sharing with other members, how to document, etc. If there is nothing like that where you work, you should seriously consider discussing it with your team members.

The book is divided into different chapters where you will develop different skills. Each chapter is aimed at tackling a specific problem that we want to solve, and that is clearly stated in the chapter objectives. It may happen that a chapter is easier for you because it is what you are used to doing, and you will be inclined to skip it. However, bear in mind that each chapter builds on the previous one and therefore you should be sure that you have the needed code in place.

## 1.6 Why Python?

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

Another factor to take into account is the fact 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, Python is accompanied by a myriad of professionally developed tools such as numpy, scipy, scikit, etc. If you are looking for professional support,

you can look at companies such as Anaconda, which provide customers with high-quality advise and troubleshooting.

Another interesting factor to take into account is that Python is easy to read. If you grab code written in Python, probably you will understand quite quickly what it does, even if you have never seen the language before. As soon as you have a bit of experience, you will be able to learn from much more complex code. This makes it an ideal language for scientists who want to leverage the knowledge generated in the past, with very short startup time.

However, for experimentalists, there is a big downside when considering Python. If you search online for instructions on how to control your experiment, you will find really few sources of information, and even less if you focus on Python alone. Fortunately, this is changing thanks to an evergrowing number of people developing open source code and writing very useful documentation. Python can achieve all the same functionality of Lab View, the only limitation is the existence of drivers for more complex instruments. With a stronger community, companies will realize the value of providing drivers for other programming environments.

From a purely programmatic point of view, one can point out that Python doesn't need to be compiled in order to run, meaning that changes done to the code can be immediately observed. Its syntax is very clear and intuitive; people making the switch from Matlab will find no problems in understanding the code. People making the switch from LabView, however, will have a higher learning curve converting from a drag-and-drop approach to a normal typed language. In any case, once some syntactic elements are known, a lot of things can be programmed.

A very important aspect of Python is that it is platform independent. Since Python is interpreted by a special program every time we run it, it will produce the same output regardless of whether we run on Linux, Windows or Mac. The python interpreter is responsible for adapting the way our code works under the hood to the different architectures. Therefore, we can develop on our office PC running Windows, run the code in the lab that has Linux installed and share it with a colleague who prefers Mac.

Many of the doubts that can arise when programming with Python will not be specific to interfacing with instruments, but more general regarding how to achieve certain functionality. Normally you can find answers to all of them just by searching online. And if not, you can always refer to the forum of the Python For The Lab website. It is a great place to share your doubts, especially because it will help us improve the contents of this book.

Finally, the relevance of Python for experimentalists is that the lan-

guage exceeds the confinement of the lab. Python allows you to control your experiments and analyze your data. It also allows you to build websites, develop machine learning algorithms, automatize your daily tasks, and many more interesting things. Learning Python increases your employability chances if you ever decide to step out of academia or if you want to move from an instrumental background and focus into analysis or beyond.

## 1.7 The Onion Principle

When you start developing software it is very hard to think ahead. Most likely you have a small problem that you want to solve and you just go for it. Later on, it may turn out, and this is especially true for people who work in labs, that that small problem is actually something worth investigating. Your software will not be able to handle the new tasks and you will need to improve it. Having a proper set of rules in place will help you have code that can adapt to your future needs while keeping you productive in the present. I like to call those rules the Onion Principle.

The rules I am talking about are not rules written in stone and you will not find them stated in a book (by the way, you won't find them here). I am talking about a state of mind that will empower yourself to develop better, clearer and more expandable code. Sitting down and reflecting is the best you can do, even more than sitting down and typing. When dealing with experiments you have a lot of things to ask yourself, what do you know, what do you want to prove, how to do it. Only then you will sit down to write a program that responds to your needs.

If you build something that cannot be expanded, it will become useless very soon. When you don't really know what may happen with your code, you should think ahead and structure it as an onion, in layers. I am not claiming that it is something that happens naturally, but you can develop your own set of procedures to ensure that you are developing future-proof code. Once you get the handle on it, it won't take you longer than being disorganized and not having the proper structure. You will 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 you start with a simple task at hand, you want to solve it quickly and not spending hours developing useless lines of code just thinking what if. I am all in for that kind of solutions; however, a strong foundation is always important. Taking shortcuts just because you don't want to create a separated file will give you more headaches even in the short term. You should build code that is robust enough to support for expansion later on. In the same

way that you take steps while performing an experiment, you should take steps when developing software.

One of the key elements to achieve a great onion-like approach in Python is to use classes. They are concise elements with clear functionality and very easy to document. They can be imported and even expanded without changing the original code, as we will see by the end of this chapter. If you come from the data analysis world, it may very well be that you never developed your own classes, but this is about to change with this book. You shouldn't be afraid of them; you should just try to understand them because I can guarantee that once you get the handle of the, you won't be able to stop thinking in their own terms.

I will not make special emphasis on why I ask you to do the things in a certain way throughout the book. However, if you pay attention you will notice that each step is incremental. You won't have to start all over again to add a new feature that you didn't think about. And this ends up being easy because the foundation given by the previous step is very solid. I am not saying that there is only one way of achieving the same results, but I will be just showing you a way of doing things. When you have the chance to improve something without much effort, you should take it, you will be very grateful to your past self.

In the end, all the common practices and development patterns that I can show you, are thought just to make you save time not only in the long run but also next week. I have, just as you are about to do, started developing software for acquiring a very simple signal, an analog input generated by a photodiode. It didn't take long until I wanted to move a piezo stage with an analog output and suddenly I needed a way of doing 2D and 3D scans of my sample. Sadly for me, there was no one around who could show me a way to being organized and clear with my code.

This book is based not only on my own experiences working in different labs but also on the experiences of the people who surrounded me, who received my code. Sometimes we cannot anticipate the ramifications that our work will have, but I can assure to you that if you start as a small onion, layer by layer, each new path that opens up will be very satisfying.

## 1.8 Where to get the code

The code that you are going to develop in this book is freely available on Github (`https://github.com/PFTL/SimpleDaq`). We encourage you not to look for the solution ahead of time. The code is there in order to give you the chance to explore alternatives. For example, the online code is

well documented. You can see how we have decided to add comments in order to implement the same solutions on your programs. If you are experienced with Python, you can directly study the code to see how we solved some of the common problems people encounter in the lab.

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

## 1.9 Organizing a Python for the Lab Workshop

Python for the Lab was born with the intention of bringing 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 there is always the possibility to adapt them to the specific needs of your group.

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

# Chapter 2

# Setting Up The Development Environment

## 2.1 Objectives

In order to start developing software for the lab, you are going to need different programs. The process to install programs is different depending on your operating system. It is almost impossible to keep an up-to-date detailed instruction set for every possible version of each program and for every possible hardware configuration. Therefore, follow the steps provided below carefully. When in doubt, 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, probably you have encountered that there are different *distributions* which are worth discussing. Broadly speaking, Python in itself is a text document in which it is specified what to expect when certain commands are encountered. This gives a lot of freedom to develop a different implementation of those specifications, each one with different advantages. The *official* distribution, i.e. the distribution maintained by the Python Software Foundation, is one of the two that we recommend in this book and can be found on Python.org. We will discuss step by step how to install it the following sections. The official 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 also start including the official Python distribution. However, some developers started to release Python distributions which are optimized for certain tasks. For example, Intel releases its own version of Python which is specially designed to use multi-core architectures. There are other versions of Python, such as Pypy, Jython, Iron Python, etc. Each one has its own merits and drawbacks. Between this wealth of options, there is one that is very popular amongst scientists called Anaconda, which is worth discussing and which is the second option we will cover in this book.

Python can be expanded through external packages that can be developed and made publicly available by anyone. Some time ago, the python package manager was very limited, and a group of developers decided to develop a tool that allowed people to install complex libraries, especially those needed for numerical computations. The main requirement was to have a package manager which could install also libraries not written in Python. This is how Anaconda was born and is still thriving nowadays. Anaconda is a distribution of Python which comes with *batteries included* for scientists. It includes many Python libraries by default but also other programs.

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 a reality that many researchers already have Anaconda installed on their computers and thus it is worth mentioning how to work with it. If you are starting from scratch, the decision is yours. If you use plain Python, you will end up with a system with just the libraries needed for achieving our goals. Once you require special libraries which are harder to install or create some dependency issues, then you can explore what Anaconda offers.

## 2.3 Installing Python

You are going to start by installing Python itself. This book is based on Python version 3.6. For most features discussed in this book, earlier versions such as 3.3 or 3.4 are also going to work, but versions 2.x are going to fail. If it happens that you have a Python version 3.x installed and don't want to update it, follow the book and see if it gives any errors. Let us know when that happens. Remember that Python version 2.x is finishing its life during 2020. It is therefore very wise to start developing code in a language that will be supported beyond that time-frame.

### 2.3.1 Python Installation on Windows

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

Download the file that corresponds to Python 3.6 or later to your hard drive. Once it is complete, you should launch it and follow the steps to install Python on your computer. Be sure that **you select Add Python 3.6 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 Python was correctly installed, you are going to 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 will allow you to interact with your computer by writing commands instead of using the mouse. We will see some of the options you have.

To start it, just go to the Start Button and search for the Command Prompt (it may be within the Windows System apps). A shorter way is to just press Win+r. It will open a dialogue called Run, allowing you to start different programs. Type `cmd.exe` and press enter. A black screen should pop up, this is the Command Prompt of Windows.

In the Command Prompt, you can do almost everything what you can do with the mouse. You will notice that you are in a specific folder on your computer. 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.

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

```
Python 3.6.3 (default, Oct  3 2017, 21:45:48)
[GCC 7.2.0] on Win64
Type "help", "copyright", "credits" or "license" for more information.
```

It will show which Python version you are using and some extra information. When you do this 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 will notice immediately its similarities. 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. In such a case, the lines that don't have the `>>>` in front are the ones corresponding to the output. Later on, we are going to work also with files, in which case there is not going to be a `>>>` in front of each line.

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 3.6 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 will 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 python.exe is located. You will 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. You will find 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 in order for it to refresh the settings. Try again to run python.exe and it should be working now.

### 2.3.2 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 will need to install Python 3 on your system. Ubuntu users can do it by running:

```
sudo apt install python3
```

Each Linux distribution will have 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 therefore you won't need to explicitly include the *3*. In case there is an error, try first running only `python` and checking whether it recognized that you want to use Python 3.

### 2.3.3 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 that counts with more than 100000 packages available. The easiest way to install and manage packages is through a command called **pip**. Pip will

fetch the needed packages from the repository and will install 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, therefore 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, most likely you will find that you have two commands, pip2 and pip3. You should use the latter in order to install packages for Python 3. On Windows, most likely you will need to use pip.exe instead of just pip. If for some reason it doesn't work, you need to follow the same procedure that was explained earlier to add python.exe to the PATH, but this time with the location of your pip.exe file.

Installing a package becomes very simple. Linux users should type:

```
pip install package_name
```

Windows users should instead type:

```
pip.exe install package_name
```

Where `package_name` has to be replaced by what you want to install. For example, if you would like to install a package called numpy, you would do:

```
 pip.exe install numpy
```

**Note**

Before installing the rest of the packages, I suggest you read the section on the Virtual Environment. It will help you keep clean and separated environments for your software development.

Pip will automatically grab the latest version of the package from the repository and will install it on your computer. To follow the book, you will 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

All the packages can be installed with pip without much trouble. If you are in doubt, you can search for packages by typing `pip search package_name`. Normally, it is not important the order in which you install the packages. Notice that since dependencies will also be installed, sometimes you will 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 in order 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.

### 2.3.4 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. This will allow you, for example, to use experimental libraries without overwriting software that other programs use on your computer. Isolated environments allow you, for example, to 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 important to isolate different environments: you will be developing a program with a certain set of libraries, each with its own version and installation method. One day you, or another researcher who works with the same setup, decide 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 will 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 exact 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 in order 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 overcomes all the mentioned difficulties. 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 will be done 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 is incredibly simple.

You can create isolated working environments for developing software, for running specific programs or to perform 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.

### 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 will guide you through with the installation and configuration. If there is something 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 important because it will allow 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 will also create a folder called Testing, in which all the packages and needed programs are going to be kept. If everything went well, you should see that your command prompt now displays a (Testing) message before the path. This means that you are indeed working inside your environment.

Once you are finished working in your environment just type:

```
deactivate
```

And you will return to your 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 will be installed. You can try to install the packages listed before, such as numpy, PyQt, etc. 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.

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

```
pip freeze
```

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

**Warning**

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

If you are a Power Shell user, first, you should install another wrapper:

```
pip install virtualenvwrapper-powershell
```

And most likely you will 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. This should allow the wrapper to work. You can

repeat the same commands that were 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 will create your 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 very easy to install the Virtual Environment package. Depending on where you installed Python in your system 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 pip install virtualenv
sudo -H pip install virtualenvwrapper
```

If you are on Ubuntu, you can also install the package through apt, although it is not recommended:

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

To create a Virtual Environment you will 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 will tell you what is being triggered when you actually run python3 on a terminal. Replace the location of Python in the following command:

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

Which will create a folder, normally /.virtualenvs/Testing with a copy of the Python interpreter and all the packages that you need, including

pip. That folder will be the place where new modules will be installed. If everything went well, you will see the `(Testing)` string at the beginning of the line in the terminal. This lets 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 will create the Virtual Environment wherever you are on your computer and not in the default folder. This can be handy if you want, for example, to share the virtual environment with somebody, or place it in a very specific 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 effectively you are isolated from your main 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 will give you a list of all the packages that you have installed within that working environment and their exact versions. So, you know exactly what you are using and you can revert back if anything goes wrong. Moreover, for people who are really worried about the reproducibility of the results, keeping track of specific packages is a great way to be sure that everything can be repeated at a later time.

### 2.3.5 Installing Qt Designer 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 will depend 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`.

### 2.3.6 Installing Qt Designer 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.

## 2.4 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. Normally it will auto-detect your operating system and offer 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. Normally, there won't be any issues, you just need to be aware of the fact that other programs which rely on Python will use the Anaconda version and not the stock version.

Similar to the different distributions of Python, Anaconda also comes in two main 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 very specific requirements, we strongly recommend downloading just Anaconda.

### 2.4.1 About using Anaconda

In the previous sections, we have seen that to install packages you use the command `pip`. However, pip only allows you to install Python packages. If you are trying to use a package which depends on external libraries written in a different language, pip will fail.

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

Anaconda developed a package manager which is able to install software written in any language, not only in Python. For scientific computing, many packages depend on specific libraries, compiled for different operating systems and architectures. Sometimes it may even be required for proper installation of a package to compile a library. Anaconda refers to the procedure of installing a program as a *recipe*. Those recipes have all the instructions for automatically installing not only Python packages but almost any program you may need for scientific computing.

For following this book, we will need only Python-based libraries, and most come already bundled with Anaconda. If you are in Windows, you need to be sure to be running within the **Anaconda Prompt**. It is a terminal on which the default Python installation points to the Anaconda installation. If you are on Linux, this is achieved during the installation and you can check it by looking at the file `~/.bash_profile`. To install packages, the command is:

```
conda install numpy
```

The packages which are installable like that are those maintained by Anaconda. Those are the official packages which come with a *certification* of quality. Many companies, for example, allow people only to install packages officially supported by Anaconda. The packages required for following this book, are all but one available on the official Anaconda repository. These are the packages which you should install with the command shown above:

- 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

The only package which cannot be installed in this way is called *Pint*, which is very useful for handling units, but it is not crucial for our program to run. The standard place to find packages provided by the community is called *conda-forge*. For example, to install Pint, you would run the following command:

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

### 2.4.2 Conda Environments

If you went through the section on Virtual Environments with Python and found them useful, you probably will be glad to see that Anaconda includes its own version of virtual environments which are also incredibly powerful. Conda Environments allow you to isolate different development configurations from each other. In this way, if you are developing a program which requires a special version of a library, you will be sure you don't alter other programs when upgrading/downgrading a library.

It is **very important** not to mix conda environments and virtual environments, since they work in very different ways. To create a virtual environment with Anaconda, the command is:

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

Where you should replace `myenv` with the name of the environment you want to create. To activate the environment, on Windows you have to execute, within the Anaconda Prompt:

```
activate myenv
```

While on Linux the command becomes:

```
source activate myenv
```

Anaconda and its possibilities is a full topic that needs to be studied separately and it goes beyond the scope of this book. The documentation on the official website is very extensive and is worth a read. Lastly, it is important mentioning that when you create a conda environment,

you can explicitly set the version of Python you want to use, even if it is not the one you have installed on your computer:

```
conda create -n myenv python=x.x
```

Where you will need to replace `python=x.x` by the version of Python you would like to use. This is great if you are debugging code and would like to see whether it works on older versions of Python, or if a program you need to run is not available for newer versions, etc.

## 2.5 Using Git for Version Control

Generally speaking, version control means keeping track of all the changes within a project. The project can be a software development project, but also writing a paper or maintaining a blog (or even writing this book). When you do version control by hand, probably you implement solutions such as changing the name of the file, which leads to generating monstruosities which look like file-rev1-by-me_yesterday.doc, file-rev2-by-john_today.doc, etc. You can quickly see that it becomes a hassle as soon as you have more than one.

Fortunately, there is software that can help you keep the history of the changes in a very efficient way: by looking only at the differences and not storing the entire file again and again. If you add one line to a file, you can just store that extra line and where it should be added, instead of copying the entire file to a new location. This will allow you to go back in time and recover exactly how things were in the past. Moreover, version control programs are great tools for collaborating in groups.

Git is one of the programs you can use for version control. It is in itself an entire world, and therefore it cannot be covered completely in this book. You can check Python for the Lab's website because there is going to be more material regarding the use of Git for scientific environments. To follow the book you don't need to use Git, but you are encouraged to do so in order to practice and add one more tool to your box.

The majority of Linux distributions come with Git already packaged and installed, you can test it by just going to a terminal and running:

```
git help
```

If this doesn't work, you can install git by using APT (or the package manager of your distribution):

```
sudo apt install git
```

On Windows, you can download git from Git-SCM. Install it, especially paying attention to add git to the path and integrating it with the Command prompt and Powershell. In the downloads page, you can see that there are also some programs that provide a graphical interface for working with git repositories. They are not mandatory, but you are welcome to try them out.

Git is a distributed version control; it means that you can track the changes locally on your own computer, without the need to connect to a remote server. Each project that you create is normally called a repository, a place where everything is stored. At some point in time, you will want to share your code with your colleagues or with the public. You will need a resource outside of your computer where you can place and synchronize the repositories in order to grant access to other developers. Some of those resources have web interfaces that allow you to manage the repositories and their users in a very convenient way. Such websites are, for example, Github, Bitbucket, or Gitlab.

Creating an account on those websites is free of charge but with some restrictions. The free version of Github, for example, doesn't allow you to create private repositories with more than 3 collaborators. Bitbucket allows you to create private repositories but puts a limit on the team size unless you start paying. Gitlab is not only a web interface, but it can also be installed on your own server. This means that perhaps your university or institute already provides a host for Git repositories. Github is the place were major open source programs can be found. If you are planning to generate code that can be interesting to others, you should consider it. If you want to keep your files secret, for example, while writing a paper, you should consider Bitbucket.

### 2.5.1 Quick Introduction to Working with Git

The best way to start working with Git is through an example that guides you through the different stages of the process. This guide assumes that you have created an account on Github.

Go to Github.com and click on the + symbol at the top right corner of the page, next to your picture. Select New repository. You can name it whatever you like, I suggest you call it PythonForTheLab. Leave all the options as they are; the repository is marked as public and is not initialized with a README file because you are going to create it later.

You will see that Github is kind enough to show you how to start working with it in a few simple steps. Create a folder on your computer

where you would like to keep your code, you can call it PythonForTheLab to keep the consistency. From the command line go to that folder and type:

```
git init
```

This command will initialize a local git repository. You will notice that there is a folder called .git and that will be responsible for managing your local repository. Within the main folder, create a text file and call it README.md. The extension md specifies the syntax that you are going to use in the file, and it is a default for readme instructions. Inside the file write whatever you like, for example:

```
# Python For The Lab

Welcome to the Readme of Python For The Lab.
```

If you go back to the terminal and type:

```
git status
```

You should get an output that looks like the following:

```
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

   README.md

nothing added to commit but untracked files present (use "git add"
to track)
```

Which is quite descriptive if you read it carefully; it is telling you that there are new files that are not being tracked by git. Therefore you should add them in order to start following their changes. We can do so by running:

```
git add README.md
```

If you check the status again you will see that the message has changed. Now that you have started tracking the file, you can commit the changes. You should think each commit as a snapshot of your code when you commit you are creating a stamp to a specific time in

your development. Each commit is associated with a message that will allow you to understand what you have been doing when you took that snapshot.

```
git commit -m "Added Readme to the repository"
```

If you run again git status you will see that there is nothing new since your last commit. If you want to see the history of the latest changes to your code, can run:

```
git log
```

Notice that you will see the list of the latest commits, their descriptions, and the author of those changes. So far, we have been working only locally on one computer. One of the advantages of Git is that it also allows you to track your code remotely. By using an external server to host your code, you will be able not only to back it up but also to synchronize between different devices. To send the changes to the repository that we have created on Github, we need to configure a remote in our local repository. As the name suggests, it is a remote location to which to send the code. You can type (replace with your own information):

```
git remote add origin git@github.com:Username/PythonForTheLab.git
```

With the command, you have configured a remote location called origin. Git allows you to have several remote locations, each with a different name. If you want to send the changes to the repository you need to push them:

```
git push -u origin master
```

The option -u is used only the first time you do a push and is not mandatory. Go to your repository on Github and check how nicely the Readme file is being displayed. One of the many advantages of the Github website is that it allows you to modify the files directly within it. Open the README.md file, click on the small pen in the upper right corner and add few extra lines to it. At the bottom of the page, you will see that it tells you to make a commit. It is exactly the same idea of what you have done from the command line, but directly to your remote repository. Add some descriptive information and save it.

In Git, there are no differences regarding local or remote repositories. They are all the same, but they are used in different ways. A remote repository is accessible from different computers, while local repositories are not. However, you can change the files in a remote

repository in exactly the same way that you change your local files. Since the files in the remote location have changed, you need to download the changes before continuing with the work. You need to pull the changes:

```
git pull origin master
```

Open again the README.md file and you will see that the changes you did online appear in your local file. The number of things that you can try is endless and I would seriously advise you to look around for more examples.

## 2.6 Editors

To complete the Python For The Lab book, you will need a text editor. As with a lot of decisions in this book, you are completely free to choose whatever you like. However, it is important 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, it is very basic simple and simple. You can have several tabs opened 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.

If you are looking for something a bit more complete, you can look into Atom. One of its nicest features is that you can extend them through plug-ins. If you look around, you will find that there are a lot of extensions that can accommodate your needs. Atom is very well integrated with Git, and therefore all the work of committing, pushing, etc. can be done directly within the editor. Both programs allow you to work on multiple files and projects.

Besides text editors, there is another category of programs called IDE's, or Integrated Development Environments. These programs include not only a text editor but also tools to check the consistency of your code, they warn you if you forgot to close a parenthesis, they are able to refactor your code or to clean it up. The most powerful one for Python is **Pycharm**, and is the one we recommend for following this book. Pycharm is complex, it allows you to control your entire development processes from creating virtual environments, to run tests, to deploy your code. The professional version of Pycharm is not free,

but it has a community edition which is very powerfull and it can give you a good idea of how to get started. If you are going to use Pycharm fror educational purposes and you have an e-mail from a University, you can install the professional edition for free through a special license to which you apply online.

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 to keep using it, you will see that it integrates very nicely with your workflow. Visual Studio is available not only for Windows but also for Linux and Mac. It has some nice features for inspecting elements and help you debug your code. The community edition is free of charge. Support for Python is a relatively new addition and therefore you will not find many tutorials online in which Visual Studio is used. However, Microsoft has released several video-tutorials showing you how to get the best out of their program.

A third option is called **Sublime**. It is a very popular editor that can be installed for free. The only catch is that every certain amount of time a pop up will appear and you will be prevented from working until you click on it. If the pop up is hindering your productivity, you should seriously consider the license. Sublime became a standard program within a certain community of developers (mostly those using a Mac). It is quick, and extensible through plugins. Depends how you look at it, one advantage/dissadvantage is that the program was not designed specifically for Python, meaning that you can use Sublime for programming in any language. On the other hand, it means that setting it up for Python specifically will require you to go trhough tutorials and recommendations for plugins.

Getting the time to familiarize yourself with tools such as Pycharm really pays out. If you make a trivial mistake like forgetting a `:`, or if you forget to import a package, IDE's will warn you. Moreover, they integrate automatically with virtual environments and many other tools, making your development a breeze. The only downside of IDE's over plain text editors is that they consume much more resources of your computer. If you are going to follow one of the three-day workshops, the recommendation is to install Pycharm. You can make use of the 30-day trial period and decide later whether you want to purchase, downgrade to the community edition or try something else.

Remember that always, the choice is yours. Whichever you make it is going to be more than appropriate for following the book. If you choose an IDE, it is important to practice before starting to develop a complex program in order to clear all the doubts you may face when dealing with the editor. It is also important to stick with your choice

for a while. Once you start developing with one editor, you should use it for a long time before changing to another. This is the only way in which you can really appreciate the differences and make an informed decision.

**Warning**

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 will respect Python's stylistic choices. Notably, Notepad++ comes configured by default to use tabs instead of spaces. This 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 their behavior is different (you can't compare a camera to an oscilloscope) but they also communicate in different ways with the computer. In this chapter, you 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

Devices can be split into different categories depending on how they communicate with a computer. One of the broadest categories is that of devices which communicate through the exchange text messages. Perhaps you didn't stop to think before how you can control an oscilloscope from your computer, but the idea is that the user sends a specific command, i.e. a message, and the device answers with specific information, another message.

To have an idea of how commands look like, you can check the manuals of devices such as oscilloscopes or function generators. Both Tektronics and Agilent have very complete sets of instructions. If you search through their websites, you will find plenty of examples. A command 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.

Some devices that belong to the category of *message based* are oscilloscopes, lasers, function generators, lock-ins, and many more. The device that was developed 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 own device or contact us and we may be able to offer you one already programmed[1].

Remember that message based refers only to how the information is exchanged with the computer, and not to the actual connection with the device. A message based device can be connected via RS-232, USB, GPIB, TCP/IP, etc. Be aware, however, that it is not a reciprocal relation: not all devices connected through RS-232, USB, etc. 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.

In the introduction, we have discussed that the objective of the project you are building through this book, is to be able 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, you will learn everything you need to perform your 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 Message Based Devices

There are two basic operations that can be done with a message based device: `write` and `read`. `write` means sending a command, typically a string, from the computer to the device, while reading means getting a message back from the device. When writing, we are normally starting an action on the device. For example, if you would communicate with a laser, you can send a command for switching on or off the output beam. If you were working with an oscilloscope, you could send a command to auto setup itself. In both examples, the command will trigger a series of changes on the device, but it will not necessarily give back any feedback. On the other hand, you can ask something from a device, for example, we can check the output power of a laser or the time divisions of an oscilloscope. This procedure will take two steps: we `write` a command asking for a value and we `read` the value from the device. This double step procedure is also called to send a `query` to a device.

[1] courses@pythonforthelab.com

Most message-based instruments come with clear documentation regarding which commands can be sent and what responses we should expect. If you have any manual at hand, you will notice that you also have some extra information regarding the connection, such as the baud rate or the line ending. The information that needs to be supplied depends on the type of connection of the device. The manufacturer gives all the important parameters needed to achieve a correct communication with the device. Many devices (but not all) follow a standard called SCPI If you check the manuals of different devices, you may notice that some structure in the commands is repeated.

An important parameter for message-based devices is the line ending. When a device is receiving a command it will read the input until the device knows that it has finished. Imagine that you are sending a value to a device, for example, you want to set the output wavelength of a laser to 1200 nm. The command could look like `SET:WL:1200`, however, the device needs to know when it has received the last number. It is of course not the same to set the laser wavelength to 120 nm than to 1200 nm. Each device specifies how to determine the end of a message. Normally it is going to be a `new line` character or a `carriage return`. Translated to python they are `\n` or `\r` respectively. But some devices take both or may specify any other character.

**Warning**

If you have the example device that comes with this course, you can follow the steps as they appear in the text. If you are using a different device, you have to adapt the commands to reflect what you have at hand.

**Warning**

Different operating systems behave slightly differently, especially regarding port naming. Throughout the book, we try to be both Windows and Linux compatible, with the focus on Linux.

When you start working with a new device, you have to start by checking its manual. You need to understand how the device communicates with the computer and which commands are available. Moreover, you need to know your device in order to know its limitations and capabilities. It is common to find in the lab fuses burned (and hopefully not a burned device) because a user didn't check the maximum current that can be supplied. The manual for the PFTL devices can be found in the appendix, PFTL DAQ Device Manual. It is short but contains similar information to what you would find in any other device's manual pages.

In the manual, you can find a general introduction to the device and some specifications regarding the communication. If you pay attention, you will notice that even though the device is connected to the USB port, it will act as a general serial device. This is very common behavior for smaller or older devices, which provide USB connectivity for convenience. Always read the manual to be sure how your device communicates with the computer and check how it is connected. Often, the same device offers more than one option for connecting.

To communicate with the PFTL DAQ device, we are going to use a package called `PySerial`, which you should have already installed if you followed the Setting Up Chapter. The first thing we can do is to list all the devices connected to the computer. This will allow us to understand how to identify yours. In a terminal type the following command and press Enter:

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

The command should print a list of all the devices that you have connected to your computer through the serial port. If you already plugged the device you want to use and you are not sure which one it is, you should unplug, run the command again, plug it back and see the differences. Once you gain a bit of experience you may start realizing which device is the one you want to use without plugging/unplugging.

Note

**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 will be something like COM1, on Linux and Mac, it will be something like /dev/ttyS1. In modern computers, you will hardly find any RS-232 connections, and most likely you are using a USB hub for them. This means that there is no physical connection straight from the device into the motherboard. The numbering can change if you plug/unplug the cables. The PFTL device, since it acts as a hub for a serial connection, can show the same behavior.

Once you identify on which port the device is connected, you can start sending some commands to it. Devices normally have an instruction to identify themselves, and the PFTL DAQ is no exception. It is always a good idea to start by checking that everything is working properly, and getting the serial number from the device seems like a good idea. Please note that in the code on the book, it always appears an example port. You should change it for what you have found in the previous step.

> **Note**
>
> For the examples with few lines of code, you can either write everything to a file and execute it by writing in the Terminal `python file.py`, or you start an interactive python console by typing `python` and then the rest of the commands directly onto the command line.

```
import serial

device = serial.Serial('/dev/ttyACM0') # <---- CHANGE THE PORT!
device.write(b'IDN\n')
answer = device.readline()
print('The answer is: {}'.format(answer))
device.close()
```

The code above is enough for illustrating how communication with a device happens. First, we import the `PySerial` package, noting that it is done by `import serial` and not `import PySerial`. Then you open the specific serial port of the device (line 3). Bear in mind that serial devices can maintain only one connection at a time. If you try to run the line twice it will give you an error letting you know that the device is busy. This is important if, for example, you are running two programs at the same time.

Once the connection is established, you send the **IDN** command to the device. Note that there are some extra details. The `\n` at the end is the `newline` character that the manual specifies. It is the way to tell the device that we are not going to send more information afterward. In order for the serial communication to work, you also need to include a `b` before the string. Adding the `b` in front of a string is one of the ways of telling Python to encode a string to binary code. As you can imagine, devices don't really understand what an *A* is, they understand only about 1's and 0's. Therefore, you need to transform any string such as `'IDN'` to bytes before sending it to the device. If you are curious about this, there is a lengthier discussion in section 3.10.

Once the command is sent, and action is going to be triggered on the device. In this case, the action is that the device will look into its own memory and will answer back with its serial number. To recover this information, you read the answer from the device with the `readline` method. Notice that `readline` will wait until a `newline` character is found. Once you receive the information from the device, you can print the answer to be sure that you are communicating with the device you wanted.

In the code snippet above, we have used the `.format` syntax to put one string into another. In Python, there are different ways of achieving the same functionality, and we are going to discuss them quickly, in order for you to understand how they work. In older versions of Python, the only way of doing substitutions was by using the `%` syntax. For example, you could do something like this:

```
a = 3
print('a = %s' % a)
# a = 3
```

What you see above is that `a` is an integer. On the second line, that integer gets transformed to a string, and embedded within the `'a = %s'` string, replacing the space of `%s`. This syntax is very flexible, allowing you to control also how to transform more complex objects, how many decimals to show, etc. In more recent versions of Python, a new syntax option was introduced, the `.format`. To achieve the same result, you would do:

```
a = 3
print('a = {}'.format(a))
```

In this case, we are substituting the `{}` by the value of `a`. Again, this syntax will also allow you to control with great accuracy how you transform any object to a string. Newer projects tend to favor this syntax over the previous one. In most cases, the result is the same. Finally, the third alternative is to use something called f-strings. This is a newer feature in Python and is starting to be widespread, mainly because of its syntactic clarity:

```
a = 3
print(f'a = {a}')
```

What is happening above, is that when we add an `f` before a string, python will look for the `{a}` and will replace the variable it finds by its string representation. In this book we are going to use the `.format` syntax because it is by far the most used one, but keep in mind that f-strings are very powerful and they are gaining more and more traction in the community.

**Exercise**

What happens if you use `read()` instead of `readline()`? What happens if you call `readline()` before writing the `IDN` command? If the program freezes (most likely sooner or later you are going to break things up), you can stop the execution by pressing Ctrl+C

**Exercise**

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

Reading the value of an analog port from the device is as easy as asking for the serial number, we just need to issue a different command. If you check the manual, you see that the appropriate command is **IN:**. Therefore, you can do something like this:

```
[...]
device.write(b'IN:CH0\n')
value = device.readline()
print('The value is: {}'.format(value))
```

Note that the `[...]` means that there is some code suppressed for brevity. In the example above, the lines suppressed are those in which you have to import PySerial and start the communication with the device. You have to interpret what the code is doing in order to decide at which position you would like to add it. In principle, the lines that we used to get the serial value from the device can also be kept, but you have to be sure not to close the communication with the device if you are trying to get a value out of it.

When the documentation is clear, working with a device is straightforward. Unfortunately, this is not the case for most real-world devices. If you are curious, I suggest you check the manual of an Oscilloscope (even if you don't own one) like a Tektronics or an Agilent. They are very thorough in their descriptions. However, devices such as digital cameras from Hamamatsu have little to no documentation. If you ever find yourself developing software for those devices, you will need a high dose of patience.

**Exercise**

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

Exercise

Now that you know how to set values and how to read values. Acquire the I-V curve of the diode. This is a difficult exercise, aimed at showing you that it doesn't take a long time to be able to achieve a very important goal.

## 3.4 Going Higher Level

In the previous section, we have seen that for communication with a device there are a lot of steps that one needs to take into account, such as the line ending, the encoding, etc. Moreover, it becomes very unhandy every time we want to send a command to have to type it. In the application we are developing we will need to set a voltage plenty of times and we will need to read values very often. If you still remember the *Onion Principle*(sec. 1.7), it is now the time to start applying it. If you completed the last exercise, probably you have written a lot of 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 is not very sustainable for the future, especially if you are going to share the code with someone else.

Now that you know how to communicate with the device, you can transform that knowledge into reusable Python code by defining a class. Classes have the advantage of being easy to import into other projects, are easy to document and to understand. 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, however, you will be able to follow the rest of the chapter and understand what is going on as you keep reading.

At this point, if you were developing code directly on the command line, it is important to stop it and move the code to a file. Writing to the command line has severe limitations, the most important one is that you are not able to save what you are doing in order to use it later on. Create a new, empty file, called **simple_daq.py**, and write the following into it:

```
import serial


class Device():
```

```
    def __init__(self, port):
        self.rsc = serial.Serial(port)

    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. Before going into the discussion of what the code above means, we can see how you would use it. At the end of the file, add the following code:

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

In the first line, you create an object `Device` using a specific port. Note that when you want to get the serial number from the device, you can now simply do `dev.idn()`. It looks much neater than having to explicitly send the command with the line ending, etc.

Having a class and not a plain script makes your code much easier to share, to reuse and to maintain. The Device class defines a method called `__init__` that will be executed every time the class is instantiated, i.e., every time an object is created when you do `Device('/dev/ttyACM0')`. This method creates the serial connection and stores it in a variable called `rsc` (short for *resource*). The `\_\_init\_\_` method takes two arguments: `self` that refers to the class itself and `port`, which stands for the port on which the device is found. You see that this is the port used to establish the serial connection.

The class Device has a second method called `idn` that can be used to get the identification from the device. The method takes only one argument, `self`, which refers to the object itself. This means that all the parameters defined as `self.something` and all the methods defined in the class are going to be available within this function. Notice that the method first writes to the device and then it reads from it. The value that is recovered from the device is returned to the user.

**Exercise**

Once you read the serial number from the device, it will 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.

You have just learned the basics of writing a class for the device. You can also write methods for reading an analog input or generating an output. The most important thing to do is to decide what inputs/outputs each method needs. For example, reading a value only needs the channel that you want to read. Setting a value needs not only a channel but also the value itself. Also, reading a means that the method will return something. When you set an output, there is not much to return to the user.

**Exercise**

Write a method `get_analog_value` 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.

### 3.4.1 General methods to reduce the amount of repetition

If you finished the exercises above, you may have already seen that `idn`, `set_analog_value`, and `get_analog_value` repeat the structure of writing to the device appending the proper line ending, encoding, and reading from the device. When you start to develop programs there is a principle called **DRY**, which stands for don't repeat yourself. Sometimes it is hard to see where you are repeating not just code, but a pattern. Don't repeat yourself is not a matter of just saving you from typing more lines of code. It is a way of reducing errors and making your code more maintainable. Imagine you upgrade the device and now it requires a different line ending. You would need to go through all your code to find out where the line ending is used and change it. If you would specify the line

ending in only one location, changing it would require just to change one line.

The examples mentioned above are simple operations, developed just to showcase what DRY means in practical terms. Keep in mind that more complex patterns would require careful consideration. Adding a line ending is just the beginning of what it is being repeated. There is a second-order level of abstraction possible. Both the `idn` and `read_analog_value` have to first write to the device and then read from it. It is possible to abstract the procedure into a new one called `query`, which will take care of everything. Enough words, let's get to work to see how to accomplish what we discussed above.

You can update the class by adding the defaults as a dictionary, just before the `__init__` method, like this:

```
class SimpleDaq():
    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 a lot of 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 is worth making it explicit in case newer devices need a different option. We also specify the read and write timeouts, which are optional parameters, but they will help to avoid the program to freeze in case the device is non-responsive.

It is normally good practice to separate the instantiation of the class with the initialization of the communication. This pattern gives us finer control over what we are doing. We can rewrite the class like this:

```
def __init__(self, port):
    self.port = port

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(0.5)
```

You can see that there are some major changes to the code, but the arguments of the `__init__` method are the same. We do this to ensure that if there is code already written, it will not fail because of a change in the number of arguments of a method. When you do this kind of changes, 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, you will need to do `{dev.initialize()}`. In this way, our code is more flexible, because it allows us to instantiate the class but we do not start the communication with the device immediately. You can also see that we have used almost all the settings from the `DEFAULT` dictionary to start the serial communication.

There is something important to point out: the highlighted line. There is a `sleep` statement in order to delay the execution of the rest of the program after initialization of the communication. The delay is only needed to prevent the rest of the program from communicating with the device before a proper channel has been established. This is a safeguard, but in most cases, it will not be needed. It is important for you to be aware that not having a small delay between opening the serial communication and sending the first command can give you some hard time tracking down errors.

So far the only difference with the previous code is the `__init__` method. We have to improve the rest of the class. You already know that for message-based devices there are two operations: read and write. However, you will only read after a write (remember, you should ask something from the device first.) It is possible to update the methods of the class to reflect this behavior. First, we can update the `write` method in order to take into account the line ending and encoding.

```
def write(self, message):
    msg = message + self.DEFAULTS['write_termination']
    msg = msg.encode(self.DEFAULTS['encoding'])
    self.rsc.write(msg)
```

The code above splits the creation of `msg` into two steps in order

to make it clearer. But nothing prevents you from doing everything in just one line. The `write` method is more useful than the serial write. It takes the message, appends the proper termination and encodes it as specified in the `DEFAULTS`. Then, it writes the message to the device exactly as we did before. See that the communication with the device is achieved through the resource `self.rsc` that is created with the method `initialize`. There is a common pitfall with this command. What happens if the user of the driver forgot to initialize the communication before trying to write to the device?

**Exercise**

Improve the `write` method in order to check whether the communication with the device has been initialized.

When developing code you always have to keep an eye on two people: the future you and other users. It may seem obvious now that you will initialize the communication before attempting anything with the device, but in a month, or a year, when you dig up the code and try to do something new, you are going to be another person, and you won't have the same ideas in your mind as right now. Adding safeguards are, on 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.

If you don't add a warning or error message to the code above, and you run the program before initializing the communication with the device, you will get an error message like the following:

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

Which doesn't mean anything at all. While if you do things properly, the the error that will appear on screen could be:

```
Exception: Forgot to initialize the Device driver at port COM1
```

You have improved the `write` method, now it is time to improve `read`. Remember that so far you have used `readline` because the device was appending a new line character at the end of each answer. The vast majority of devices behave in this way, but one has to be aware that is not always going to be the case. You can take into account this and make a more flexible method:

```
def read(self):
    line = "".encode(self.DEFAULTS['encoding'])
    read_termination = self.DEFAULTS['read_termination']\
```

```
        .encode(self.DEFAULTS['encoding'])

    while True:
        new_char = self.rsc.read(size=1)
        line += new_char
        if new_char == read_termination:
            break
    return line.decode(self.DEFAULTS['encoding'])
```

The code above starts by defining an empty string with the proper encoding. You have to do this in order to accumulate the message into that string. For convenience, we also defined the `read_termination` variable, encoded properly. Remember that if you don't encode the `read_termination` you won't be able to check whether the character which is returned by the device is the termination or not. Once you have these two variables in place, you start an infinite loop `while True`. Since you don't know how long the message is going to be, you need to read from the device as long as needed. Within the loop, you read from the device one character at a time ( `size=1` ). This character is appended to `line` and if it matches the termination character, the *while* loop is ended with the command `break`. Once the message is complete, it is decoded and returned to the user.

Remember that using a `while True` statement can be a risk. There is no guarantee that the loop is ever going to end. If it doesn't end, your program is going to freeze in that block of code and you will have to terminate it externally by pressing Ctrl+C. This can happen, for example, if your device is using a different line ending than the one you specified, or if the encodings do not match. If you are developing code for sensitive equipment, or for users who do not want to deal with this kind of low-level problems, you need to build safeguards. We will see one of the possibilities later.

The `read` method defined above looks much more complex than just using `readline`. It is also much more flexible when it comes to customization. At this point, writing your own `read` method is more an intellectual exercise than a real need. Most likely you will be tempted not to go into all those troubles and just use the `readline` but if you keep reading the chapter you will see that everything makes sense in the context of a larger objective.

**Note**

If you face the situation of having a loop that doesn't end or a program that takes too long to complete, remember that you can stop the execution by pressing Ctrl + C in the terminal where the program runs.

**Exercise**

You may have seen that we have added a `read_timeout` to our class, but we didn't use it yet. Find a way to stop the loop if the `read_timeout` is reached and issue a `Warning`.
**Hint**: the package time will give you the current time with a high degree of accuracy. You can import it using `from time import time` and use it with `time()`.

We have developed a robust way of writing to and reading from the device. The only missing part is to condense together both methods into a new one called `query`. Since we already did all the heavy work in the previous two methods, the `query` is going to be surprisingly simple:

```
def query(self, message):
    self.write(message)
    return self.read()
```

There are no secrets nor caveats, we took care of everything by writing a proper write and read method. Now that you have these methods available, you should update the rest of the code.

**Exercise**

Re-write the `idn` and `get_analog_value` methods to make use of the new `query` method.

**Exercise**

Re-write the `set_analog_value` method to make use of the new `write` method.

What we have completely forgotten to add to our code is a proper way of closing the communication with the device. We can call that method `finalize` and will look like this:

```
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`.

## 3.5 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 you know from reading the manual, the PFTL DAQ device can generate two analog outputs, each from 0 to 3 Volts. You will need to create a method that allows you to change those voltages. If you did it in the previous exercise, you can skip this section, but bear in mind that if the syntax is not the same, you will need to adapt the code later on. The method can look like this:

```
def set_analog_value(self, channel, value):
    write_string = 'OUT:CH{}:{}'.format(channel, value)
    self.write(write_string)
```

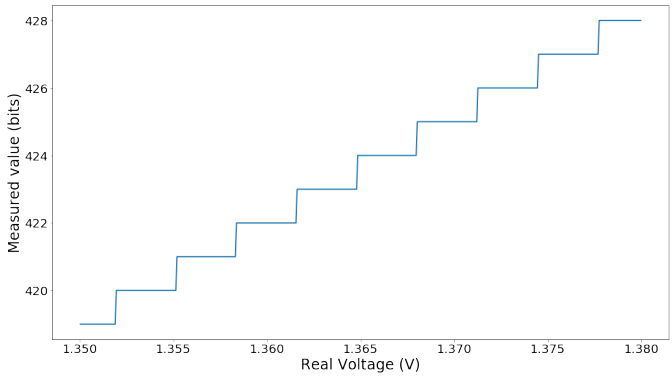
The method `set_analog_value` takes two arguments, the `channel` number and the `value` to output. You format that information into a string as specified in the manual of the device, and you call it `write_string`. This string is then passed to the method `write`, that will take care of appending the line ending and encoding of the message. Doing it in two steps, first defining the string and then calling the method is only a matter of readability, especially for longer commands is very handy.

We are ready to do something with the DAQ card other than reading noise. We need to hook up the LED following the instructions provided in the appendix and check what happens when you change the voltage output from the DAQ. What you should observe is that the LED becomes brighter or dimmer depending on the voltage you set as output. Moreover, you should observe that for a range of small voltages there is no light emitted. What you are observing is a known phenomenon of diodes, also called the I-V curve. Measuring this behavior is the whole experimental task that we are going to address.

### 3.5.1 Analog to Digital, Digital to Analog

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

For example, the PFTL DAQ device establishes that when reading a value, it uses 10 bits to digitize the range of values between $0\,\mathrm{V}$ and $3.3\,\mathrm{V}$. In the real world, the voltage is a real number that can take any value between $0$ and $3.3$. In the digital world, the values are going to be integers between $0$ and $1023$ ($2^{10}-1$). This means that if the device gives us a value of $0$, we can transform it to $0\,\mathrm{V}$. A value of $1023$ will correspond to $3.3\,\mathrm{V}$ 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 very important topic for anybody working in the lab. There are 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:

$$\frac{3.3\,\mathrm{V}-0\,\mathrm{V}}{4096}\approx 0.0008\,\mathrm{V}=0.8\,\mathrm{mV} \tag{3.1}$$

The equation above shows you how the resolution of your experiment is affected by the digitalization of the signals. You will not be able to

create voltages with a difference between them below $0.8\,\mathrm{mV}$, and you won't be able to detect changes below $3\,\mathrm{mV}$. Later in the book, we will come back to this discussion when you need to decide some parameters for visualizing your data.

Keep in mind that digitizing is everywhere. Digital cameras have a certain *bit depth*, which tells you which range of values they can cover, i.e. their dynamic range. Oscilloscopes, function generators, acquisition cards, they all have a certain digital resolution. You will always be sure that the resolution you get is enough to observe the properties you are interested in.

## 3.6 Doing an experiment

Up to now, we have developed everything that we need to actually measure the current that goes through the light emitting diode, you only need to combine setting an analog output and then reading an analog input.

**Exercise**

Write a method that allows you to linearly increase an analog output in a given range of values for a given number of steps. Don't forget to add a delay between each update.

**Exercise**

Write a method that is able to record a given analog input while an analog output increases linearly.

**Exercise**

Make a plot of your results.

Doing an experiment at this stage is left as an exercise to the reader. In the following chapter, we are going to see how to perform experiments in a much more flexible way. If you try to solve the exercises, you will see that by having classes, your code is much more reusable. It is very easy to share with a colleague that has the same device, and it can be adapted and expanded. You also should 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 you 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 special experiment with the capabilities of a device, for example.

**Remember the Onion**: We have discussed in the Introduction, that one should always remember the onion principle when developing. If you see the outcome of the exercises you just finished, you will notice that you are failing to follow the principle. You have added a lot of 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, you have achieved that extra behavior with a loop in a program. If you start transferring the logic of your own experiment to the driver class, you will start creating a gigantic class that others will not be able to (re-)use.

When developing software, especially when dealing with devices, one has to separate what the device can do and what extended functionality we can achieve. A scan of an analog output is a consequence of sequential changes of a value, and therefore we should have those ideas in a different portion of the code. In the beginning, it may be very hard to realize what is the true reason for separating the code like this, but with time it will become clearer. When you develop the driver for a device, such as what we have been doing in this chapter, we have to keep as close as possible to what the manual specifies.

## 3.7 Using PyVISA

What we did in the previous section is very interesting, but as you may have guessed, we are not the first ones who want to achieve the same. Plus, you may ask yourself what happens if you have a device that can communicate through different connections, not only serial. You will have to develop a new driver for each one and that is not handy.

Some decades ago, big 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 normally hard to install and do not work on every operating system. But they do allow to switch from a device connected via COM to a USB connection without changing the code.

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 enough. If you are working on more complex projects, you should definitely check some of the

solutions provided by bigger vendros such as Tektronix or National Instruments. In the next few paragraphs, we will show you how to get started with pyvisa-py, but it is not a requirement of the book. You can use it as a reference for your other projects.

First, you need to install `pyvisa`, which is a wrapper around the VISA standard. This package will allow you to control devices directly from within Python:

```
pip install pyvisa
```

If you already have a backend on your computer, then this is all you need. If you don't, and whish to install the pure-python implementation, you can do it like this:

```
pip install pyvisa-py
```

If you are working on a clean environment, you will also need to install PySerial since it is not marked as a dependency of pyvisa-py. Pyvisa-py relies on lower-level libraries for communicating with devices, but it does not require you to install them beforehand. You will have to install them as you need them. The documentation of pyvisa-py has very useful information.

If you are using **Anaconda**, the commands above have to be replaced by:

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

To see who to work with PyVISA, let's start in a python interpreter directly, 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')
'General DAQ Device built by Uetke. v.1.2017\n'
```

If you follow the code, you can see that after importing `visa`, we start the resource manager specifying that we want to use the Python backend. If you have any other backend installed on your computer, this is the place to define it. Then, we list all the devices connected to the computer. Bear in mind that this will depend on the other packages that you installed. For example, we have only PySerial, and therefore

pyvisa-py will only list serial devices. You can install PyUSB in order to work with USB devices, etc. The rest of the code is very similar to what we have done before.

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 you can use, such as transforming the output according to some rules, establishing the write termination, etc. If you were to follow the pyVISA path, you can start by reading their documentation.

**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 you can understand what is actually going on. Now you know exactly what commands need to be sent, you are aware of the encoding and line termination. You are also aware of the fact that to read from the device, you first have to write something to it. The other reason is practical. PyVISA works great with the National Instruments backend, but it is hard to install. Pyvisa-py is still slightly work in progress and to keep the consistency of the book through time, it was better not to ask it as a dependency.

Once you gain confidence with the topics covered in this chapter, you will be able to explore other solutions and alternatives.

## 3.8 Introducing Lantz

Defining a class for your device was a very big 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, as soon as you want to develop drivers for a new device, you will find yourself repeating a lot of the things you have done right now. Setting the line ending, the encoding, etc. And it only works for serial devices, when you want to add a USB or GPIB device you will have to re-think everything.

Moreover, there are more sophisticated properties that can be improved. For example, you could use some cache in order to avoid reading too often from a busy device or re-setting a property that didn't change. We could also set some limits, for example to the analog output values. Imagine that you have a device that can handle up to $2.5\,\mathrm{V}$. If you set the analog output to $3\,\mathrm{V}$ you would burn it.

Fortunately, there are packages that were written especially to address this kind of problems. We are going to mention only one because

it is a project with which we collaborate: Lantz. You can install it by running:

```
pip install lantzdev
```

> **Note**
>
> We introduce Lantz here for you to see that there is a lot of 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 properly 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 are finished with this one.

Lantz is a Python package that focuses exclusively on instrumentation. We strongly suggest you check their documentation and tutorials since they can be very inspiring. Here we will just show you how to write your own 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 will be able to start in the proper direction.

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 will also add a simple method to get the identification of the device. Note that the first import is a `MessageBasedDriver`, exactly what we have discussed at the beginning of the chapter.

```
from lantz.messagebased import MessageBasedDriver
from lantz import Feat

class MyDevice(MessageBasedDriver):

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

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

```
if __name__ == "__main__":
    dev = MyDevice.via_serial('/dev/ttyACM0')
    print(dev.idn)
```

There are several things to point out in this example. First, we have to note that we are importing a very special module from Lantz, the `MessageBasedDriver`. Our class `MyDevice` inherits from the `MessageBasedDriver`. If you are unsure of what inheriting means, I suggest you check the appendix C. Surprisingly, there is no `__init__` method in the snippet above. The reason for this is that the instantiation of the class is different, as we will see later. The first thing we do is to define the `DEFAULTS` of our class. At first sight, you probably see that they really look the same to the ones we have defined for our own driver. The `ASRL` option is for serial devices. In principle, you can specify different defaults for the same device depending on the connection type. If you were using a USB connection, you 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, you 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 the write and read termination are automatically used by Lantz.

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 will check 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 will be able to 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.

In order to read or write to the device, you 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. Since our device doesn't have a way of knowing the value that an output was set to, we have to trick it a bit. When we initialize the class, we will have 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 is for reading the value from the device, 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 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. When you use `@Feat` in Lantz, we are always forced to define the first method, also called a `getter`. This is 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 you instantiate the class, the two commands can be used like this:

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

Even if the programming of the driver was slightly more involved, you can see that the results are very clear. A property of the real device

appears also as a property of the Python object. Remember that when you execute `dev.set_output0 = 500` you are really changing an output in your device. The line looks very innocent, but it isn't. A lot of 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`.

You may have noticed that the method works only with the analog output 0; this means that if you want to change the value of another channel, you will have to write a new method. This is both unhandy and starts to violate the law of the copy/paste. If you have a device with 64 different outputs it would become incredibly complicated to achieve a simple task. Fortunately, Lantz allows you to program such a feature with not too much effort:

```
output = [None, None]

[...]

@DicFeat(keys=[0, 1], values=(0, 4095, 1))
def output(self, key):
    return self.output[key]

@output.setter
def output(self, key, value):
    self.write('OUT:CH{}:{}'.format(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, as before, are the limits of what you can send to the device. the last `1` is there just to make it explicit that we take values in steps of 1. You 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. You can also check what happens if you set a value outside of what you have established as limits. The examples above only scratch the surface of

what Lantz can do. It is strongly suggested that you check their website and follow the guides and examples. Even if you don't use Lantz for your driver, you can get a lot of inspiration regarding how to work with your code and how to improve your programming strategies. We have covered how to set an analog output because it was the hardest task. You can do the following exercise:

**Exercise**

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

**Note**

Lantz does a great work working with units as well. If you read the documentation, you will see that you can specify the units directly in the `@Feat` and Lantz will take care of the conversions to the natural units of your device. It will also check that the input is within the limits you have specified.

## 3.9 Conclusions

In this chapter, we have covered a lot of details regarding the communication with devices, 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 it is important that you 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 messaged based, etc. 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.

## 3.10 Addendum 1: Unicode Encoding

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

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

As you can imagine, having more than one *standard* is incompatible with the definition of a standard. Imagine that you write a text in French, using a special encoding, and then you share it with someone else. That other person does not know which encoding you 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 a person. Remember that this doesn't even consider the possibility of having text written in a non-Latin script such as Russian or Chinese.

If you are old enough, you may remember that some websites were not rendering correctly, and it was very common to find weird characters on e-mails. This was all due to the mess that different string encodings generated. I even remember that some web hosting companies did not allow hosting Chinese websites because of the overhead that decoding Chinese characters imposed on the servers. This unbearable situation gave rise to a new encoding standard called, as you guessed,

**Unicode**.

Unicode uses the same definition than ascii for the first 128 characters. Which means that any ascii document will look exactly the same if decoded with Unicode. The advantage is that Unicode defines encoding for millions of extra characters. This includes all the modern scripts, but also ancient ones such as Egyptian hieroglyphs, allowing people to exchange information without problems. Unicode is also the standard for most emojis with which you may be very familiar.

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 will be no conflict. Bear in mind that 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 you want to choose how to encode a string, you 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'`.

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.