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 in charge of asking the user for some input, performing a measurement and displaying the results. However, for many experimentalists, the software that comes packaged is not sufficient and is barely customizable.

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 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 comments section of the website.

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 daily tasks. However, when it comes to controlling a setup, many scientists prefer to be bound by what 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.

Imagine 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 few seconds to trigger a measurement. However, you will never be able to do this by hand. Even in optimized conditions, you need more than 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 only way in which you can bring your instruments to a new life.

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 functional, 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. So far everything is happening through the command line, with messages being printed to screen. But you are going to go one step further.

Thanks to the solid structure that you have achieved up to this point, building a Graphical User Interface will be relatively easier. You will learn how to design a window that accepts input from the user and displays a plot with the results to the user. 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 by others in 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 book, you can find a refreshment of how to work with classes and other syntactic examples of Python. You should read that chapter through, especially 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 using the device that accompanies the course, 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 Uetke DAQ Device

Together with the book, you are going to receive a device that will act as a Data Acquisition board, or DAQ for short. The device is able to acquire analog signals but 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, that is able to change the state of an experiment, makes the development process much more compelling. The Uetke DAQ is a toy device, easy to replace, but capable of performing quality 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 that you start in the same way as the book does and later on 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 free and open source and therefore there is no overhead when implementing it. Compared to the price of the licenses of programs such as Matlab or LabView, we can make some compromises while using Python. On the downside, Matlab and Lab View have a relatively good professional support and communities built around them. Answers to questions regarding Python have to be found by searching online, most likely in Stack Overflow.

If you search online for instructions on how to control your experiment with Python, you will find really few sources of information. 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 as 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.

Python doesn't need to be compiled to execute a script, 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. Thus we can develop on our office PC running Windows, run the code in the lab that runs Linux and share it with a colleague who prefers a 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 a 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.

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. Here is where having the proper set of rules in place will help you and I like to call them the Onion Principle.

The rules I am talking about are not rules written in stone and that you can find clearly stated in a book (by the way, you won't find them here). I am talking about a set 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. Variables that are not self-descriptive, lack 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 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 started like a small onion, layer by layer, they are going to be very satisfying.

1.8 The Agile Process

The Agile software development describes a particular approach to software development methodology based on iterative development. Its core value is that requirements and solutions evolve through a collaborative process between developers and customers. The principles of the Agile methodology are summarized in four statements in its manifesto. They are:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Agile in itself doesn't establish any particulars regarding programming but just how to develop a workflow and a working environment. In the last few years, Agile has attracted a lot of attention, extending its concepts away from software development, and moving into project management. Even though the core principles were drawn from a programming context, scientific projects could also benefit from the Agile ideas, but with some reservations.

Some of the main points of an Agile process, such as responding to change over following a plan are a routine reality in a lab. Plans are hard to follow because a researcher cannot be sure of what lays ahead. More interesting questions arise or some things just don't work as expected. Being flexible to overcome obstacles and to change focus when more important questions can be answered, is a needed attribute of successful researchers.

Collaboration over negotiation is also common in a scientific setting, where a lot of agreements are done verbally, without almost any negotiation. Sometimes it is not even an agreement, but just mere curiosity that sparkles a collaboration. Collaboration in science often starts with complementary capacities, for example, two different groups which developed expertise in different techniques. Negotiations come at a much later stage, only when the collaboration was successful and researchers need to agree on how to share the credit.

The other two principles of the Agile methodology, however, are worth discussing in detail in the context of scientific work. To value individuals and interactions is a general principle to which few researchers will oppose. It is better to allow creativity than to be fixated on a specific process or tool. Moreover, great ideas may appear from a discussion or an interaction with another researcher. However, it is also important to value the processes and tools that each researcher has available in the lab. Even though questions can appear in any setting, only places with the proper setups will be able to answer them. The opposite is also valid, posing questions to which one has the proper tools to answer.

The context of every research is different and every field is particular; valuing individuals is part of a healthy working environment, but one should not lose sight of the established processes. Processes can guarantee the quality of data and the moral integrity of researchers. It doesn't mean things cannot be changed for the better, but there should be a limit to how much an individual can be valued over an established process. A brilliant scientist will stop contributing to the group if he or she doesn't document the work or doesn't follow the procedures established to share resources, for example.

Working software over extensive documentation should be better rephrased as Results over extensive documentation. In this particular point, an Agile process can be the opposite of what a lab manager wishes, but it is exactly what commonly happens. Reproducing results is only possible when extensive documentation is available, allowing others to perform the exact same measurement. The pace at which science is moving, however, values quick results over thorough research, allowing little or no time to provide extensive documentation about experiments and results.

Documentation in science has a different value than for software development, but the rhythm at which both evolve is fundamentally the same. Science is based on building on each other's work, while software development can be done to overcome a very specific obstacle. The difference between a scientific work and software development is, therefore, the life cycle of the results. While one expects to build experiments on top of the knowledge generated in a lab, he may not use an available program as a starting point for further development.

The Agile manifesto provides not only the four values stated at the beginning but also twelve principles to follow. Some of which are very relevant to a lab manager. For example:

- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

- Continuous attention to technical excellence and good design enhances agility.

- Simplicity —the art of maximizing the amount of work not done— is essential.

- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

When you are developing software within a scientific environment is very important to be critical about what you are requested to do and how to do it. If you are used to developing software for customers or for small projects, you will notice that what people expect is completely different from what researchers expect. The software that you leave in a lab is going to be used, hopefully, for many years, and future researchers are going to need to implement new options and functionalities into your program. Keep this in mind when you are deciding how to document or if you plan to use a very obfuscated Python method.

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 and with an eye of criticism. When in doubt, check the instructions that the developers of the different packages provide.

2.2 Installing Python

You are going to start by installing Python itself. This book is based on Python version 3.6. Most likely 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.

2.2.1 Installation on Windows

Sadly Windows doesn't come with a pre-installed version of Python, however, 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 to your hard drive. Once it is done, 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 but it is unlikely. Testing your installation

To test that 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 dialog 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. For example, if you want to go one level up you can type cd .. if you want to enter into a folder, you type cd Folder. It is out of the scope of this book to cover all the different possibilities that the Command Prompt offer, 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.

2.2.2 Installation on Linux

Most Linux distributions come with pre-installed Python, so we have to check whether it is already in your system. Open up a terminal (Ubuntu users can do Ctrl+Alt+T). You can then execute python3 and check if it starts. If it works you should see something like this appearing on 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.

2.3 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]
```

Pip will automatically grab the latest version of the package from the repository and will install to your computer. Installing a package is normally equivalent to having a folder with the package name and all of its contents in a special location where Python automatically looks for packages. After installing a package, you can try to find where it is actually located.

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

Note

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

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]
```

. For the listed packages, it is not important the order in which you install them.

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 into a window.

2.3.1 Installing Qt Designer on Windows

To install Qt Designer, you will need to install the Qt5 libraries. Go to the Qt Website and select the Open Source version. You can follow the steps listed on the website in order to download and install Qt5. Alternatively, you can download the offline installers that will make it easier to install the library on computers without internet access. Together with the library, Qt Designer should have also been installed. You can try searching for it within the list of programs, or check the following directory to see if it is available:

```
C:\Qt\5.10\mingw53_64\bin\designer.exe
```

As a complement to Qt Designer, you also have Qt Creator, which can be downloaded from the same page. It is a very complex program that can handle not only the creation of windows but also the coding. It is a bit of an overkill for the purposes of this book, but you should be aware of its existence. Qt Designer is built into Qt Creator, therefore if you install it, you will be able to follow this book without problems.

2.3.2 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 in a terminal.

2.4 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 software 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.

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 are going to use. 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.

2.4.1 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 configure. If there is something failing, try to find help or examples online. There are a lot of great examples at StackOverflow.

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

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

At this point, you have a working installation of the virtual environment that will allow you to isolate your development from your computer. To create a new working 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, 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.

2.4.2 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 pip install virtualenv
sudo 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 in 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 (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.

2.5 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 files such as file-rev1.doc, file-new-rev1.doc, file-rev1-friend.doc. 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 to 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 Uetke'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 running:

```
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 and therefore your code will be public. 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, 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 other, you should consider it. If keeping your files secret, for example, while writing a paper, you should consider Bitbucket. 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 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 it 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 few simple steps. Create a folder where you would like to host 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. Different 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 for 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, and it is simple and powerful. 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 Sublime and Atom. Both of them look very much alike. One of the 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 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 perhaps Pycharm. Pycharm is not free, but it has a community edition. If you have an e-mail from a University, you can install the professional edition for free.

Another very powerful IDE for Python is Microsoft's Visual Studio, which is very similar to Pycharm. 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.

If you are new to Python, the best is to start simple. Both Sublime and Atom are great choices. Once you start building more complex software, then you should check complete IDE's such as Pycharm or Visual Studio. Both of them integrate nicely with Virtual Environments, and therefore they are aware of the packages that you have installed, missing dependencies, etc. They also allow you to run the code without leaving the editor. The downside of IDE's over text editors is that they consume much more resources of your computer.

The choice is yours. Whichever one you make it is going to be more than appropriate for following the course. Remember to practice before starting to 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 while before changing to another.

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 they behave differently, they communicate differently 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, however, a great number of them belong to the message based category. The interaction with all these devices happens through the exchange of text messages, both from the user to the device as the other way around. The common behavior is that a user sends a specific command and the devices answer with specific information. Some devices that belong to this category are oscilloscopes, lasers, function generators, lock-ins, and many more. This book is accompanied by a device that is also message-based, as we shall see later.

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 be communicating with a laser, you can send a command for switching on the output. 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 in 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 `query` a device.

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. All the parameters given by the manufacturer are important in order 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 what would happen if there was no standard ending for a message. Imagine that you are sending a value to a device, for example, you want to set the output wavelength of a laser to 1200nm. 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 than to 1200nm. 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 the 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 devices that comes with the book 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 a very common behavior for smaller or older devices, which provide USB connectivity for convenience. Always check 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, in Windows, it will be something like COM1, in Linux, 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 display the same behavior.

Once you identify which one is the device you want to communicate to, you can start sending some commands to it. Devices normally have a command to identify them, and the PFTL DAQ is no exception. It is a good idea to check that everything is working properly to get the serial number from the device. Please note that in the examples, 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 section by just typing `python` and then the rest of the commands directly into the command line.

```python
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 the 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. This is a way of telling Python how to encode the string before sending it to the device. Devices don't use the same type of strings that Python normally uses, and that is why you have to convert it to binary (hence the `b`) before passing it to the PFTL DAQ. Converting strings to binary or decoding them and understanding when to do it is complicated, but once you grasp it, there are not going to be problems.

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` command is found. You can then print the answer in order to verify that you are actually communicating with the device you wanted. Notice that when you want to embed information into a string, you can use the combination of `{}` and the `.format` command. Finally, you close the communication with the device in order to liberate it in case another process is going to use it.

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))
```

Working with a device is pretty straightforward when the documentation is clear, which unfortunately 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.

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

As you have seen, when communicating with devices, there are a lot of things that one has to take into account, such as the baud rate, the line ending, etc. Moreover, it becomes very unhandy every time we want to send a command to type it. If you still remember the *Onion Principle*, you will notice that now is when it comes to action for the first time. If you completed the last exercise, probably you have written a lot of code, to do the measurement. If you wish to change some 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. At this point, the code starts to be more complicated and long, and because you want to reuse it, you need to start developing into a file, not just in the command line. You can call the file **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 example above shows you how to start a class for communicating with a device. Having a class and not a plain script makes your code much easier to share, to reuse and to maintain. If you don't remember how to work with classes, you can check the appendix Review of Basic Operations. The class defines a method called `__init__` that will be executed every time the class is instantiated, i.e., every time an object is created. This method creates creating 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 in 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 encapsulates the two steps needed to get the serial number: first, a `write` command with the appropriate string and then a `readline`. The value that is recovered from the device is returned to the user.

Once you have written the class, it is time to use it. The easiest is to add some extra code to the end of the **simple_daq.py** file. Later on, you will see how to use classes defined in other files. You can add the following:

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

You begin by instantiating your `Device` class with one argument, the port. This will trigger the method `__init__`, as explained earlier. Notice that even if `__init__` takes two arguments, you only have to explicitly pass one; the other argument, `self` is implicitly added. In this case, the communication with the device is established as soon as the object is created. You can use the method defined, in this case, `idn()` to recover the serial number and print it to the screen. Again, `idn()` takes only one argument, `self`, that is implicitly passed to the function when you do `dev.idn()`. To run the file, just type `python simple_daq.py` and press enter. You should see the serial number appearing in the terminal.

Now you know the basics of writing a class for the device. You can also write methods for reading an analog input or generating an output.

Exercise

Write a method `get_analog_value` which takes two arguments: `self` and `port` and that returns the value read from the specified port.

Exercise

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

3.4.1 General methods to reduce the amount of repetition

As you may have already seen, both `idn` and `get_analog_value` repeat the structure of writing to the device appending the proper line ending, and reading from the device. Those are just simple operations; more complex ones would require several steps of writing and reading from the device and it wouldn't be handy to append every time the line ending and the encoding. It is possible therefore to abstract the procedure into a new one called `query` that takes care of everything. It is useful also to separate the line ending from each command. In that case, we can be sure that if, for example, you get a new device with the same commands but a different line ending we can update just one line of code.

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,
                'write_timeout': 1,
                'read_timeout': 1,
                }
    rsc = None
```

Now you can see that there is a lot of new information in the class. We will discuss later why we do it before the `__init__` method, it is just worth knowing that the properties can be accessed in the same way by using `self.DEFAULTS`. Splitting the instantiation of the class and the initialization of the device is also a common good practice. So we can now rewrite the rest of 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. Notice that now, nothing happens when you instantiate the class, it just stores the `port` as the property `self.port`. When you will want to start communicating 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 do not immediately start the communication with the device. 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. In any case, you should be aware that not having a small delay between opening the serial communication and sending the first commands can give you some hard time tracking down errors.

So far the only difference with the previous code is the `__init__` method. Now it is time to actually improve the rest of the class. You already know that normally 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, the `write` method should take into account the line ending and the encoding.

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

You can see that now the `write` method is more useful. It takes the message, appends the proper termination and encodes it as is specified in the `DEFAULTS`. Then writes the message to the device as 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.

As you probably noticed at this point, when developing code you 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 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, not 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, on which you will be able to build.

```python
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 if the encoding would be different you will start to see errors appearing. For convenience, you 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 being 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 it is 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 `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 CtrlC. 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 of 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()`.

And now we are finally there with what I promised; the only missing part is to condense together both the read and write into a new method called `query`. Since we did all the heavy work in the previous two methods, the `query` is going to be surprisingly simple.

```python
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.

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

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

Notice that the first step is checking 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. Such a method can look like this:

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

The method `set_analog_value` takes two arguments, the `port` number and the `value` to output. You format that information into a string as specified in the manual, 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 the information. 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.

At this point, you are ready to try to do something with the DAQ card other than reading noise. Hook up the LED following the instructions, and check what happens when you change the voltage output from the DAQ. At this point you should be seeing that depending on the voltage you set to output, the LED becomes more or less bright. Moreover, you should observe that for a range of voltages there is no light emitted. What you are now seeing is what you are going to measure, i.e., the I-V curve.

At this point, you are on the verge of doing something great. You have everything that you need to actually measure the current that goes through your diode, you only need to combine the setting of an analog output and the reading of an analog input.

Exercise

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

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.

You already 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 log 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 utilize.

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 is a consequence of sequential changes of an output, and therefore we should split those ideas into two different places. In the beginning, it may result very hard to notice the true reason for this but with time it will become clearer. When you develop the driver for a device, and you include a method for performing a scan even if the device doesn't have this option, several problems can arise in the future.

First, if someone else uses your driver, they will be tempted to add their own logic, perhaps they don't wish to do a linear scan and either alter the method or develop a new one. These improvements are going to become very hard to track and to maintain because they don't follow the manual of the device you are using. The changes can even ruin downstream code that was working fine for you. Secondly, it can happen that one day you change your device for a different one that has a built-in option for doing scans, but the inputs it takes are different from what you have programmed today. You either develop an ad-hoc driver, limiting the functionalities of your device, or you refactor all your code in order to accommodate the new needs.

At this point is when the idea of having an onion plays its role. You should start with some core functionalities. If you are writing a driver for a device, it is clear that those are going to be exactly the ones that are supported by the hardware. When you want to do more complex operations, such as a scan, you are going to provide them with a different layer of your program. Separating the logic from the driver is a common design pattern that is normally called MVC, and is going to be covered in the next chapter. For the time being, don't stress yourself too much and keep in mind that some things of what you have achieved in this chapter are going to change.

3.6 Adding Units to the code

There is one great example of why using the proper units is important: a multi-million satellite fell from its orbit because of a mix up of metric and imperial units. Fortunately, there is a Python package called *Pint*, that allows you to work with units. By using *Pint*, you will be adding a new type of variable called a `Quantity`, which is a combination of a number and a unit. *Pint* has a lot of options that you should explore yourself by reading their documentation. We are going to point out some of the more important ones that can be useful for your developments.

```
>>> import pint

>>> ur = pint.UnitRegistry()
>>> meter = ur('meter')
>>> b = 5*meter
>>> print(b)
5 meter
```

```
>>> c = b.to('inch')
>>> print(c)
196.8503937007874 inch
```

In the example above, you can already grasp the potential of *Pint*. First, you import the package and start the *Unit Registry*. In principle, Pint allows you to work with custom-made units, but the fundamental ones are already included in their *Unit Registry*. Then, because of convenience, we define the variable `meter`, as actually the unit meter. Finally, we assigned the value `5 meters` to a variable `b`. Remember that `b` is a `Quantity`, therefore it is not just a number, but a number and a unit attached to it. *Pint* allows you to convert between units, and this is how we create the variable `c`, which is 5 meters converted to inches. But this is not all, look at the following example:

```
>>> d = c*b
>>> print(d.to('m**2'))
25.0 meter ** 2
>>> print(d.to('in**2'))
38750.07750015501 inch ** 2
>>> t = 2.5*ur('s')
>>> v = c/t
>>> print(v.to('in/s'))
78.74015748031496 inch / second
```

You can see in the example above that *Pint* can handle complex units such as speed, current, etc. You can operate on them as you would with pen and paper, and you can convert from one to another very easily. Converting from one unit to another will become very handy later on when you want to pass a value to your device, and you want to be sure that the value you are passing has the proper units. An important feature that you may also need at some point is to get the number associated with a *Quantity*, without the units:

```
>>> current = 5*ur('A')
>>> res = 10*ur('ohm')
>>> voltage = current*res
>>> print(voltage)
50 ampere * ohm
>>> print(voltage.to('V'))
50.0 volt
>>> print(voltage.m_as('mV'))
50000.0
```

Exercise

In the device driver that you have developed, change the way you return the value of an analog input to include its units. You will have to transform from integers to the real voltage.

Exercise

Change the way you handle the setting of an analog output in order to account for real-world units.

3.7 Introducing Lantz

Defining a class for your device was a very big step in terms of usability; now you can easily share your code with your colleagues and they can immediately start using what you have developed with really few 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 to the values we can pass to different properties in order not to go beyond what is established by the manufacturer. If you followed the exercises of the previous section, you could have seen that careless users set the device to a value higher than the specified working range. Imagine that it is stated that the LED you work with can handle up to 2.5V, setting the analog output to 3V would burn it. Fortunately, there are packages that were written with this approach in mind. We are going to mention only one because it is the project to which we collaborate: Lantz. You can install it by running:

```
pip install lantz
```

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. I strongly suggest you check their documentation and tutorials since they can be very inspiring. Here I 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 I 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. I 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.

```python
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 the `MessageBasedDriver`. If you are unsure of what inheriting means, I suggest you to check How to Work With Classes. Surprisingly, there is no `__init__` method in here, because we are going to construct the object in a different way. 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 established in our driver. The `ASRL` option is for serial devices, this allows you to specify different defaults for the same device but 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 included in the example is `idn` because, even if simple, it already shows some of the most interesting capabilities of Lantz. First, you see that we use `query` instead of `write` and `read`. A query is a command that automatically writes to the device and reads the answer, it also appends the `write_termination`, and reads until the `read_termination`. Exactly what you have done before, defining your own `query` method could have been skipped if you would have used *Lantz*. The idea of forcing you to develop your own `query` method is to understand what is going on and how communication with devices actually work. Once you understand the basics, using complex packages is going to be much easier and profitable.

As you have probably noticed, there is a `@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. Don't be impatient, more on this will come soon. Another advantage is that you can treat the method as a property. 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 result 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 a property called `output0`, with a `None` value. Every time we update the value of the output at port 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, you 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=list(range(0,2)), 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 device has just two channels and it doesn't make sense to use the `range` command, but you can see how it would work if you ever have 64 output channels that you want to control. 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.

3.8 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. Finally, we have covered Lantz, a Python Package that allows you to build drivers in a systematic, clear and easy way.

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 through 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.

We have briefly shown you that you can also put units into work by using `Pint`. We didn't cover all the options that Pint supports, but you should feel free to explore and try them out. Nothing is going to break, that is why the PFTL DAQ device is built with simple pieces easy to replace. Converting from one unit to another is very useful because you are leaving a very well established procedure behind. Even if you forgot to document what you were doing, it will always remain clear what units you were employing.

Remember, documentation is your best friend in the lab. You always have to start by checking the manual of the devices you are using. You have to be careful with their limits. Not only because you can set a voltage outside of the range you want, but because if you employ an instrument outside of the range for which it was designed, you can start generating artifacts. Also, when in doubt, always check the documentation of the packages you are using. PySerial, Pint, 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 their communication channels. The developers of packages are normally very active responding to questions.

3.9 Addendum

At this point, you may be wondering why we have defined the `DEFAULTS` dictionary before the `__init__` method of the class. In a lot of simple applications, you will not notice the difference, because you can address the property just as `self.DEFAULTS`. The advantage of defining the defaults before the instantiation of the class is that the properties belong to the class itself and not to the object. Let's explain it better with a simple example. In the code below, you will find a class that defines two properties, one outside of the `__init__` method and one within it. You will also find a method that prints both properties.

```
class Test:
    prop_a = "This is the first property"

    def __init__(self):
        self.prop_b = "This is the second property"

    def print_properties(self):
        print('Property a: {}'.format(self.prop_a))
        print('Property b: {}'.format(self.prop_b))
```

After defining the `Test` class, you can use it and see what is the difference between `prop_a` and `prop_b`:

```
>>> t = Test()
>>> print(t.prop_a)
```

```
"This is the first property"
>>> print(t.prop_b)
"This is the second property"
>>> t.print_properties()
"Property a: This is the first property"
"Property b: This is the second property"
```

This behavior is exactly the same as always, regardless of where the property was defined. But now note what happens before instantiating the class:

```
>>> print(Test.prop_a)
"This is the first property"
>>> print(Test.prop_b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'T' is not defined
```

Remember that `Test` is a class that was not instantiated. What you see is that `prop_a` exists even before creating an object, while `prop_b` exists only after we run `Test()` and therefore the method `__init__` is executed. Moreover, we can do the following:

```
>>> Test.prop_a = "This is a new property"
>>> t = Test()
>>> t.print_properties()
"Property a: This is a new property"
"Property b: This is the second property"
```

From now on, every new `Test` object will have `prop_a` defined as `"This is a new property"`. For example, imagine you have several devices connected at the same time, and they all have a different line ending. Instead of refactoring your code, you could simply alter the `DEFAULTS` in the class and then instantiate it for every device. Python is a very flexible language that allows you to do almost everything that you imagine with objects. Sometimes it can get you a while to understand when to use one or the other approach. If in doubt, always start with the simplest and the one you understand better.

Chapter 4

Layout of the Program

4.1 Objectives

In this chapter, you are going to learn how to lay out your program following a pattern that will make your code easier to understand, to maintain and to share. Normally, you should have defined the structure of the program before you start writing code. However, development is not always linear and projects that begin small end up controlling very complex experiments. Sometimes you can already know how the future looks, what experiments you may want to perform in a year from now or even more. In any case, building on top of a solid foundation allows you to be future proof.

In this chapter, You will learn about some common design patterns and general ideas that are used in the long run. What you are going to learn in this chapter is a collection of best practices, which doesn't mean you have to follow them to the letter, but they are a very good starting point. You will learn about the MVC design pattern, that will allow you to separate different elements of your code into reusable containers. Moreover, you will create a specific folder structure that will enable to use code shared by others. The MVC design pattern forms a layered structure, exactly in sync with the ideas that we have exposed when discussing the *onion principle* for development.

4.2 Introduction

Computer programs are meant to be a solution to a more or less complex problem. It seems unlikely today that anybody would invert a matrix by hand or with a calculator. Nor to imagine that somebody would plot a complicated function with pen and paper. Several packages exist to solve those needs, from Python to Matlab to Origin. In the same fashion, the complexity of experiments is such that it is impossible to control all the needed variables by hand. Even in simpler experiments, there will always be a computer responsible for acquiring data, saving it, and many other functions.

Developing a computer program for controlling an experiment is not a trivial task. One has to consider the limits of the devices, acquisition rates, conditions for closing a shutter or switching off the power. On top of all that, one has to write programs that most likely are going to be used by others, and that can be developed further on by another person. It doesn't make sense to write software today that will be obsolete next week just because few parameters of the experiment changed.

To address these concerns, there are few techniques that a developer can follow in order to achieve a high degree of productivity and, at the same time, producing software that can be enhanced later on. The idea of establishing best practices is not increasing the burden of developing software, but to streamline the process. The cornerstone of every great program is to have a design pattern that can be followed by all the developers. Therefore, when developing software for controlling a setup we have to keep an eye on different considerations:

- What you develop should be readable by current and future colleagues
- It should be easy to add solutions developed by others
- The program should allow exchanging devices that achieve the same goal (i.e. oscilloscopes of different brands, etc.)
- The code developed in one context has to be available in other contexts (i.e. in other experiments)

As you can see from the list above, it is very desirable to be able to reuse code, not only within the same lab with the same colleagues but also to use code that was developed in other institutions and that your code can be reused. Reusing code not only saves a very long time, it also empowers collaboration between different labs, possibly enabling researchers to gain weight when asking for specific features from manufacturers. Reusing code needs that some standards are specified, in such a way that another person can easily include the code in their projects. This is exactly what you did in the previous chapter when defined a class for controlling the device and not a plain script.

For a set of rules, it is crucial that they don't hinder the developer's capabilities to reach solutions in a short time. For example, if because of following common standards, the time it takes to find a solution is multiplied by a factor of two or more, developers will end up bypassing those rules. The design principles that we are going to expose in this chapter are broad enough as to do not impose a lot of work to the developer but useful enough that the code structure will be clear even for someone not familiarized with the specifics of the program.

4.3 The MVC design pattern

A design pattern is nothing more than a set of rules that determine where different parts of the code are going to be placed and how are they going to interact with each other. Different design patterns can be used in different contexts and with different goals. If you have ever encountered a manual on how to develop web applications, you probably also found entire chapters related to the design patterns that developers follow. The same principles can be applied to any software development, and particularly for software to control lab equipment.

Generally speaking, the Model-View-Controller pattern (or MVC) is a set of rules on how to structure your code in order to make it reusable and easy to maintain. The MVC framework, although originally developed for desktop computing, has become popular in other fields, for example for designing web applications. Web and desktop applications have specific definitions for the model, the view, and the controller. Lab applications, therefore, have to update those definitions in order to consider the different elements that make up an experiment.

An experiment will have devices generating data on one side and the user supervising the experiment on the other. The software sits in between the two ends, it will communicate to the devices the user commands, and it will output back data for the user to interpret. When we have to develop software for an experiment, we are going to start by building drivers for the devices. We are going to define the steps that make up an experiment, we are going to allow the user to change some parameters, and we will show the data back to the user. If we lay out our code in a smart way, we will be able to utilize the same code in different projects and, more importantly, exchanging devices or changing the steps of an experiment will be simple. Let's see what is the role of each of the elements in the MVC pattern for the lab.

A *Controller* is the real driver, responsible for communicating with the devices. It can be a Python class you developed yourself or a Python package that you have installed in a virtual environment (such would be the case of PyDAQmx, Lantz, etc.) The driver has to be general and has to reflect the capabilities of the device, nothing more. For example, if a device is able to acquire just a data point at a time, the driver shouldn't include a function for acquiring an array of data using a loop. This should be very similar to the discussion we gave in the previous chapter regarding the class that was developed for controlling the device.

The *Model* is where all the logic is defined. In the models, you are going to define how you are going to use a device for your own experiment. Generally speaking, models define how the user interacts with the available data. In an experiment, this can be translated to how the user interacts with a device. For example, if you want to acquire a time trace and you have a device that can acquire only one point at a time, you should make a loop in which several points are acquired. If, on the other hand, the device is able to acquire time traces and you are interested in the Fourier transform of the data, you can gather the information from the device and transform it as you desire. The main advantage of splitting *Controllers* and *Models* is that it becomes simple to upgrade or replace a device. You will need to update the *Model* in order to reflect the new options of the device, but the logic of the experiment is left intact.

The *View* is, technically, the place where you can locate everything related to how you show data to the user, and how the user can change the parameters of the experiment. In practice, it is the collection of files that build up a Graphical User Interface (GUI). Within the GUI you will set, for example, the length and delay of the time trace you want to acquire and in turn, you will use the model to acquire the data. You can plot the results back to the user and save them to disk, etc. It is important to note that, in this case, the user interacts through the view with the model and never directly with the controller of a device.

Warning

If you are new to developing code for the lab, it may seem that splitting *Controller* and *Model* is a waste of time. When you have only one device that you use for only one goal, it may very well be the case. However, when you wish to include code developed by others or when you want to share your code, it is crucial that you split the capabilities of your device from the logic of your experiment. If you don't do so, all your code is going to work only when repeating the same experiment.

You have to remember that the meaning of *Model*, *View* and *Controller* changes depending on each developer or community. It is important to note that people developing a web application are not dealing with devices in the real world as developers in the lab do. Therefore, how the MVC pattern is used can change from one field to another. What is important is that if you stick to some decisions while you are developing, you will find that your code is much clearer and easier to maintain, especially when someone asks for help on a program you have developed months or years before.

You also have to remember that it may happen that you find a design pattern or strategy that better suits your needs. That is perfect, the only thing you have to remember is that you have to make it clear to others, and you have to stick to it when coding. If you switch halfway, bugs are going to be very hard to solve, and sharing code would become virtually impossible. The extra time it takes to write few more lines and keep everything tidy is completely worth the effort in the short-term (around 3 months), and is going to be crucial in the mid to long-term.

Using the MVC pattern with a fixed structure, i.e., with three folders called Model, View, and Controller is a very good starting point. It gives the user a clear idea of where it may be the file that they need to change, or how the bug was generated. Remember, if you add a **README.md** file to every folder, it will be nicely rendered when exploring the code on Github. You can use that file to give a hint to the users on what is going on and where to edit the things that may be important.

4.4 Structure of The Program

So far the discussion has been general and is applicable to any project in any language. Nut now it is time to specify how to implement the MVC pattern into a Python project. Remember that the layout we are proposing here is based on our own experience when developing larger projects. It is a good starting point, even if it looks more complicated than what it could be if you put everything into the same folder or into the same file. Moreover, everything you learn in this chapter will help you understand how other packages are structured.

In the program you are developing in this book, you will follow the MVC design pattern quite literally. This means that you have to create three folders called *Model*, *View* and *Controller*. As explained before, controllers are going to host the drivers for your devices. You already developed the driver for your device in the previous chapter, so it is appropriate to place it there. Models are going to be responsible to make explicit how to use those devices. At the end of the last chapter, you already saw that you were including some logic into the device, when you started to develop the methods to linearly vary the output voltage, for example.

Exercise

Create a file in the Models folder called **analog_daq.py**. Inside, start a class called `AnalogDaq` and move the methods that you think appropriate from the Controller into the Model.

You can also define another, very important model, called *Experiment*. The Experiment model will have all the logic to perform a measurement. You have to notice the difference between the logic when using a device and the logic when performing an experiment. For example, the model for the experiment can include steps such as open a shutter before starting a measurement or can deal with how do you save the data. It is also responsible for keeping track of the parameters you need to perform an experiment.

Exercise

Create a file in the Models folder, called **experiment.py**. Define a class called Experiment() and some methods that you think are going to be useful. You can, for example, add a method for switching on or off the LED. You can also add a method for doing a scan. The methods can be empty. What you have to start thinking about are the parameters that you need and the order in which every method can be called.

The View folder is going to require a bit more of work than the other two. However, you can already start thinking about how the user is going to interact with your program. Most likely, you have already thought that sometimes the LED will be plugged into the output port number 1, sometimes to port number 0. You don't really want to alter the code every time. The same happens, for example, with the port you want to monitor, the time delay between steps, etc. It is convenient, therefore, to define these parameters in a text file that is easy to read and to modify.

The files that the user is going to supply need to be read and transformed into parameters that can be used by the experiment model. This behavior is exactly what you could expect from a View. It is the nex between the user and the rest of the program. We are going to use text files in a format called YAML, which is very easy to generate and to read. For the time being, don't worry about it, we will cover it later.

Exercise

Create a file in the View folder, called **load_files.py**. Create a function that takes as an argument the path to a file. Think about all the parameters that a user needs to supply in order to perform an experiment and write them down within the file itself.

Remember that a program interacts with a user in different ways. By the end of the book, you will have developed some nice graphical user interfaces, but you can also live without them. A program can output information to the command line, and the user can take actions based on what is being displayed. Building a GUI is a complex topic, and we don't want to deviate the attention from what is really important here.

Once you have the three folders (Model, View, and Controller) created, and you have created some files in each one of them, it is time to take a look at how the importing process works with Python.

4.5 Importing modules in Python

Probably you have already seen some lines that look like this:

```
>>> import numpy as np
>>> from time import sleep
```

The first one is importing the numpy package, but changing its name to `np`. Changing the name makes it easier to work with because you need to type only two letters, `np` instead of `numpy`. The second line is importing one specific function from a package called `time`. It doesn't really matter what do they do, but to realize that the import process was different in both cases. *Numpy* is a complex package, with a lot of modules that can be used. The same goes for *time*. However, in the lines above you have imported only one the module *sleep* from package *time*. If you want to use it, you can simply do:

```
>>> sleep(1)
```

While for using *Numpy*, you will need to specify which module you want:

```
>>> np.random.random(1)
```

When working with your own code, you can import different modules in the same way. Open a terminal and navigate to the root folder of the project, where you have the Model, View, and Controller. If you start Python, you can type into the interpreter:

```
>>> import Controller
```

Now you have the controller available to use. For example, you can do the following:

```
>>> dev = Controller.simple_daq.SimpleDaq('COM1')
```

And you can use the device as you have been using it in the previous chapter. Importing the *Controller* may not be exactly what you want, because sometimes there are many devices and you need only one. You can specify which module to import, exactly as you have done with `sleep`. To import only the SimpleDaq class, you can do:

```
>>> from Controller import simple_daq
>>> dev = simple_daq.SimpleDaq('COM1')
```

There are two things important to notice. First, if you change the code of the *SimpleDaq* class and import it again, you won't see those changes reflected. You should exit from Python and start again. The second thing is that, when you import the controller, you may also start communicating with the device. This is happening because at the end of the **simple_daq.py** file, you have added some lines for using the class. Those lines also get imported and executed. When you import modules, and when you design modules, it is very important being in control of what is going to be executed.

To avoid starting the communication with the device when you do `import simple daq`, you have to add one extra line of code at the end of the file. It will look like this:

```
if __name__ == "__main__":
    d = SimpleDaq('/dev/ttyACM0')
    print(d.query('IDN'))
    d.write('OUT:CH0:4000')
    input('Press to read value')
    print(d.query('IN:CH0'))
    d.finalize()
```

Next time you `import simple_daq`, the code that follows the if statement will not be executed. However, if you run the file by itself by typing `python simple_daq.py`, it will. This is very useful because it allows you to distinguish the two cases, when the file is directly executed and when the file is imported. In many cases, you can use the bottom of the file to show how the class is used or to perform some quick tests. In the code above you can quickly see how to start the communication, query for the serial number, change the analog output, etc.

To understand a bit more about how the `if __name__` works and have a clearer picture of the importing procedure in Python, create a new file called **dummy_controller.py** in the *Controller* folder, and paste the following lines of code:

```python
print('This is the dummy Controller')

def dummy_example():
    print('This is a function in the dummy Controller')

if __name__ == '__main__':
    print('This is printed only from __main__')
```

From the terminal, enter into the Controller folder and type

```
python dummy_controller.py
```

The output that you see should be:

```
'This is the dummy Controller'
'This is printed only from __main__'
```

What you see is that the entire code got executed but not the function. `dummy_example` is only defined, but never executed.

Exercise

What do you expect it to happen if you import `dummy_controller`?

Things are going to be different when you import the file. Open the Python interpreter and type the following:

```
>>> import dummy_controller
"This is the dummy Controller"
>>> dummy_controller.dummy_example()
"This is a function in the dummy Controller"
>>> from dummy_controller import dummy_example
>>> dummy_example()
"This is a function in the dummy Controller"
```

The first thing you should notice is that what is written at the end never gets executed, meaning that the `if` statement is not `True`. This is very useful when we want to have code that works standalone (when we execute it directly, for example) but we don't want to execute if we import it. In the case of the real controller, we wanted to leave some examples at the end to show how it can be used, but when you are importing a class, you don't really want it to start communicating, you just want to have the class available.

The other thing that you should have noticed is that depending on how you import from a file, the first print statement is executed or not. If you type `import dummy_controller`, you will see that the first print statement is there, while if you type

```
from dummy_controller import dummy_example
```

nothing happens. This is very useful because it means that in the first case you are really executing everything within the file, up to the `if` statement. In the latter case, however, you are only getting the specific function you want.

Finally, it doesn't really matter how you imported the dummy controller, the `dummy_example` function will always be available. It can happen that you either have to type

```
dummy_controller.dummy_example()
```

or simply

```
dummy_example()
```

but you will always see the same output. If you were sitting in an outer directory, you would have to do

```
Controller.dummy_controller.dummy_example()
```

and it would still work. When working with modules, there is plenty of flexibility on what to do. Bear in mind, however, that the same flexibility comes accompanied by some usage patterns that may not be clear.

Imagine you add a second function to your `dummy_controller.py` file, you can import both at the same time by typing

```
from dummy_controller import dummy_example, second_function
```

You could also do

```
from dummy_controller import *
```

The second option, however, is highly discouraged. There are several disadvantages to doing so. First, if you are reading the code and you see something called `second_function` you have no idea where it came from. Also, if the `dummy_controller` creates variables, they will appear in your own program, perhaps overriding something you wanted to preserve.

Note

We should establish some naming conventions in order to avoid confusion later on. In Python, any file that defines variables, functions, classes, etc. is called a `Module`. The folder that contains modules is called a `Package`.

Working with the imports in Python is sometimes easier than understanding them, especially when trying to pay attention to all the different definitions. The Python Documentation has a great chapter covering a lot of the ideas here discussed. Many of the properties and behaviors can be learned by trial and error, though it can be very time consuming and it may lead to unexpected errors.

Perhaps you noticed by now that the way your package is imported and the way *Numpy* is imported is very different. *Numpy* can be imported regardless of where you are, but your package can only be imported if you are in its own directory. When you perform an *import*, Python searches for modules in specific locations, and once it finds one, it stops searching. *Numpy* is located in one of such folders, but your package is not. One of the ways in which you can let Python know where your package is located is by adding the folder of your package to a system variable called *PYTHONPATH*.

If you are on Windows, you can follow the steps of Chapter 02, when you were dealing with the details of adding environment variables when installing Python. If you are on Linux, it is enough to run the following command:

```
export PYTHONPATH=$PYTHONPATH":/path/to/PFTL"
```

After doing that, you will be able to type

```
from Controller import *
```

wherever you are in your computer and Python will find the appropriate folder. In Linux, the change is not permanent, you will need to run the line again next time you start the Terminal. There are ways of modifying the environment variables in a permanent way, but I leave it to you to find out. There is plenty of information online.

There is something very important to add to our packages, the **__init__.py** file. Sometimes it is important to let Python know that a specific folder is a package to prevent directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later, on the module search path. The **__init__.py** file can be an empty file, but it can also be used in smart ways. Normally the **__init__.py** file can be used for some initialization. You can create therefore an **__init__.py** file within the *Controller* directory and place the following:

```
from .simple_daq import SimpleDaq

Controller_variable = "var"

print("This is the init of the controller")
```

Go back to the root folder, one level up from the Controller folder and start Python.

```
>>> import Controller
"This is the init of the controller"
```

It is super simple and clean. If you want to use the *SimpleDaq* class, you can just type `Controller.SimpleDaq`. Moreover, you should have seen the print statement appearing on your screen. This means that the `Controller_variable` is also available, you can see it by typing:

```
>>> print(Controller.Controller_variable)
"var"
```

You could also import only the Controller class or the variable. You should just do:

```
>>> from Controller import Controller_variable
>>> from Controller import SimpleDaq
```

How many times do you see the print happening? Interesting, isn't it? Python takes care of executing the code only once, the first time it encounters an import statement. We will look again at the **__init__.py** file later on. For the time being it is important that you realize that sometimes things happen within that file. A lot of code can be executed within the **__init__.py** file and therefore you should also look into them when there is something mysterious that you cannot understand or when you don't find the file from which an import is happening.

Exercise

Read the Documentation and find a way in which the code `from Controller import *` imports only the `SimpleDaq` class and not the variable we introduced in the `__init__.py` file.

4.6 The Final Layout

Now you have a clear separation of your code into Model, Controller, and View. However, these are not the only folders that you are going to use. Most likely you will want to provide some examples of how to use your code, or the documentation for your package. We have also discussed using text files for the input data, and those files should be located somewhere, for example within a *Config* folder.

If you create extra folders next to the three main ones, the structure of the program will start to be polluted. It won't be clear what is part of the package, what is a user-specific setting, etc. Therefore you need to create a folder for the main part of the code and next to it the extra folders that you need. Your folder structure will look like this:

```
Docs/
Examples/
    Config/
PythonForTheLab/
     Controller/
     Model/
         daq/
         experiment/
     View/
          GUI/
```

You can see that there are three folders at the top level, *Docs*, *Examples* and *PythonForTheLab*. The last one is holding the *Model*, *View* and *Controller*. You can also see that in the *Model* we have created two separate folders, one for dealing with the daq and one for dealing with the experiment. If you followed the steps to add the folder to your Python path variable, it means that now you can do the following, for example:

```
>>> from PythonForTheLab import Controller
```

The only missing folder is the one in which we are going to store the data that we want to pass to the experiment. In this case, you should consider different scenarios. At this stage, you are the only one working and using the code, but later on, maybe there are more users. Or, you are the only developer, but there is more than one user of the same setup. In any case, when you configure your experiment, you don't want people to change it, but they should be able to do it for themselves.

In order to be flexible, the best idea is to provide a folder with Examples of how to configure your program, or how to generate the configuration files. These are just examples, aimed at exposing all the configuration options. However, when you want to run the experiment, you will do it starting with a different set of files, stored in a different location.

Let's start simple. In the *Config* folder that lays within the *Examples* folder, you have to create a file called **experiment.yml** to hold all the parameters of the experiment. Earlier we have decided that the format of the file was YAML. If you are not familiar with the format, it looks like this:

```
Experiment:
  name: This is a test Experiment
  range: [1, 10, 0.1]
  list:
   - first Element
   - second Element
```

This file is plain text but with a very special structure. YAML is very simple to read and has just a few rules, the most important one is that the indentation is done with 2 spaces. The choice for YAML instead of XML or JSON is that it requires much fewer typing and is easier to write, while still keeps all the functionality you need. Of course, this choice is personal, and people more inclined to other types of files can adapt the code to their own needs. Once the file is created, you need to read it with python. You can either type the following commands or write them to a file.

```
import yaml

with open('Config/experiment.yml', 'r') as f:
    e = yaml.load(f)

print(e['Experiment'])
for k in e['Experiment']:
    print(k)
    print(e['Experiment'][k])
    print(10*'-')
```

We will talk a lot more about yaml files later on. But for the time being it is important that you know how to start working with them. The first thing is to import the module that allows reading the files, called PyYAML, but that is imported with the line `import yaml`. You first open the file that is going to be interpreted by `yaml`. The `with` statement is very handy when working with files. It basically allows you to forget about closing the file, even if something goes wrong while reading. The only thing to remember is that you call the file `f` only within the `with` statement. Transform a file into a Python variable, you can use `yaml.load()`, which will interpret the file as a dictionary.

As you may remember, the elements of dictionaries are addressed with keys. In this case, there is a main key called `'Experiment'`, and the sub-keys `'name'`, `'range'` and `'list'`. If you want to use one of those elements, you can type `e['Experiment']['name']`, for example. The code above just prints out what each of the elements, separated by a horizontal line. Note that yaml imported the file directly as a dictionary, but some of the elements are particular, they are not all strings such as the `'name'` is.

Exercise

What is the type of variables that yaml generated for the range and the list?

In the same way that you can read a YAML file, you can also write it. For this, you use the `dump` method. If you define a dictionary, you can write to a file very easily:

```python
d = {'Experiment': {
    'name': 'Name of experiment',
    'range': [1, 10],
    'list': (1, 2, 3),}
}

with open('Config/new_experiment.yml', 'w') as f:
    f.write(yaml.dump(d, default_flow_style=False))
```

The code above follows the same logic than for reading, but keep in mind that you are opening the file with the `'w'` option. This means that every time you run the code, you will overwrite the file and lose the previous contents. Open the file that you have just created with a text editor and you will see that it is very similar to the one you created yourself. There may be some small differences, but you can still understand what is written and you can modify it if you want to.

Exercise

Create a numpy array and store it using yaml. How does it look like in the file? What happens if you read it back?

Earlier you were asked to type down all the parameters that you needed for performing an experiment. You can transform that information into a YAML file within the *Config* folder.

Exercise

Create an experiment.yml file in the *Config* folder hosting all the information that you need to perform an experiment.

And of course, since you have learned how to read and write the files, you can update your *View*.

Exercise

Update the function that you created in the *View* folder. Read the yaml file and check that all the parameters are there. Raise an exception if a parameter is missing or is of the wrong type. Return a dictionary.

4.7 Conclusions

In this chapter, you haven't done much coding but we have discussed some general ideas that should accompany you through every software development that you undertake. How to lay out the code is very important, because it is going to give you the structure you need to maintain your program without too much effort. Moreover, since the paradigm that you are using is common to developers in other fields, you can benefit from tools that were not designed specifically for experimental work.

When developing instrumentation software, you will always have to answer the question of who is going to be the user of your program and who is going to build on your development. Most likely the first user and developer are going to be yourself, but this can quickly change. Being able to answer questions from a user with a different perspective and with a different level of programming skills is fundamental for your program to succeed in the mid-term.

The strategies proposed in this chapter do not come naturally to every developer, and even if you know them, you will try to find a shortcut to developing only scripts, to put everything in the same file and forget about it. The truth is that there is no experiment that is performed once. Key to better science is reproducibility, and the clearer the code that allowed you to perform an experiment, the easier it is going to be to perform it again, even by someone else.

Chapter 5

Writing the first Model for a Device

5.1 Objectives

In this chapter, you are going to develop a model for a real device. You are going to start seeing why the Model-View-Controller pattern is so useful. Models are specific ways in which the user can interact with the devices or with other data sources. In the models is where you are going to start developing the logic of your experiment, where you are going to establish the safeguards to avoid things going wrong.

Even if hard to believe, you should spend the most of your time working and improving the models. The controllers can be downloaded or quickly developed, the view is more often than not a simple script to read a file, or you can use existing libraries. The Model, however, is where your true experiment is defined. Is where you have to be very focused and decide in which order the steps are taken and what things can be given for granted or not.

In this chapter you are going to develop the core of the program, and hopefully, the basis for your future program when you are working in your own lab.

5.2 Introduction

When developing models, you have to keep in mind that you will start specifying how the devices are going to behave. This means that models are somewhat specific to your experiment and sometimes the names you give to the functions can reflect a specific task, but it doesn't mean that it cannot be used in a different context. For example, a confocal microscope normally needs a couple of analog outputs to perform a scan, and at least one input to record a signal. The analog outputs can be plugged into a piezo device that moves either the sample or the beam. If you do this systematically, you can construct an image pixel by pixel. You move to a location, record the input, move to a new location, etc. If you want to study the current that flows through a diode as a function of the voltage applied, you could use the same code developed for the confocal microscope without changing a single line. However, using a function called `do_confocal_scan` to measure a current through a diode may be misleading.

Developing a model for the first time is a bit tricky because it implies that you know what is going to happen in the future. In a normal situation, you will have a lot of back and forths with the models, but for the purposes of this book, it is important that you follow the recommendations carefully. For developing a model, you need to understand how your experiment works and what do you want to achieve. The goal of this book is to measure the current that flows through a diode. But if you are building a confocal microscope, for example, you should really understand which analog outputs you are using, how your piezo is connected, what are the limits of the device, etc.

It is impossible to overstress how important it is for you to understand your own experiment before starting to code. Remember that what you are developing in this book is the simplest of the examples. Nothing can really be broken, no expensive device is going to receive an input voltage higher than what it can handle. What you probably have in your setup is much more complex. Maybe you have to use external triggers, maybe you have to push some device to the edge of its capabilities and you should be aware of what the limits are. There is no better friend in the lab than the manuals and the technical support.

5.3 Base Model

A model in Python is a class with some special elements. The first step when building a model is to think what methods we are going to need, and if possible what is the output that each method is going to produce. A method may produce an array of values, a single value, no value, etc. Remember that the idea of having models, amongst other things, is that if in the future you change the device, you only need to write a new model with the same functions and the rest of your code will work without further problems. Since we are writing a model for a DAQ (an acquisition card), we should create an appropriate package in the *Model* folder. Create a folder called *daq* and remember to add an empty `__init__.py` file in each new folder you make. Perhaps you already did this in the previous chapter.

Instead of starting with the model for our DAQ, you can instead create a base class, in which you will keep track of all the functions that a proper model should have. You can create a file called **base.py** and define a class called `DAQBase`:

```python
class DAQBase(object):
    def __init__(self):
        pass

    def idn(self):
        pass

    def get_analog_value(self, channel):
        pass

    def set_analog_value(self, channel, value):
        pass

    def get_digital_value(self, channel):
        pass

    def set_digital_value(self, channel, value):
        pass
```

This class doesn't do anything, it only defines the methods that you need to develop for each device. The `pass` takes care of functions in which nothing happens and nothing is returned either. Having the base class defined allows you to do several things. One is to keep track of all the methods that you need to define when you develop a new model, as we are going to do later. The other is that you can inherit this class, and be sure that all the functions are already present, even if they don't do anything.

Each method is quite self-explanatory. The `idn` refers just to identify the device. You also see that setting and getting values require a port, and a value. You can also start to notice that for this simple example, the model is only a relay for the Controller class. For the time being, there is no method in the base class that doesn't exist in the controller itself.

5.4 Device Model

Once you have established what are the base methods that your model needs to define, you can start developing a model for the real device. There are two important modules that you will need to import, the base model and the controller for your device. Create a file called **analog_daq.py** and add the following:

```python
from ...Controller import SimpleDaq
from .base import DAQBase

class AnalogDaq(DAQBase):
    def __init__(self, port):
        super().__init__()
        self.port = port
```

First, you have to notice that you are using relative imports, i.e., the three `...` before `Controller`. This means that Python will look three directories up for a module called *Controller*. One dot is for the current directory, two brings you to the *Model* and three to the root of the package. When you are not sure whether the package you are developing is in the *path* or not, using relative imports is a way of ensuring that you can still find what you need to import. If the package is in the path, you could change the first line to:

```python
from PythonForTheLab.Controller import SimpleDaq
```

Going back to the code, the `AnalogDaq` class inherits the `DAQBase` class, meaning that all the methods and properties defined in it are going to be available also in the new class. When you inherit a class, you need to be sure that the `__init__` method of the parent is executed. This is what the line `super().__init__()` is doing. We request a port from the user, that for the time being is just stored in a property called `port`. You can use the class, and you will see that the methods defined in `DAQBase` are available also in `AnalogDaq`, even if they don't do anything:

```python
>>> from analog_daq import AnalogDaq
>>> dev = AnalogDaq('COM1')
>>> dev.idn()
```

To make the class actually work, you need to start overwriting the methods defined in the base class. This just means defining a new method with the same name. You can start with the `initialize` to open the communication with the device.

```python
class AnalogDaq(DAQBase):
    def __init__(self, port):
        super().__init__()
        self.port = port
        self._driver = SimpleDaq(self.port)

    def initialize(self):
        self._driver.initialize()
```

You can see that in the `__init__` method, you are instantiating the SimpleDaq with the specified port. You store the object in a property called `_driver`. The driver is going to handle all the communication with the device. Remember that the MVC pattern imposes that the user will never communicate with the device directly, but always through the model. If there is something that the device can do but the model can't, you should expand your model before bypassing it.

Python doesn't have a way of declaring private properties as other languages do. Private properties are those properties that can be used only within an object, but not from outside. After you instantiate the class, you can get access to the driver just by typing `dev._driver`. The fact that the property starts with an underscore is a sign that it should be left alone and not used from outside, but there is no easy way to enforce this in Python. The first one that has to be consistent with its use is yourself.

The `initialize` method uses the driver to initialize the communication with the device, using the specified port. Probably you have realized it already, the `DAQBase` class doesn't have an `initialize` method defined. Don't worry about it now, but you will see why you should have added it also there. Now it is time to improve the class, for example by defining a `get_analog_value` method.

```python
def get_analog_value(self, port):
    value = self._driver.get_analog_value(port)
    return value
```

Having a *model* as a relay for methods defined in the *controller* doesn't look too useful. But now is when things start to get interesting. The controller returns arguments that are simply integers, but that you can translate to a voltage, as specified in the device manual. As you can imagine, improving the rest of the class is up to you.

Exercise

Complete the `get_analog_value` by setting the proper units to the returned value. Don't forget to use Pint.

Exercise

Overwrite the method `set_analog_value` from the base class to actually communicate with the device. Use units! Think about the limits, and what happens if a user sends a plain number instead of a *Quantity*

After you have defined the methods, you can test that everything is working. You can write a simple script that initializes the communication with the device, sends some outputs and reads some inputs. Create a file called **test_daq.py** and write something like the code below:

```
import numpy as np
import pint

from analog_daq import AnalogDaq

ur = pint.UnitRegistry()
V = ur('V')

daq = AnalogDaq('/dev/ttyACM0') # <-- Remember to change the port
daq.initialize()
print('The DAQ serial number is {}'.format(daq.idn()))

# 20 Values with units in a numpy array
volt_range = np.linspace(0, 3, 20) * V
current = [] # Empty list to store the values

for volt in volt_range:
    daq.set_analog_value(0, volt)
    current.append(daq.get_analog_value(0))

print(current)
```

At this stage, you should be able to go through the code yourself. It is not very handy because if you need to change which port you are using you need to go through the code, or if the device is connected to a different port. However, you can see that it is very simple to perform a measurement. If you would be working with a confocal microscope, you could have done basically the same. Probably you have noticed that when you try to get the serial number of your device nothing happens. This is because you didn't define an `idn()` method in your class, but it is important to point out that there were no errors because the method exists in the class, inherited from the base.

Exercise

Fix the `idn()` method to display the correct information.

5.5 Dummy DAQ

You have just seen how to develop a model for a device such as the PFTL DAQ. However, when you develop software for the lab, sometimes you would like to test your program without having the real device connected to the computer. It can be a safety precaution or simply because you are not working on the lab computer. To achieve this, you can define a fake model that generates data without interacting with a device. You can call these devices whatever you like, in this case, we use **dummy**. The code will look like this:

```
from .base import DAQBase

class DummyDaq(DAQBase):
    serial_number = '1234ABC'

    def __init__(self):
        super().__init__()
```

You start by importing the base class `DAQBase`, as you did for the real *model*, but you don't need the controller since you are not going to communicate with any device. The class defines a fake serial, just to be able to return it as if it was the real device. Now you have to think what do you actually expect from a fake device. Since you are developing a dummy DAQ, you don't need to do anything when setting a value to an output, but you do need to get a value when you read back. And you need to use the proper units as well. It can look something like this:

```
import numpy as np
import pint

from .base import DAQBase
```

```python
ur = pint.UnitRegistry()
V = ur('V')

class DummyDaq(DAQBase):
    serial_number = '1234ABC'

    def __init__(self):
        super().__init__()

    def idn(self):
        return self.serial_number

    def get_analog_value(self, port):
        return np.random.random(1)*V
```

The `idn` is quite clear, it just returns the serial number that was defined during initialization. The `get_analog_value` accepts a port as a parameter, even if they don't use it, but you need to keep the compatibility with what was already defined. The value that is returned is a random number with volts as units. Remember that the random generator outputs values only between 0 and 1. You can try to run the test, but using this DummyClass instead. Open the file **test_daq.py**, and replace the line:

```python
# from analog_daq import AnalogDaq
from dummy_daq import DummyDaq as AnalogDaq
```

And run **test_daq.py**. You will see that an error appears because you are using the `initialize` method in the test, but it is not defined within the class. This happened because you included `initialize` in the `AnalogDaq` but you never added it to the DAQBase. Now you see the power that being organized and systematic has. Keeping track of the methods in a general, base class, allows you not to make the same mistakes. Moreover, it would have allowed you to run the test without issues.

Exercise

Add an `initialize` method into the base class.

If you run the test after completing the exercise, you will see that there are no errors appearing, even if you don't modify the dummy class. Now you see that you can exchange the models in a relatively simple way, and the script that is responsible for measuring works without problems. Of course, this is a simple example, where you exchange a real device for a dummy one, but if you would have two different devices, exchanging them would have been just as simple and the measurement would have been performed.

Exercise

Make the `get_analog_value` return the value of a sine function and not just a random value.

5.6 Conclusions

This chapter is focused on showing you what does *Model* mean in the MVC pattern when developing software for the lab. Of course, the examples proposed in this book are very simple and sometimes the gain is not going to be apparent. However, as soon as you start building more complex applications, you will see the benefits of splitting the Model from the Controller, and later on, the View from the Model.

You have built three classes, a base class that helps you keep track of the methods and properties that each device is going to have. Keeping this class up to date and clean is very useful because it allows you and other developers to have a clear starting point. You have seen that by skipping some steps you run into errors, that at this moment are easy to debug and solve, but later they may become more obscure, depending on what triggered them.

In this chapter, you have also started to use more complex options of classes in Python. Hopefully, you can understand what is going on and learn from the examples. Once you see the potential that defining classes have, you will never stop thinking about them. This has also been a chapter where you had to code a lot yourself, making your own decisions. If you ever get lost or think that you are departing too much from what we are asking, you can always check the GitHub repository of the book, where you can find the code for every chapter.

Chapter 6

Writing The Experiment Model

6.1 Chapter

In the previous chapter, you have already seen how to define a model for a device and for a dummy device. Models specify the specific ways in which a user can interact with a device and how data is presented back. For instance, you have specified that you wanted to send and receive the information in volts instead of plain integers.

However, with the model of the devices alone you cannot perform a measurement. In this chapter, you are going to see how to define a model for the entire experiment and not only for a device. In this model, you will include all the steps that build a measurement, including checks and saving data. The experiment model is going to be your go-to place to understand why a measurement doesn't make sense, and also the place where all the information is stored in order for you to reproduce a measurement.

6.2 Introduction

In the **MVC** design pattern for the lab, the *Model* is where all the logical steps and design decisions should be placed. In the previous chapter, for example, you have decided that you wanted to use volts as the default units. However, this is not always the case. Imagine you use the same DAQ device to drive a piezo stage, volts are going to be translated into nanometers or microns and you have to account for that. In the experiment you are going to perform, you want to determine the current passing through a diode, but the physical quantity that you are actually measuring is a voltage.

Following the *onion principle*, it makes sense to define volts as the default units for the device, because it is what it outputs and what it measures. If you later want to use the same model for another purpose, it is handy that it reflects what it does rather than already calibrated to measure a current instead of a voltage. However, you can build a more general model that takes into account how to translate from voltage to current, or any other calibration that you may need.

A model for experiments will allow you to quickly establish the interactions between different devices. For example, you may wish to switch on a laser only if the interlock signal is on. Or you may want to close a shutter as soon as a scan is over. For the measurement of the I-V curve, you are going to be dealing with different inputs and outputs from just one device, but in more complicated setups this is not always going to be the case.

Writing an experiment class has a twofold advantage. On one hand, it is naturally going to lead to reproducible experiments. If you keep track of the parameters you use and do version control on the model, you can go back exactly to the same conditions in which you have performed a specific measurement. On the other, the experiment model is going to act as a backend on which it is going to be relatively easy to build a Graphical User Interface (GUI) with elements that are easy to reuse. More on this is going to appear later on in the book.

6.3 Starting the Experiment Class

The first step is to create a folder called **experiment** inside the **Model** folder. First you need to create a class and defe the properties that you are going to need for performing an experiment. Create a file called **IV_measurement.py** and add the following to it:

```
class Experiment():
    def __init__(self, user=None):
        self.user = user
        self.daq = None
        self.properties = {}
```

You can see that the class starts defining tree properties, a `user`, a `daq`, and `properties`. The user is set to `None` by default, but it is useful to keep track of who has generated data. Sadly, there is no way of doing this in an automatic way if the lab computer only has a user-defined. However, you can try the package *getpass*, which allows you to get the username by typing `getpass.getuser()`.

For the time being, the experiment class is independent, meaning that you don't need to import anything from other packages nor from your own code. Since measuring the IV curve depends only on a DAQ card, it is practical to have a dedicated property for it. Soon you will see how to develop a way of importing the daq into the class. Finally, you have set a `properties` dictionary. If you remember from some chapters ago, you have defined some YAML files to determine what properties you will use. It makes sense, therefore, to store them directly as a dictionary within the class itself.

At this point is important to show what kind of functionality we are after. If you were performing a measurement, you would simply do the following:

```
>>> from Model.experiment import IVCurve
>>> exp = IVCurve(user='Me')
>>> exp.load_properties('Config/experiment.yml')
>>> exp.load_daq()
>>> exp.do_scan()
>>> exp.plot_results()
>>> exp.save_data()
>>> exp.save_meta_data()
```

Step by step, import the model for the experiment that you want to perform. Instantiate it with a specific user, in this case just `'Me'`. Then you specify where the properties are stored and load them. You load and initialize the communication with the daq, based on what the properties say. To perform a scan based on the properties, there is a method called `do_scan`. In principle, the model could have a quick way of plotting the data and, more importantly, to save the data and metadata needed to reproduce the measurement.

One of the advantages of defining a class for a measurement is that you can also use it in other projects. For example, you could trigger a measurement from a Jupyter notebook and analyze the results directly within it. You can also trigger measurements remotely and you will be able to programmatically change the parameters that you used for a scan. Let's start simple and build on complexity.

Exercise

Write a method for loading the properties from a YAML file. Remember that you have already done a version of this in the Lay Out chapter.

Once you have a method for loading a YAML file with the properties for the experiment, you also know all the properties that you have available to perform a measurement. Which brings you to the following exercise:

Exercise

Write a `do_scan` method that relies on the parameters stored at `properties` to perform a measurement. Assume that `self.daq` is the model for the daq you are going to use. Store the data of the measurement in a property of the experiment class, remember that you should add them to the `__init__`.

The two previous exercises are based on things that you have done more than once in the previous chapters. But in this chapter you are making all these information systematically available, not only to you but to other developers. Moreover, once you start using the class in a real scenario, you will really appreciate the advantages of this approach.

Exercise

Write a method for saving the metadata of your experiment, i.e., all the information stored in the `self.properties` dictionary. **TIP**: Where do you save the data? Perhaps you can add one more key to the original YAML file to specify the destination folder.

So far, you have developed the basic methods for performing an experiment and saving its data to a file. We are missing the crucial step of loading the DAQ model into the experiment model. There are two options to achieve it. The first one is that you manually import and initialize the DAQ and then you pass it to the experiment model. The second is to load the appropriate daq based on what is defined in the configuration file.

The code for the first option would look like this:

```python
from Model.daq import DummyDaq
from Model.experiment import DaqControl

daq = DummyDaq()
daq.initialize()

e = DaqControl(user='me')
e.load_daq(daq)
```

While the code for the second option looks like this:

```python
from Model.experiment import DaqControl

config_file = 'Config/experiment.yml'
e = DaqControl(user='me')
e.load_config(config_file)
e.load_daq()
```

The two approaches are very different. In the first one, you are importing the daq you want, you initialize it, and then you add it to the experiment class. If later you want to change the model you use for performing the experiment, you will need to change the code. This may seem simple enough, but it will become harder when you have an experiment with several devices. Moreover, if you don't define which device you are using in the `properties` dictionary, that information won't be saved into the metadata.

Loading a module based on information stored as a string in a dictionary is slightly more involved, but it is also a good opportunity to learn few new things about Python. Moreover, you can combine both approaches into the same method. The only difference is whether you add a DAQ model as an argument of the method or not. You will also refactor parts of your previous code in order to make it more robust.

6.3.1 Loading a DAQ

First, you can start by the simplest approach, which is to import and initialize the daq outside of the experiment model and then load it. The method `load_daq` will look like this:

```python
def load_daq(self, daq):
    self.daq = daq
    self.properties['Experiment']['DAQ']['name'] = str(self.daq)
    self.properties['Experiment']['DAQ']['port'] = self.daq.port
```

The code is straightforward. The `load_daq` method gets one argument, daq, that is stored into the class property `self.daq`. You also update the `properties` dictionary to keep the name of the model that you are using. There are few things that need to be addressed. First, is to check what happens when you do `str(self.daq)`. Open a terminal and start python.

```python
>>> from Model.daq import DummyDaq
>>> daq = DummyDaq(port=1)
>>> str(daq)
'<dummy_daq.DummyDaq object at 0x7fe8c53c0c50>'
```

The result of doing `str(daq)` may not be what you were expecting. It is, however, a good starting point, because it allows you to identify which class did you use to define the daq. Fortunately, this can be improved. Open the file where you store the `DummyDaq` class and add the following to it:

```
def __str__(self):
    return "DummyDaq"
```

If you repeat the steps above, you will see that now, the output of doing `str(daq)` is exactly `"DummyDaq"`. The `__str__` method of a class tells Python how to convert an object to a string. It is also responsible for the behavior of `print(daq)`. Now you will have a nicely formatted string to store in your `properties` dictionary. Remember that if you assign the same output for the `__str__` method, to two different classes, you won't be able to distinguish which one you used.

There is something else that is worth learning at this stage. If the user supplies a class, you won't be certain that it is the correct class. It is wise to verify every user input that can alter the functioning of the program. For example, loading the daq can happen at the beginning of your code, but perhaps you use it much later. If an error occurs because it was the wrong class the program is going to crash and you are going to lose very valuable time because of a simple mistake.

If you think about what the DAQ Models have in common, you should realize that it is that both of them are child classes of the DAQBase class. Therefore, if you want to know if you are dealing with the proper model, you should check whether the daq variable that `load_daq` is getting is actually derived from the base model. There is no easy way of doing this in Python, and therefore you can build a workaround. First, update the base model for the daq, adding a property called `_type`:

```
class DAQBase(object):
    _type = 'DAQModel'
    def __init__(self, port):
        pass
```

Every child of `DAQBase` will have a property called `_type` that can be used to identify the model. Therefore, you can update the `load_daq` in the experiment model with the following:

```
def load_daq(self, daq):
    if daq._type is not 'DAQModel':
        raise Exception('The DAQ Model specified \
                    should be a child of DAQ Base')
    [...]
```

It is not a particularly elegant solution, but it works in most cases. Remember that with this approach you should be consistent in the naming conventions. If different models end up being called DAQModel, the check may give you unexpected warnings. The same works for the `__str__` method since you are using it to identify the model you used, if two different models end up calling themselves in the same way, you won't be able to tell which one you used.

The first approach on how to load the daq model into the experiment is relatively easy. Now it is time to work on the second approach. The idea is that you will load a specific class based on what the YAML file specifies. In the first approach, you have updated the properties of the experiment model, adding an extra key called `DAQ`. Therefore, you should do the same in the YAML file **experiment.yml**:

```
User: me
DAQ:
  name: DummyDaq
  port: 0
```

Next time you use the method `load_config`, you will also have a key for the DAQ. Importing a module into Python can also be done within a function, at any step of the program. You can also do it within the `load_daq` method of the Experiment class. Since you want to be able to provide a daq externally, you have to do the following:

```
def load_daq(self, daq=None):
    if daq is None:
        name = self.properties['Experiment']['DAQ']['name']
        port = self.properties['Experiment']['DAQ']['port']
        if name == 'DummyDaq':
            from PythonForTheLab.Model.daq import DummyDaq
            self.daq = DummyDaq(port)

        elif self.properties['DAQ']['name'] == 'RealDaq':
            from PythonForTheLab.Model.daq import AnalogDaq
            self.daq = AnalogDaq(port)
        else:
            raise Exception('The daq specified is not yet supported')

    else:
        # Here goes the same code you had before
```

The first to notice is that we have added a default value for the `daq` argument. Adding a default value allows you to use the method both as `load_daq()` and as `load_daq(daq)`. If you call the method without an argument, `if daq is None` will be `True`. The way of importing the different models is quite straightforward, you check the name and make the import. However, it is not very flexible. If you ever add a model for a new device, you need to manually add it to the experiment model.

Exercise

Go ahead and trigger the exception by defining a name of the DAQ different from 'DummyDaq' or 'RealDaq'. What happens to the code after the exception? Does it get executed?

The way in which you are handling the import of different classes in the code above is not a particularly elegant solution, but it works. Switching between DAQs is something that can happen, but it won't happen often enough to justify more complexity at this time. It is worth noting, however, that if you plan to make your code available to users with different devices, you should find a less pedestrian way of doing importing and initializing the classes that you need. However, it is a more complex behavior, and it will not be discussed in this book. If you are interested in this and other useful ideas, you can check *Advance topics for Python in the Lab*.

6.4 Defining a Scan

The goal of the project is to measure the IV curve of a diode. Therefore, doing a scan means changing an analog voltage output in a controlled way and, at the same time, recording an analog input. This kind of measurement is very common in a lot of different experiments, not only for electronics.

Exercise

Think at least three different examples of experiments that you can perform by changing an analog output and recording an analog input.

Performing a scan requires some parameters, that you should have defined in previous chapters, in any case, you can find them below as well.

```
Scan:
  port_in: 2
  port_out: 1
  start: 0.1V
  stop: 0.7V
  step: 0.1V
  delay: 10ms
```

`port_in` and `port_out` are the numbers of the ports needed for the scan, i.e., the analog in and the analog out. `start`, `stop` and `step` are the analog output parameters, note that units are added, and we will see how to deal with them later. It is important to point out that they don't need to have the same unit but they have to be the same quantity. For example, you can mix `mV`, `nV`, `V`, etc. The delay means how often the output is going to change from one value to the next.

The advantage of starting by defining the parameters in a YAML file is that now you know what parameters do you have available and what are their names. The next step is to create the method for doing the scan. You have already done the scan before, so you can adapt the code that you have developed in the past. If you remember, the general idea is to generate an output and record an input within a *for loop* with the range given by `start`, `stop` and `step`. You should have something like this:

```python
def do_scan(self):
    scan_params = self.properties['Experiment']['Scan']

    start = scan_params['start']
    stop = scan_params['stop']
    step = scan_params['step']
    port_in = scan_params['port_in']
    port_out = scan_params['port_out']
    delay = scan_params['delay']
    num_points = int((start-stop)/step) + 1

    scan_values = np.linspace(start, stop, num_points)

    for value in scan_values:
        value = value
        self.daq.set_analog_value(port_out, value)
        data = self.read_analog(port_in)
            sleep(delay)
```

The code is quite clear. The first step is to store the parameters of the scan into a variable called `scan_params`. This is only a way of shortening the amount of typing required later on. The loop is responsible for changing the output value and reading the analog input. As you can probably see, there are some issues that you will need to solve, one by one.

Exercise

In the loop, the variable `data` is being overwritten on every step. This is not what you want if you intend to do anything with it. Store the information into an array. The array should be a property of the class, i.e. it should be defined in the `__init__` method and you can refer to it as `self.NAME_OF_VARIABLE`.

The advantage of having the data stored in a property is that it becomes available to other methods of the class and that it is accessible from outside of it, for example for plotting. However, storing the data is only one of the concerns that need to be addressed. The other one is that the parameters that you are using have units attached to them, and yaml is reading them as a string that needs to be transformed into units.

We have covered how to use units in Python when you developed the first driver for the PFTL DAQ. You have to use the Unit Registry from a package called Pint. The code looked like this:

```
>>> from pint import UnitRegistry
>>> ur = UnitRegistry()
>>> var = ur.Quantity('5V')
```

With this code, *Pint* interprets a string, `'5V'` and transforms it into a quantity. You could add the same code at the beginning of the *experiment* model, and transform the parameters when you use them, or you can have a more elegant approach. In the root of your project, the folder that contains the folders *Model*, *View* and *Controller*, create a file called **__init__.py** and add the following code to it:

```
from pint import UnitRegistry
ureg = UnitRegistry()
Q_ = ureg.Quantity
```

At the top of the file where you define the experiment model, you can add:

```
from PythonForTheLab import Q_
```

And you can use `Q_` for transforming the parameters into quantities, simply doing `Q_(parameter)`.

Exercise

Update the code to transform each one of the appropriate parameters to a Quantity. Remember that the port numbers are supposed to be integers, not quantities.

Once you start working with units, you need to take care of other concerns. For example, to calculate the number of points in your scan, you should transform all the parameters to the same quantity. One of the possible ways of doing it is by picking the units of one of the parameters, like this:

```
units = start.u
stop = stop.to(units)
step = step.to(units)
num_points = (stop-start)/step
num_points = int(num_points.m_as(''))+1
```

The `u` grabs the units from a *Quantity*. You transform everything into the same units to avoid problems with the number of points you are computing. However, when dividing quantities, the result will be another quantity, just that *dimensionless*. To transform it to a number, you have to append `.m_as('')`. Doing `num_points.m_as('')` means that you want to take the magnitude of the variable, without units, and the `''` are there because we are dealing with a dimensionless quantity. It is possible to do things like `start.m_as('pV')`.

There is one more parameter that you need to take care of. The `delay` has units of time, and the function `sleep` takes seconds as an argument. The easiest way to deal with it is simply writing the following:

```
sleep(delay.m_as('s'))
```

Which ensures that you use the delay magnitude expressed in seconds. If you use milliseconds or hours for that matter, it will be translated into the proper number before the `sleep` function uses it. With this changes, your code is ready for performing a scan. There are some concerns, such as how do you stop a scan while it is running. But the core ideas and methods are already there.

6.5 Conclusions

With this chapter, the core code for a functioning experiment is finished. Since the beginning of the book, you have developed from a driver to the logic of how to perform a scan, including how to use text files to store the parameters that you need. Of course there are things that can be done better but, in any case, what you have achieved so far is functioning and useful.

When you develop your own program, it can be wise to start with the experiment model. Once you established the steps of a measurement it will become clear what do you need to develop in order to achieve it. It is a good way of being focused on a clear objective. If you are developing a driver, or a model, without knowing to what end, you may end up creating methods and options that you don't really need.

In this chapter, you also start to realize that if you plan to open your code to others, not just as developers, but as users, you will need to verify the input that they supply. For example, you learned how to verify that the DAQ model is actually what you want. Following that idea, there are several improvements to your code, for example, you can check if the units of the parameters used are correct.

6.6 Extras: Defining a Monitor

The clear objective of the book is to show you how to build the software that you need in order to measure an I-V curve of a diode. However, there are some other options that you can add to the program, such as continuous monitoring of a signal. In a lot of experiments, measuring a signal is all you need, especially in the phase of setting up. Imagine that you are monitoring a temperature, perhaps you need to wait until it reaches a certain value, or you monitor the signal generated by a photodiode in order to align a laser.

Monitoring a signal is somehow simpler than performing a scan because you have fewer options. The parameters that you need are the port to monitor, the total time, and time resolution of the monitor. The total time is for how long do you want to acquire a signal before it starts overwriting the information, and the time resolution defines the time interval between consecutive acquisitions. The main difficulty with a monitor is that the loop should be infinite and you should be able to stop it somehow. We will see that in the next chapter, for the time being, you can complete the exercises:

Exercise

Add one more block in the YAML file with the parameters that you need to monitor a signal

Exercise

Define a new method, called `monitor_signal` that used the parameters defined in the previous exercise to continuously acquire a value. If you want to test how the monitor works, you can make the loop stop after a certain amount of time, for example using the module `time.time()`.
HINT: You should define a numpy array where to store the data. In order to move all the values one step to the left (or right), you can use `np.roll`.

Chapter 7

Run an experiment

7.1 Objectives

In this chapter, you are going to learn how to run a proper experiment. You have developed all the code you need to acquire data, but you need to put everything together and make it work. This chapter is also the opportunity to wrap up everything that you have achieved so far and be sure that all the classes, packages and configuration files are where they are supposed to be.

Once you finish this chapter, you will have working knowledge on how to start controlling your own experiments in the lab. Of course, you will need to adapt the concepts explained in the book to the devices and settings that you have, but the foundations are strong enough to build reliable and expandable software.

7.2 Introduction

The contents of this chapter are going to become ubiquitous in almost all your lab experiments. Most likely a big part of your job is to either repeat measurements in order to improve your statistics or to change some parameters in order to find correlations in your data. A smart way of doing both tasks is through a computer with the appropriate devices hooked to it and with the appropriate software to control them.

Up to now, you have developed a package for controlling your experiment, but now you have to use the package. It is a good idea to separate the files that build up the package from the ones that specify a specific status of the setup. That is why you have created a separate folder for *Examples*. But this is not enough, when you run the experiment and save the data, you will need to do it in a different location. This will prevent polluting the base code and will allow you to separate the settings from different users working on the same computer.

Remember, that in order to be able to reproduce results of your experiments, it is important to keep track of the parameters used to perform a measurement. Through this chapter, you are going to learn some ideas on how to use YAML files to store the metadata of your experiment, together with the data and use it to repeat a measurement. Moreover, you will learn how you can extend the Experiment model without writing new methods.

7.3 The Layout of the Project

In the previous chapters, you have developed a lot of code, and it may have ended up in the wrong location. That is why it is important to recap on how the structure of the package looks like.

```
ROOT
   Docs
   Examples
       Config
           experiment.yml
   PythonForTheLab
       Controller
           __init__.py
           simple_daq.py
       __init__.py
       Model
           daq
               analog_daq.py
               base.py
               dummy_daq.py
               __init__.py
           experiment
               __init__.py
               IV_measurement.py
           __init__.py
       View
   README.md
```

If you don’t have the same structure in your project, you will have to either adapt the code for making the proper imports or reproduce the structure. Pay attention to the fact that every folder has its own *__init__.py* folder, and in some of them there are instructions for making the imports shorter. If you see something that you can't understand, you can always refer to the repository on Github.

For the contents of this chapter to work, you should have the ROOT folder added to the path. In this way, you can use it as

```
from PythonForTheLab.Controller import SimpleDaq
```

regardless of where you are in your computer. It was briefly explained in Chapter 4 when you learned how to import different modules to Python.

After you have added the folder the PATH in your computer, you can start working in any folder that you wish. The files that we are going to create from now on in this chapter are not going to be part of the main package, so you don't mix them, just create a separate folder wherever you like.

7.4 YAML file for configuring

In our experiment, we are going to perform a 1D scan of the voltage generated at an analog output, while recording the voltage at an analog input. The first step is to create a file **experiment.yml** with the needed parameters. The contents should follow the contents that you have in the examples folder. It can be something like this:

```
User:
  name: Me
Experiment:
  DAQ:
    name: DummyDaq
    port: 1
  Scan:
    port_in: 0
    port_out: 0
    start: 0.V
    stop: 3.3V
    step: 100mV
    delay: 200ms
```

7.5 Running an Experiment

Once the configuration file is defined, you have to start using the experiment model. Since the program will run from the command line, create a file called **start_cli.py**, which you will trigger to perform the measurement, and now you can start adding what you need.

```python
from PythonForTheLab.Model import Experiment

e = Experiment()
e.load_config('experiment.yml')
e.load_daq()
```

Remember to change the path to the YAML file with the configuration for your experiment. In principle, you could have several configurations for different experiments and you load each one accordingly.

Exercise

Now already know how to trigger a scan with the default parameters. Perform a measurement and print the data to screen.

If you use the experiment model as is, and you trigger the `do_scan` method, you will notice several things. The first is that you don't know if the program is running or not. When you are facing this situation, the easiest solution is to use some `print` statements in order to know where you are at. For example, you can add `print(value)` inside the loop for the scan, in this way you can see the value that you are passing to the analog output being printed to screen. Printing to screen is also a useful debugging tool. For example, you could print to screen the parameters for a scan before performing it, to be sure that everything is in order.

The other thing that perhaps you have realized is that if you want to stop the scan, you have to interrupt the execution of the program, which leads to unexpected results, for example, you are going to lose the acquired data. Finally, you should have realized that the program blocks the execution of any other code while the scan is running. If you want to plot the progress, you will have to wait until it is finished. Solving this is slightly more complicated because it is going to imply the use of *threads*. However, there is something important to point out first.

Once you instantiate the Experiment class and load the config file, you will have a dictionary located at `e.properties` with all the information contained in **experiment.yml**. Those parameters are used by the `do_scan` method, but nothing prevents you from changing them before doing it. For example, you can add the following to the code:

```
e.properties['Experiment']['Scan']['start'] = '0.5V'
e.properties['Experiment']['Scan']['step'] = '500mV'
e.do_scan()
```

When the scan is performed, the parameters that are going to be used are the ones just defined, and not the ones in the YAML file. This is very handy because you could even wrap the code in order to study the dependence of the results with some parameters which perhaps you didn't think of before. For example, imagine that you want to see if the IV curve depends on the step size or the delay between points. You could do the following:

```
for delay in range(10, 100):
    d = delay*Q_('ms')
    e.properties['Experiment']['Scan']['delay'] = d
    e.do_scan()
```

You can see that by having a model class for the experiment, the biggest work was already resolved. There are some details that you will have to address, for example, how to save the data, but they are normally simple to resolve. Also, you have to remember that the scan is happening with parameters not defined in the YAML file. In order to maintain the idea of doing reproducible research, you should always save the parameters you used as part of the results.

7.6 Plot results and save data

You know how to perform one or several measurements, but now you need to do something with the data you have just acquired. First, you are going to plot it and then you are going to save it to a text file.

For plotting you are going to use a library called PyQtGraph which was developed mainly to generate fast data visualizations. If you are familiar with any other plotting libraries such as matplotlib, feel more than free to adapt the example to them. The examples are based on PyQtGraph because it is the library that you are going to use while building a GUI. The first thing you have to do is to import the library and make a plot of the data.

```
import pyqtgraph as pg

pg.plot(e.xdata_scan, e.ydata_scan)
```

It is a super simple example but it works. You are using the data stored directly within the class. If you called the data differently, adapt the code. At this point, you should see the curve of your results appearing. If you want to add labels and a title to the plot, you can do the following:

```
PlotWidget = pg.plot(title="Plotting I vs V")
PlotWidget.setLabel('bottom', 'Port: {}'.format(
    e.properties['Scan']['port_out']), units="V")

PlotWidget.setLabel('left', 'Port: {}'.format(
    e.properties['Scan']['port_in']), units="V")

PlotWidget.plot(e.xdata_scan, e.ydata_scan)
```

There is no much complexity at this point; once you have the data it is endless the number of things you can do. You can fit a model, repeat a new scan with parameters based on your results, etc. However, saving the data is crucial if you are intending to use it in a publication or if you want to do a better analysis at a later stage. You should have developed a method for saving data in the experiment model, but in case you haven't done it yet, you can follow the example below:

```
from datetime import datetime

filename = "Data_scan_{:03}.dat"

header = "# Data saved by Python For The Lab\n"
header += "# Creation date: {:%Y-%m-%d %H:%M:%S}\n".format(
    datetime.now())

header += "# Number of lines: {}\n".format(len(e.xdata_scan))
header += "####################################\n"
```

First, you define the filename in such a way that we can add numbers to it. The `\{:03\}` means that there are going to be 3 spaces for the number, therefore the filename will look like **Data_scan_001.dat**, **Data_scan_002.dat**, etc. You can try it if you do something like

```python
print(filename.format(34))
```

Later you define the header of the file. First, you have a title and you also write down the creation date and the number of points you have acquired. The creation date is essential for keeping a tidy record because it will allow you to match the data with whatever you have written in your lab journal. The number of lines is a bit old-fashioned; it is very useful when you are reading line by line and you need to stop right at the last line before an error occurs. Note that at the end of each line there is a `\n` in order to force a new line after each statement.

Now you have to be sure that you are not overriding a previous file with the new one. You need to check if the file exists, like this:

```python
import os
os.path.exists(filename.format(0))
```

The command will return `True` if the file `Data_scan_000.dat` exists or `False` if not.

Exercise

Check what is the smaller number for which a file doesn't exist. *Hint*: the `while` loop is your best friend in this case.

Once we know which number is available for saving our file, you will just write the data. Assuming that the number for the file is `i`:

```python
with open(filename.format(i), 'w') as f:
    f.write(header)
    for i in range(len(e.xdata_scan)):
        line = "{:4.4f}, {:4.4f}\n".format(
                e.xdata_scan[i], e.ydata_scan[i])
        f.write(line)
```

As you can see, the header is written in just one go. Thanks to the `\n` characters that were appended to every line it is going to be correctly displayed if you open the file. The saving of the data itself happens in a for loop, that covers every data point acquired. The format specifies that each point is going to be saved with a precision of 4 decimal points. If you want to read about the possible formats, you can check PyFormat or the Python Documentation. Go ahead and open the file to see how does the final version looks like.

If you are confident with the results, you can add this procedure as a method in the experiment model. You can also add a parameter to the yaml file for specifying the name of the files that you are producing.

Remember, if you change the code and a new parameter is needed, you have to update the file in the *Examples* folder to reflect it. If you fail to do so, you will be solving a lot of issues every time someone wants to start a new experiment.

7.7 Running the scan in a nonblocking way

When a function takes long to execute, such as the case fo the `do_scan`, it is said that the function is blocking the rest of the program. Sometimes the functions take long to execute because they are computationally expensive, sometimes they take longer to execute because the process is slow. For instance, when you do a scan, the majority of the time the program is waiting in a `sleep` before changing the value and reading again. Waiting is not computationally expensive and therefore your computer is capable of performing other tasks at the same time.

The idea of what you want to achieve is that the long function, in this case, `do_scan` should run in the background while the rest of the program is able to do some other things. For example, you can plot the results while you are acquiring them, or you wish to stop the scan without killing the program. Python offers a very interesting package called `threading`, which will allow you to do exactly what you are looking for.

7.8 Threads in Python

You can think about threads as different lines of execution. A Thread would be similar to opening different terminals and running Python in each one of them. The advantage is that since you are running all your threads in the same Python interpreter, you can easily exchange information within them. Let's see first a simple example of a function that takes long to execute:

```python
import threading
from time import sleep

def func(steps):
    for i in range(steps):
        sleep(1)
        print('Step: {}'.format(i))
```

```
t = threading.Thread(target=func, args=(3, ))
print('Here')
t.start()
sleep(2)
print('There')
```

If you run the code above, you should see the following output:

```
Here
Step: 0
Step: 1
There
Step: 2
Step: 3
```

What is happening is very interesting. If you run the function `func` on its own, you will see that the numbers appear one by one, and while the function doesn't end, you won't be able to do anything else. However, when you create a thread, i.e. when you execute `threading.Thread`, the target function is going to run in a separate space, therefore not blocking your program. When you create a *thread*, you have to specify the function you are going to execute, bearing in mind that it is the function itself and not the instance, i.e., you don't append the `()`. If the function takes arguments, you add them as a comma-separated list, even if it is only one as in the example above.

When you create the thread, the function is not executed, it is waiting for the `start()` signal to start running. What you see is that even after the thread starts, the `print('There')` statement is being executed. First, you print `'Here'`, you start the *thread*, you wait for two seconds, in which `func` prints its first two steps, you print `'There'` while the thread continues its execution. Working with threads can get very complicated when you need to share information, or when you need to be sure one specific thread finished before you do something else, etc. For simple examples, however, it is very handy and easy to understand.

You can also have several threads running at the same time, and you can assign names to them:

Python

```
def func(steps):
    print(threading.currentThread().getName(), 'Starting')
    for i in range(steps):
        sleep(1)
        print('Step: {}'.format(i))
    print(threading.currentThread().getName(), 'Finishing')


first_t = threading.Thread(name='First Thread',
                           target=func, args=(3, ))
second_t = threading.Thread(name='Second Thread',
                           target=func, args=(3, ))
print('Here')
first_t.start()
second_t.start()
sleep(2)
print('There')
```

Go ahead and run the script to see what happens.

7.9 Threads for the experiment model

Going back to the experiment model, you can already test what you have learned by executing the `do_scan` method in its own thread. Since the method doesn't take any arguments, the line should look like:

```
t = threading.Thread(target=e.do_scan)
```

While the scan is running in the background, you can do other things in the main thread, such as plotting. Let's see how you can do that. You need to refresh the plot while the acquisition is happening, and therefore you will need to update the plot within a loop. One possible way of doing it is like this:

```
import threading
import pyqtgraph as pg
from time import sleep
from PythonForTheLab.Model import Experiment

e = Experiment()
e.load_config('config.yml')
e.load_daq()
t = threading.Thread(target=e.do_scan)
t.start()

PlotWidget = pg.plot(title="Plotting I vs V")
PlotWidget.setLabel('bottom', 'Port: {}'.format(
        e.properties['Scan']['port_out']), units="V")
PlotWidget.setLabel('left', 'Port: {}'.format(
        e.properties['Scan']['port_in']), units="V")
PlotWidget.plot(e.xdata_scan, e.ydata_scan)

while t.isAlive():
    PlotWidget.plot(e.xdata_scan, e.ydata_scan, clear=True)
    pg.QtGui.QApplication.processEvents()
    sleep(1)
```

The beginning of the code is a combination of what you have been doing for running the scan and plotting and the use of threads. The important part is what is happening in the `while` loop. First, you check that the thread is still running with the `isAlive()` method. Then, within the loop, you update the plot with the data in the experiment class. It is very important to note that even if the `do_scan` method is running in a different thread, its data is accessible from the main thread. There is an extra line in which you declare a `QApplication`. For the time being, do not worry about it. Its only purpose is to use the plot that you have already created a few lines above.

Exchanging information with a class means that you cannot only read from it but that you can also write to it. Therefore, a clever way of stopping a scan is by checking the status of a property within the loop. For example, you can add the following code to the `do_scan` method of the experiment model:

```
if self.stop_scan:
    break
```

What is happening is that if the property `stop_scan` is set to `True`, the for-loop in which you are running the scan will stop. Remember that if you add a new property such as `stop_scan`, you have to initialize it in the `__init__` method of the class. It is also important to set the property to `False` every time you start a new scan. If you fail to do that, after you stop a scan you won't be able to run another one. The `do_scan` method will look like this:

```
def do_scan(self):
    self.stop_scan = False
    [...]
    for value in scan:
        if self.stop_scan:
            break
    [...]
```

The `[...]` means that there is code that is not being displayed, for brevity.

Exercise

Run the example with the threads again, but this time change the value of `e.stop_scan` to `True` after a certain number of plot updates.

Exercise

Use the input from the keyboard to stop the scan

Finally, it is important to point out that sometimes you actually need to wait for a thread to finish. Imagine that you want to perform several scans, you don't want to do them at the same time, but one after the other. To wait for a thread to finish you can use the method `join()`, like this:

```
t = threading.Thread(target=e.do_scan)
t.start()
print('Thread Running')
t.join()
print('Thread Stopped')
```

In this example, `Thread Stopped` will appear only after the Thread has finished. Moreover, when you start working with this design pattern in mind, you will realize that there is nothing that forbids the user from triggering two scans at the same time, in two different threads. This can give a lot of headaches when you start developing complex applications. Sometimes, the errors that appear are going to be very hard to debug. It is very hard to transmit all the different things that can happen once a different person starts using your software.

Exercise

Add an extra property to the experiment model called `scan_is_running`. Set it to `True` when the scan runs and `False` when the scan stops. Use it to avoid two scans to run simultaneously.

Threads work also in Jupyter notebooks. If you start a thread in a cell, you will see that it immediately releases the interpreter. You can move to a different thread and keep working, updating a plot, etc. Whenever you want to stop the scan, you can do it as before, changing the value of the `stop_scan` property.

7.10 Word of Caution with Threads

Threads are a very complex topic in any programming language, and even if the examples that you have seen up to now seem straightforward, you need to be cautious about how you implement threads in larger programs. Today computers with multi-core processors are ubiquitous. However, when you run a Python program it is going to run only within one of the cores of the computer. Moreover, each core can run only one thread at a time.

When you run multiple threads, the processor very quickly switches from one thread to another. The first thread runs for a while, then it stops, another thread runs for a while, stops, etc. Switching from a thread to another is not free for the processor and even if the process is not doing anything, the processor has no way of knowing it and will switch to it anyways. While you have processes like the ones you have developed up to now, which are not computationally expensive, you won't notice performance issues. However, if you start dealing with image analysis, data processing, etc. you will have to find better alternatives.

If you want to test the limits of threading in Python, you can generate large random arrays. Run the process on different threads and check the status of your computer. You will notice that not all the cores available are in use, but one of them is going to be at 100%. Moreover, you can check what takes longer if generating 4 random arrays in 4 different threads or one after the other.

You shouldn't confuse the idea of multithreading with parallelizing code. When you parallelize code you are able to use all the cores of your computer at the same time. Threading just allows you to run code at the same time on the same processor. The subject is vast and exceeds the objectives of this book. However, you need to be aware of the limitations of the techniques that you use and the degree of applicability that they have. If you want to develop code that runs in parallel, you can look at libraries such as *mpi* or *multiprocessing*.

7.11 Conclusions

This is a chapter where you can see how all your previous work plays out nicely together. Developing drivers and models may seem a lot of unnecessary work, but when you see everything coming together you start to realize that it really pays off being organized and systematic with the code. If you stop reading the book in this chapter, you would have already acquired a lot of knowledge that can improve your work in the lab. You have learned the MVC pattern, you have developed models, and worked with threads. You have also seen how to store metadata and use it to run an experiment again.

The most important message that you should take with you after these few chapters is that the code you write today should be useful tomorrow. You have started with simple tasks, such as communicating with a serial device, and some chapters later, the same code is being used for performing experiments. If you think about it, the base driver was never updated since you first developed it, it just changed folders. This is, in principle, what you expect to happen with all the code you develop. Each class, each function, can be a building block for something else. You will see how this materializes in the next two chapters when you develop a Graphical User Interface (GUI).

When you work in the lab, you will notice that every device is different, has different requisites, and a different behavior. It is very important that you understand, first of all, what do you want to do with the devices that you have available. Devices for the lab are not home appliances and may have options which you should understand by reading the manual. Perhaps you need external triggers, an amplifier, a filter, etc. Once you understand what experiment you are actually performing, then you can start programming, but not the other way around.

Chapter 8

Building a Graphical User Interface for the Experiment

8.1 Objectives

Building GUI's is a very rewarding experience because you will really see the outcome of your program. It is much more impressive to show to someone a beautiful window with a couple of buttons than hundreds of lines of code. When you develop a working User Interface, you will really have the feeling of delivering an end-product. A window pops up, an error message appears, an experiment is performed. In this chapter, you will learn the steps needed for designing a window and plotting the data acquired by a device. The chapter involves some design, some coding and a lot of thinking. It is strongly advised to drive inspiration from the examples in case you get stuck for too long in one of the exercises.

8.2 Introduction

Building GUIs may seem more complicated than what it really is. It is, of course, not an easy task but once you get the fundamentals you will see that, as with any other skill, it is possible to learn. Most experiments can be performed right from the command line, as you have seen in the previous examples. You can have a great level of control running a Python script, or import your models into a Jupyter notebook. However, in some cases, it can be very handy to control the parameters of an experiment and to monitor the output in a window especially designed.

There are several options for building GUIs with Python. For example, in the previous chapter, you have already seen how to plot the data while the experiment is running. It is a bit rudimentary but it is indeed a *user interface*. In this chapter, we are going to focus on one particular library for building interfaces called **Qt**. Qt was developed as an application framework that allows developers to build apps that look native in different systems without changes to the code. This means that if you develop an application and run it on Windows it will look like a Windows program, but the exact same application will look like a Linux program if you run it on such a computer.

Moreover, Qt comes with a lot of interesting features to connect different parts of your code. For example, when moving a slider in the UI, the value of a parameter in your experiment model should change. Qt also provides the tools you need to create all the elements that a program needs, from menus to buttons and the flexibility to define your own elements, such as a plot. Qt was not developed with Python in mind, but fortunately, some projects have advanced a lot in making it compatible. There are mainly two, called PyQt and PySide. The difference between them is the license under which they are released. If licenses are a concern for you, for example, because you are developing commercial software, you should definitely check their differences. In the case of this book, we are going to use PyQt5 because the programs we are developing have no commercial interests.

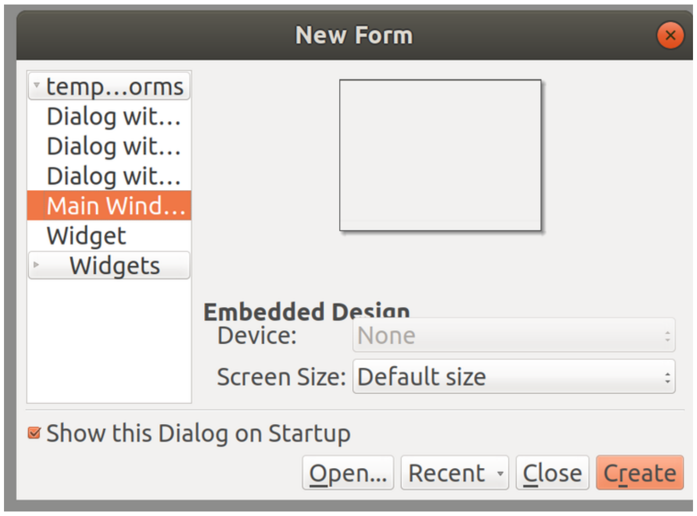
You have to be aware that the current version of Qt is **Qt5**, and therefore the examples in this and the next chapter are based on **PyQt5**. However, up to some years ago, the most spread version was **Qt4**. There is no backward compatibility between them, meaning that code that worked with **PyQt4** not necessarily will work with **PyQt5**. It is very important that you don't mix both libraries. If there are applications on the computer where you are developing the code that depends on *PyQt4*, it is very important to isolate your development with a Virtual Environment, as explained in Chapter 02.

In this chapter, you will find some design and some coding. Up to now, it was simple to include the code in the book for you to see, however, design files are not *copy/pasteable*. All the design files are available for you to download from the Github Repository.

8.3 Simple window and buttons

Now it is finally time to start using the empty folder from the M **V** C design pattern: the **View**. The folder will be a python module, and thus it needs a `__init__.py` file, like all other modules. Inside the **View**, create a folder called `GUI` where you are going to store the files that build the windows. Soon enough, you will see that when you work with Qt you will have two different types of files. One will be the way the program looks and will have *.ui* as the extension. Then, you will have the Python modules responsible for building up the interface.

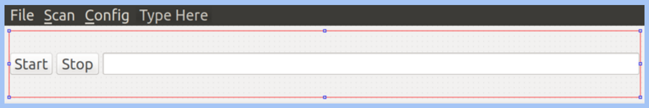
You will start by designing a simple window with two buttons, one to start and the other to stop a scan. You will also add a line where you can see if the scan is running or not. The first step is to open the program **Qt Designer**, which was installed in Chapter 02. The program will welcome you with a window like the one below:



Since you are creating your first window, you have to select *Main Window* as displayed. Don't worry too much about the technicalities at this stage, it is important that you start going and you will slowly get the concepts later on. The only important thing to know now is that in Qt, *widgets* are the building blocks of any user interface. A button is going to be a widget, a plot, slider, etc. are all widgets Now you can start playing around with the designer, and familiarize yourself with it.

Exercise

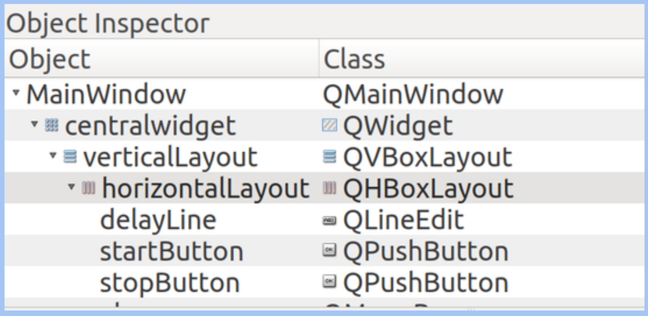
Generate a window that looks like the image.



If you look closely, you will notice a menu as well, at the top, called *File*. For the time being it is optional, but you will need the menu later on. The designer has a lot of options that can be useful, for example specifying the layout. The layout is a way in which objects are going to be arranged in the window; a vertical layout means that different objects are going to be stacked one under the other. You can also find a horizontal layout, a grid layout, and a form layout. Feel free to explore how they behave. Remember that you can nest them for enhanced flexibility. It is also possible to work without layouts, specifying the exact position and size of each widget in the window. As you can see, Qt doesn't impose design restrictions, it is really up to you to decide what and how to build things.

Exercise

Nest a horizontal layout (for the buttons) within a vertical layout that later on will hold the plot of the data.

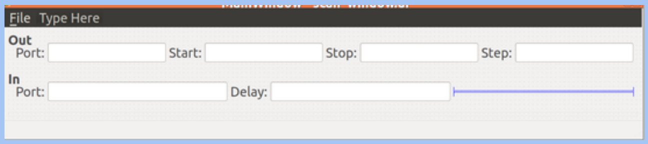


Object Inspector

Object	Class
MainWindow	QMainWindow
centralwidget	QWidget
verticalLayout	QVBoxLayout
horizontalLayout	QHBoxLayout
delayLine	QLineEdit
startButton	QPushButton
stopButton	QPushButton

Exercise

Improve the window to look like the image. This is going to be the entry point for your program, and from here you are going to configure and trigger scans.



At this point, you have an idea of what can be done with Qt Designer, but what you need to learn now is how to give different functionalities to the different widgets on the screen. There is, however, something important to pay attention when dealing with the designer. The names that you assign to each widget is going to be the way to identify it also within Python. However, it is very complicated to find out the name you gave to each element from the designer file, it is much easier and faster doing it directly from within the program itself. Since now you will be dealing with files coming from two different sources, i.e., Python files generated in an editor and designer files, it is very important that the naming conventions are consistent. If you are in doubt, or you see some bugs that you find to resolve, you can always download the files from the Github repository.

Note

If you look for documentation online on how to use PyQt in your applications, you most likely will find that you should compile the `uic` designer files into a python file, that you can import as any other module. This step is not at all necessary and will just add more difficulty to an already complicated process. Rendering the uic file directly in your code is much simpler, faster and more robust in the long term.

The best idea is to start simple. You are going to write a script that can generate and open an empty window. First, create a file called **simple_open_window.py** within the **Examples** folder and add the following to it:

```python
import sys

from PyQt5.QtWidgets import QApplication, QMainWindow

app = QApplication(sys.argv)
m = QMainWindow()
m.show()
app.exit(app.exec_())
```

If you run this script, you will see a window popping up in all its splendor. Let's see what is actually happening line by line. The `sys` module is responsible for everything that involves the operating system, as you shall see later. Then you import two modules from the PyQt5 library. You have to understand that in PyQt everything always starts with a `QApplication`. You have to start an application before you can open a window or create widgets. You also import `QMainWindow` which is the widget that is going to hold all the design elements needed for an application.

Every PyQt program will start in the same way. First, you define the `app` using the module `QApplication`. Pay attention to the fact that it takes one argument, in this case, `sys.argv`, which is a list of the commands passed when executing the script from the command line. For example, if you run the script as `python script.py -option test`, you will have a list in `sys.argv` stating that the *option* you used is *test*. PyQt allows you to specify some parameters, such as the style of the app. However, even if you are not going to use them, you have to specify an argument, and better be consistent throughout different programs.

In the next couple of lines, you create a `QMainWindow`, which is also what you have done in the Designer. You can already start seeing that there are two ways to achieve the same results, one is programmatically, writing the code to generate Qt windows and the other is graphical, with programs such as Qt Designer. After you've created the *Main Window*, you need to make it appear with the `show()` command. If you have ever used *Matplotlib* you can see the similarities.

Finally, you have run the `app`. Check that you do it in a nested way, you call `app.exit` and as an argument, you use `app.exec_()`. An App in PyQt can be thought of as an infinite loop that will keep every piece of the Qt program running. Normal Python scripts exit when they are done, but a GUI needs to run until you stop it. This is what `app.exec_()` does. By passing it as an argument to `app.exit` you can be sure that for whatever reason the execution is interrupted, the application process will be stopped. It is a safe way of terminating an application, which can be very important if you are running large projects.

Of course, a simple, empty window, is not what you are after. What is important to note is that if you run the same script on different computers, the outcome will be slightly different. The look and feel of every window will match with your operating system, this is one of the many advantages of using Qt. A good way of expanding the code is by including the file that you have developed with the designer.

```python
import sys
import os

from PyQt5.QtWidgets import QApplication, QMainWindow
from PyQt5 import uic

app = QApplication(sys.argv)
m = QMainWindow()
file_dir = '/home/user/programs/PythonForTheLab/PythonForTheLab/View'
uic.loadUi(os.path.join(file_dir, 'GUI/scan_window.ui'), m)
m.show()
app.exit(app.exec_())
```

The core of the script is the same as before, the new lines are for establishing the path of the *View* folder and to load the design file. You will see a better way of doing this in the program, but for the time being it works well. Go ahead and run the script. You will see that the window is displaying the design that you had generated. You should be proud of yourself. There is a good chance that this is the first time that you actually have an application written from scratch by yourself and running on your own computer.

8.4 Adding buttons and interaction

Now you have a window that may look great but it is not doing anything. You need to start building your code in order to allow actions to be triggered by pressing a button, for example. But for having code that is reusable and expandable, you have to rewrite your previous example. You will need to define a new class that inherits from `QMainWindow` and that allows you to personalize it as much as you want. You have to create a new file within the **View** folder and call it **scan_window.py**. You can adapt the previous example like this:

```python
from PyQt5 import QtWidgets, uic

class MainWindow(QtWidgets.QMainWindow):
    def __init__(self, parent=None):
        super().__init__(parent)

        p = os.path.dirname(__file__)
        uic.loadUi(os.path.join(p, 'GUI/scan_window.ui'), self)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    m = MainWindow()
    m.show()
    app.exit(app.exec_())
```

As you can see, the code after `if __name__` is almost the same than what you have done before. If you execute the file, you will notice that the window is opened with the design. Going back to the new class you have defined, you can see that it inherits from the `QMainWindow` class, because what you are developing is the main window of your program. The `__init__` method should always take one argument called `parent` because it is used by Qt to keep track of the relationship between widgets. When you use the `super()` to initialize the parent class, you need to pass the same parent.

You can see that the way to import the designer file is slightly modified. Since you need to perform a relative import, the first step is to establish the location of the current file, and that is what you do when you call the `__file__` property. Next, you append the path to the *ui* file to the location of the **scan_window.py** file and load it, as what you have done before. When you want to import files within Python, you have to specify precisely where they are located. Working with relative imports is not always safe with Python, especially for larger projects. Sometimes to root folder is not the folder that contains the file, but the folder from which you trigger the program. That is why it is always safer to use absolute paths for imports and not just relative ones.

Once you have a subclass of `QMainWindow`, you will have complete control over what happens in the window, where the user is clicking, you will also be able to add new elements, to update them, etc. The advantage of defining a new class is that you can reuse it in different projects and that it can be easily embedded into larger programs. For the time being, you only want to do a scan, but it can be that tomorrow the scan is only a part in a larger series of experiments and the main window becomes just one more window.

One of the building blocks of any app is buttons. However, if you downloaded the example design file from the Github repository, you will notice that there are no buttons in it, just a container called `buttonBox`. First, you will learn how to add elements to the GUI directly from the Python code, then you are going to learn how to trigger actions when a button is pressed. The logic behind Qt elements is that first, you initialize them, then you add them to the position where you desire and then you program how they are going to behave. Let's first create two buttons, one start, and one stop. You should add the following the to `MainWindow` class:

```
self.startButton = QtWidgets.QPushButton('&Start')
self.stopButton = QtWidgets.QPushButton('S&top')
self.buttonBox.addButton(self.startButton,
            QtWidgets.QDialogButtonBox.ActionRole)
self.buttonBox.addButton(self.stopButton,
            QtWidgets.QDialogButtonBox.ActionRole)
```

The string specified when declaring the buttons is the text that will appear in them. Adding an `\&` before a letter is just for enabling the user to quickly reach the button by pressing Alt+S or Alt+T. The next couple of lines are for adding the buttons to the `buttonBox`. You can add the buttons to any layout that you have specified within the designer. The main difference is that the method you have to use is `addWidget` and takes only one argument, the widget (in this case the button) that you wish to add.

Exercise

Create a designer file with different layouts and add buttons to each, in order to see how they behave.

Now that the buttons are ready in the GUI, let's start with a very simple idea: printing to the terminal `'Button Pressed'` when a button is pressed. To achieve this, you have to understand one of the most powerful components of Qt programming: **Signals**. A signal is what enables you to capture an event and trigger an action. You can hook those signals to one or more functions (also called *Slots*) and Qt will take care of executing them when appropriate. If you remember, the `QApplication` can be regarded as an infinite loop, taking care of the behavior of the applications, including the connection between *Signals* and *Slots*.

In Qt, there are a lot of different signals that get fired at very precise moments. For example, when you click with the mouse on a button it will generate a specific signal. But it is not the only one. Moving the mouse, releasing a button, hovering over an element, the list goes on and on. You can even trigger your own signals with your own conditions, for example when the user presses a specific combination of keys while moving the mouse. But before getting too complicated, let's just start adding some action when clicking the `startButton`. Update the Main Window class with the following line of code and function:

```
class MainWindow(QtGui.QMainWindow):
    def __init__(self, parent=None):
        [...]

        self.startButton.clicked.connect(self.button_clicked)

    def button_clicked(self):
        print('Clicked')

    [...]
```

If you just run the file and click on the start button, you will see that there is a message being printed to the terminal. What is happening is that the `startButton` has a `signal` called `clicked` and you have connected that signal to the method `button_clicked`. Bear in mind that you don't put the `()` when you connect the signal, you are interested in just the method. `button_clicked` is very simple, it only prints to screen a message when it is executed. This example looks a bit basic, but if you have been following all the way, you can realize that triggering a scan is just calling the proper `Experiment` method.

Exercise

There are other signals for a `QPushButton`, for example `released` and `pressed`. Hook them up to your own methods and play around. When is the `released` signal triggered? And the `pressed`?

If you want to know which signals are available and which method does every QtWidget possess, you can refer to the Qt documentation. It is the official documentation and therefore the syntax is for C++, but you can easily adapt it to Python, especially if you follow the examples outlined in this book.

8.5 Hooking the Experiment to the GUI

It is time to start interacting with your device straight from the GUI. Create a new file called **start_gui.py** in the **Examples** folder just next to **start_cli.py**. You will put together everything what you already know. The idea behind having a class for the experiment is that you can instantiate and pass it to the GUI as arguments to the different windows. For example, this is how the **start_gui.py** will look like:

```python
import sys
from PyQt5.QtWidgets import QApplication

from PythonForTheLab.Model import Experiment
from PythonForTheLab.View.scan_window import ScanWindow

e = Experiment()
e.load_config('Config/experiment.yml')
e.load_daq()

ap = QApplication(sys.argv)
m = ScanWindow(e)
m.show()
ap.exit(ap.exec_())
```

This example is exactly a merge of both previous examples, the *CLI*, and the *GUI*. Pay special attention to the highlighted line. You are passing the experiment class to the `ScanWindow`, but `ScanWinow` class does not take arguments other than its parent. If you run that file now it will give an error, however, feel free to go ahead and try, you will see that the error is very descriptive. Note also that you are passing the Experiment already instantiated, with the configuration and the daq loaded. It is a matter of convenience, and not of necessity. You will see later that those two actions can also be triggered from within the GUI.

In order to accept the experiment as an argument, you need to update the code of the `ScanWindow` class.

```python
class ScanWindow(QtGui.QMainWindow):
    def __init__(self, experiment=None, parent=None):
        super().__init__(parent)
        self.experiment = experiment

    [...]
```

Note that you have specified a default value for the experiment variable: `None`. This trick is very important when you are refactoring a class or a function. Imagine you have a large project and you are using the `ScanWindow` in different locations, or even more importantly, your package is used by someone else. If you just add the experiment as an argument, what will happen is that any piece of code that looks like `ScanWindow()` will throw an error. This implies changing the code in every line that has used the class. If you add a default value, you guarantee that older code is still compatible. For larger changes, however, this may not be sustainable.

Warning

It is not possible to maintain backward compatibility always. In the example that you are developing, the experiment is going to become a very important piece. Older code that doesn't supply an experiment object will eventually lead to something broken downstream.

Exercise

Now the experiment class is stored as `self.experiment`; can you replace the method `clicked` in order to print the `idn` of the daq card?

Exercise

Replace the behavior of the start button. Trigger a scan with the parameters already stored in the experiment. What happens to the GUI if you do so? **warning**: I suggest you use parameters in the YAML file that make your total scanning time relatively short.

8.6 Threads to the Rescue

If you have finished the last exercise, you probably noticed that while the scan is running your GUI freezes. Depending on your operating system and the time it takes the scan to run, you will get a notification saying that your app does not answer and you will be asked if you would like to terminate it. It means that you need to find a way to prevent long-running tasks from interrupting the main loop of our Application. After all, you don't see Firefox freezing every time a page takes a bit longer to load, or your photo editing software when it exports the pictures with a new resolution.

The way in which Python programs can handle this is with the use of threads. When you run a script, everything is going to happen from top to bottom, one line at a time. If a function takes too long to execute it will be halted there. Remember that taking too long doesn't mean it is computer intensive work. For example, you can be waiting for user input, or perhaps you want to trigger something every 10 minutes and you put a `sleep` statement. Threads are a way of handling this behavior, by allowing several processes to run simultaneously. In our case, we want our scan experiment to run in a thread separated from the thread in which the main GUI is running.

Note

If you finished the last chapter, you have already learned how to work with threads in Python. Working with threads in Qt follows basically the same principles, but with the added benefits of signals and slots. It is, normally, a good idea not to mix Python threads and Qt threads when possible.

Working with threads in Qt is relatively easy, especially thanks to *Signals*. First, you need to create is a new class that you are going to call `WorkerThread`. Workers are going to run whatever method you give them to run but will do so in a separate thread. You should create a new file in the **View** folder and call it **general_worker.py**. The idea of calling it *general* is because it will run any function you want. Worker threads can also be specialized, meaning that they will take care, for example, of the acquisition of a camera, the input from a user, etc.

```
from PyQt5 import QtCore

class WorkThread(QtCore.QThread):
    def __init__(self,  function, *args, **kwargs):
        super().__init__()
        self.function = function
        self.args = args
        self.kwargs = kwargs

    def __del__(self):
        self.wait()

    def run(self):
        self.function(*self.args,**self.kwargs)
        return
```

When working with Qt threads, you need to reimplement some important methods and that is why this class may look particularly complicated. You start by creating a new class called `WorkerThread` that inherits the QThread class. The `__init__` method takes one required argument, called `function`, which is the function that the thread is going to execute. `*args` and `**kwargs` is just a way of accepting any number of variables. You don't know exactly how many arguments the function is going to take since it can *a-priori* be any function. You store all the information as properties of the class (`self.function`, etc.). The `__del__` method is there just to be sure that the Thread is properly closed if you decide to stop it before it finishes running.

The important part of the code is the `run` method. This method is going to be called from the main part of the code when we want to execute it. In this specific case, you just execute the function that you passed at the beginning while instantiating the class. Notice that you are passing to the function the same arguments that the `__init__` took, namely `*args` and `**kwargs`. Once the class is ready, it is time to see how to use it from the main code. If you go back to the **scan_window.py**, you will need to update the code to run the thread properly. If you finished last section's exercise, you should have ended up with code like this:

```
def button_clicked(self):
    self.experiment.do_scan()
```

Since you are trying to do proper coding, you need to update the names of the methods for them to make sense. For example, you can have a new method called `start_scan` that will look like this:

```python
def start_scan(self):
    self.worker_thread = WorkThread(self.experiment.do_scan)
    self.worker_thread.finished.connect(self.worker_thread.deleteLater)
    self.worker_thread.start()
```

Remember to update the code where you are connecting the signal `clicked` of the start button to the method. First, note that you created a `worker_thread` as a property of the `ScanWindow` but was not defined within the `__init__` method, which is violating some design principles but you can still deal with it. The only argument that you are passing to the thread is the function that you want to execute, in this case, `do_scan`. Pay attention to the fact that you are passing the function without `()`. If you add the `()` you would pass the result of the function and thus you will have to wait until it finishes for creating the `worker\_thread`. Then you connect the signal `finished` from `worker_thread` to one of its own methods. It can be slightly convoluted, but this will ensure that when the thread finishes it is going to be cleared. If you run the same program for months you really don't want to start accumulating in memory millions of garbage threads. Finally, you `start` the thread. Pay special attention to the fact that you don't use the `run` method explicitly. You can go ahead and run the program again. Is it still freezing when you perform a scan?

Exercise

You have only the start button working. Hook the stop button to a new method `stop_scan`. To stop the execution of a thread you have to use the `quit()` method, for example: `self.worker_thread.quit()`.

Exercise

Thanks to what you have developed in the previous chapter, you can also stop the scan by changing the value of a property. Update the `ScanWindow` to use it.

Exercise

How can you prevent the program from trying to stop a thread that is not actually running? A great idea would be to add a property called `self.scan\_running` and set it to `False` in the `__init__`, set to `True` when you start the scan and back to `False` when the scan ends. Try to implement this in your code.

Exercise

Prevent the user from triggering two scans at the same time, for example, you can disable the start button by using the method `setEnabled(False)` or `setEnabled(True)`

8.7 Conclusions

This chapter may have been one of the more interesting and compelling chapters in the book. You have built a user interface and you have hooked an experiment to it. The GUI is far from complete, but nevertheless, it is a great achievement what you have developed so far. With what you have done here you can already do plenty of things if you are creative enough. You can have a user interface for triggering actions on your computer, in your experiment, etc. The options are endless and you are only halfway through, so imagine what is going to happen next.

8.7.1 A Word on Qt

Qt was developed for C++ programmers and was later ported to Python. This means that a lot of the documentation and examples that you can find around are for another language and thus you will need to translate it into Python. If there is something you can't understand and there are no obvious answers online, ask in Python For The Lab, the community over there is very helpful and knowledgeable.

When developing programs, normally you read code from top to bottom, one line at a time. This is also what computers do when interpreting or compiling programs. When you start dealing with *user interfaces* it becomes almost impossible to read code from top to bottom. For example, you never know when a user will trigger an action, stop it, in which order, etc. That is why the last exercise of the previous example is very important. You should anticipate, at least to a certain extent, the mistakes that a user can commit.

Of course, there are also limits to how much you can anticipate. When developing programs for the lab, normally you assume that the users of your code are going to be aware of what they are doing and how the experiment is designed. However, more often than what you think, mistakes happen. Perhaps even yourself forget about a detail, or you are in a hurry to acquire the last few data points. Therefore, thinking ahead about common mistakes and displaying appropriate messages can save you and the users of your program a lot of time later on.

Getting to understand how Qt works and how you can do different things with it is a lengthy process. Every app that you see, not only on your computer but also on your mobile phone or embedded platform such as smart watches or thermostats can be built using Qt. It is completely up to you to put a limit on what you need to learn in order to perform an experiment. Do you want to make the interface prettier with icons? Do you want to verify the user input and display warning messages? Do you want to stream a camera signal? All those are possibilities few keyboard strokes away, but they are of course much more involved than what you have developed until now.

8.7.2 A Word on Signals

Working with *Signals* is incredibly handy, and once you get used to them you will want them to be available everywhere, especially when you are doing some threading in Python. However, signals make your code harder to follow for beginners. You connect your signal once and you forget about it. Every time a signal is fired, something will be triggered, such as a method or a function. Checking what is called and which arguments are being passed becomes more complicated. Even the best IDEs available for Python won't be able to track what is really happening in your code.

If you are developing code that you expect others to understand, you should either provide all the documentation they need or limit the number of nonordinary packages that you are employing. For example, if the users of your code are used to analyzing data, you can count on them knowing *numpy* and *matplotlib*, but most certainly they are not aware of *PyQt* and its special models such as *signals* and *slots*. If you are building code for others, be sure you use libraries common to them when possible, or document your progress to make it understandable.

Chapter 9

User Input and Making Plots

9.1 Objectives

In this chapter you are going to see how you can process user input for changing the parameters of your scan and how to plot the results while they are being generated. Dealing with user input can lead to unexpected errors that are sometimes hard to anticipate. Plotting the data while it is being acquired is a great feature, especially when you want to make quick decisions before an entire scan is done.

9.2 Introduction

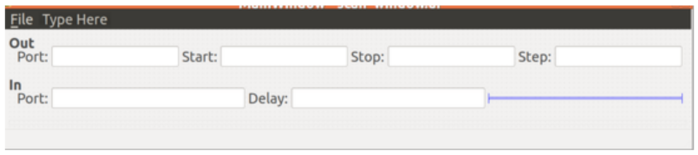
In the previous chapter you have already seen how to handle some user input: you have a start and a stop button able to trigger the scan. It is useful, but it doesn't really allow you to change the parameters of the scan. When you want to take input from a user, you will have to deal with the validation of the values. More often than not, a user will mistake the input port with the voltage range, or they will forget what the `delay` does. Sadly, this user can very well be yourself in the future.

Incorrect input data from a user can not only lead to the wrong experiment, it can also mean a big problem in your setup. Imagine that you set a voltage to a value higher than what your device can handle. Or you set a piezo stage to a position that will break your sample (a very, very common issue in microscopes). To avoid these issues, the program has to check that the input makes sense before being passed to a device, and this is called data validation.

Having a GUI that allows you to change the parameters of an experiment is very handy. Much more so is to add the possibility to monitor a signal while the experiment is running. Plotting in real time is not hard but it can be computationally expensive, depending on how many data points are you willing to show and how often do you want to update the plot. In this chapter, you will learn how to develop a program that takes into account these issues.

If you have experience analyzing data with Python, most likely you have come across *Matplotlib*, a great library for generating very professional plots. Even if it is a great tool for making figures, it is not the proper tool for real-time visualization. Updating Matplotlib plots is an intensive task for the computer and therefore it is not going to allow you to update the plots as often as you would like. That is why, for data visualization, you are going to use another, perhaps a lesser-known library, called PyQtGraph which integrates everything that you need for plotting within user interfaces.

9.3 Getting User Input



In your GUI, you have added some parameters that the user can modify before triggering a scan, namely, those are the output port, the start, stop, and the step of the voltage, the input port, and the delay between measurements. You need to read the values and update them into the experiment class before triggering the `do_scan` method. If you remember from the command line example, updating the values of the scan is as easy as doing

```
experiment.properties['Scan']['start'] = new_start
```

In the GUI it is going to be the same principle. If you want to read from a `QLineEdit` (the name of the widget that holds the information) you just need to do the following:

```
self.outPortLine.text()
```

This command will give us access to whatever is written in the `outPortLine`. Bear in mind that you can not only read from the line, you can also set the text that you want to display, like this:

```python
self.outPortLine.setText("Text to display")
```

Exercise

Now that you know how to read and set the values of a `QLineEdit`, update the `__init__` method of the `ScanWindow` in order to set the initial values of the scan, which are available within `experiment.properties['Scan']`.

To allow the user to change the parameters, you need to read them and update the experiment class. By now you should have a broad idea on how to achieve this, but it is important to point out that the experiment class relies on receiving values with units, not just numbers in the appropriate parameters. In the chapter when you defined the experiment class you also impose some restrictions on how the parameters should be defined. If you go back to the class, you will see that the `do_scan` method transforms strings into quantities. Therefore, the parameters for the scan should be strings with the appropriate units, and the experiment class will take care of the conversion.

```python
self.experiment.properties['Scan'].update({
        'port_out': int(self.outPortLine.text()),
        'start': self.outStartLine.text(),
        'stop': self.outStopLine.text(),
        'step': self.outStepLine.text(),
        'port_in': int(self.inPortLine.text()),
        'delay': self.inDelayLine.text(),
    })
```

Note that you are using a special method of dictionaries called `update`, that allows you to update a dictionary with the values given by another dictionary. If it is unclear to you how does it work, you should just practice with an easier example, a dictionary with just a handful of keys and values. Remember that the experiment class takes strings for the properties with units, but an integer for the ports, that is why we transform the input and output ports to `int`. It is as easy as that, few lines of code, little complication and now you are getting user input into consideration.

When you start playing around with your program, and especially when you give it to other users to work with, a lot of problems are going to arise. One of the most common ones is people forgetting to put units into their values. You can build your entire software just with plain numbers, but in the long term, this is very unreliable. Soon enough you will forget if the numbers you were supposed to use were in centimeters, millimeters, etc. Go ahead and try to see what happens if you don't include units, or if you try to start the scan when some of the values are left blank.

Validating user input is a complicated task; at some point, especially for software that is supposed to be used by smart people with good intentions, you have just to let it be. For example, you can check that the output port is a valid option if you have only two outputs it should be either 1 or 2. But you should also check that the values in every option are the ones that the device can handle, etc. And then, imagine they are wrong, what do you do? Do you create a new window with an error message? It is up to you and your use case to judge how much involvement is needed.

Exercise

To validate user input, you can also use signals. `QLineEdit` has a signal called `editingFinished`, that is triggered whenever you stop writing and move away from the line. Hook that signal from whatever input you like to a new method that will verify that the values are correct (for example, that the output port is either 1 or 2). If the input is not valid, disable the `startButton` by doing `self.startButton.setEnabled(False)`, or the opposite if the value is correct.

9.4 Plotting the results

Plotting the results while you are acquiring the data is the last step to have a fully functional program for performing experiments. You may remember (or you should refresh it if you don't) that the `do_scan` accumulates the data in a variable called `ydata_scan`. It means that each data point is available as soon as it is acquired. This is true for your very simple daq device, but it may not be exactly true for more complex acquisition cards. Anyways, let's see how to plot.

You may remember the PyQtGraph module that you used for plotting some data while we were looking at running the experiment from within the command line. You are also going to use it here, and then you are going to make a plot inside of the `ScanWindow`, let's see:

```
import pyqtgraph as pg

[...]
```

```python
class ScanWindow(QtGui.QMainWindow):
    def __init__(self, experiment, parent=None):
        super().__init__(parent)

        [...]

        self.main_plot = pg.PlotWidget()
        layout = self.centralwidget.layout()
        layout.addWidget(self.main_plot)
        self.ydata = np.zeros(0)
        self.xdata = np.zeros(0)
        self.p = self.main_plot.plot(self.xdata, self.ydata)
```

If you remember from a couple of chapters before, when plotting data you were also defining a `PlotWidget`, but at the time you had no user interface around it. In the previous chapter, you have seen that the building blocks of Qt are called Widgets, and in the snippet above you see that everything is coming in place together. The first step is to create the widget you want to use, in this case, it is called `self.main_plot`. Once you have the widget, you need to add it to the window. In the previous chapter, you have added buttons to a `ButtonBox`, but the plot should be added to the layout of the window. At this point is where some explanations regarding Qt are in order.

When you define windows in Qt Designer, some important things are happening under the hood. For example, when you define a `QMainWindow`, you should also define the central widget of it, which is like letting the window know which widget is the most important one. In the designer, this happens by default and perhaps you don't even notice that as soon as you create a window, there is also an empty `centralwidget` into which you put all the elements. When you use Python, you can reference the central widget by `self.centralwidget`. In the `ScanWindow` the `centralwidget` has a vertical layout associated with it, and that you can use by calling the method `layout()`. Note that not all widgets have a default layout defined, sometimes it can even happen that the layout is a part of the widget, such as any other button, line, etc.

Once you have access to the layout, you add the plot widget to it. Because the `centralwidget` has a default vertical layout, the plot will be appended at the bottom. If it would have been a horizontal layout, it would have been appended to the right, etc. Qt is very flexible when it comes to specifying where and how widgets look like. Remember that everything that you have defined in the Designer can be later changed programmatically. Normally, it is better to have a good starting point and change the look of your window as little as possible. After you have added the plot to the layout, you can initialize it with empty data. In this case, you use two zeros and plot them. If you remember from a couple of chapters earlier, the output of the `plot` method will allow you to personalize the plot, adding labels, units or new data. That is why you store it as `self.p`. You can go ahead now and run the program, you should see a black plot within your window. Well done!

There is only one more detail to mention, mainly for people who are going to work with complex plots and fast refresh rates. The `plot` method in PyQtGraph triggers a lot of actions under the hood that will be responsible for drawing the lines of your plot. It is, therefore, an *expensive* function to run, and it is not recommended to do it repeatedly. Therefore, you can plot empty data, or as in this case just one point with coordinates (0,0) and later you update the contents of your plot, without recreating it. Of course, you can avoid taking things for granted and try to plot over and over again and see if this discussion was a Premature Optimization.

Exercise

If you followed the previous chapters, you should have used PyQtGraph to plot the results of your scan from within the command line. Use the same examples to set X and Y labels for the plot.

The previous exercise is a just a trick for you to think ahead. With what you have done so far, you cannot set the axis, because you don't know which port you are going to scan and which one you are going to read. You need to wait until the user decides to trigger a scan to update the labels of the plot. So, let's first solve this problem. The method you would like to update is `start_scan`. To add the proper labels to the plot, you can do the following:

```
xlabel = self.experiment.properties['Scan']['port_out']
units = self.experiment.properties['Scan']['start'].u
ylabel = self.experiment.properties['Scan']['port_in']

self.main_plot.setLabel('bottom', 'Port: {}'.format(xlabel),
                    units=units)
self.main_plot.setLabel('left', 'Port: {}'.format(ylabel), units='V')
```

Remember to set the labels only after you have updated the properties of the experiment in the method. If you do it the other way around, you will be using the old values. You are very close to having a fully working GUI, but you are just missing one last thing. If you trigger the scan now, you will have nicely set up the labels, but you are not plotting the data yet. If you followed the example on how to update the plot when working from the command line, you will see that updating the GUI is just slightly adapting the code.

9.4.1 Words of caution when refreshing GUIs

When dealing with GUIs and experiments you have to take into consideration several properties, not only of the experiment but also of your own computer. Computer screens normally refresh at a maximum rate of 60 frames per second (fps). However, your eyes most likely will not be able to tell the difference beyond 30fps. 30fps is equivalent to updating every 33ms, while many experiments can acquire data at much higher rates than that. Even 60fps, which are equivalent to a refresh time of 16ms, can be longer than what your experiment is handling. It is important therefore to decouple the experiment refresh rate from the update process. If you try to update your plots at higher frame rates, depending on the complexity of your data, at some point the GUI will start lagging behind.

The same discussion can be given with the number of pixels that you are displaying. Imagine you acquire a signal with a time resolution of 1 second over an hour. You will end up with 3600 data points. A 4K display has roughly 4000 horizontal pixels. This means that you have almost as many data points as total pixels on the screen. If you plot your data, for example, in a window that takes half the screen size, you are going to lose information. When you try to plot more data points that pixels, the plotting library will take care of the reduction. This adds to the computational cost of plotting, plus you are going to miss the fine resolution you were after. When dealing with GUIs at some point you have to ask yourself what is the point of acquiring with a resolution that you are not able to see.

This discussion, by no means, implies that your acquisition cannot be faster or with a higher resolution. For example, you can acquire data with a fast CCD at thousand frames per second. However, if you display all the frames to the user or only one in every 30, the user will not be able to tell any difference. While you display a subset of the data, you can save the rest for later analysis. Confocal images, for example, are built by scanning the position of the sample (or of the beam) and can generate images with many more pixels than cameras and than screens. You can decide to plot only a region of the data or decide how to smooth it in order to preserve the features that you are after.

9.4.2 Decoupling acquisition and refreshing

Enough of a theoretical discussion, now it is time to implement in your code the decoupling between the acquisition and the update of the plot. You need to define a refresh time somewhere, and since it is a parameter that you may be interested in changing you can add it to a configuration file. In the `YAML` file where you defined the experiment, you can add a `refresh_time` option. If you have downloaded the YAML file from the repository, you may have already this parameter present. So far, the GUI is able to trigger a scan in a separate thread, you need to add another method that updates the plot periodically. If you want to trigger a periodic action in Qt, you can use a widget called `QTimer`. This widget will emit a signal periodically, after a defined period of time. This signal can be hooked to any method or function, as you have seen in the previous chapter with buttons. You have to start by creating the timer within the `__init__` method:

```
self.update_timer = QtCore.QTimer()
self.update_timer.timeout.connect(self.update_scan)
```

In the code above, you can see that the time was not set yet and the timer was not started. You have just created the timer and hooked its signal `timeout` to a method called `update_scan` that still has to be defined. Before going to that method, you also have to update the `start_scan` method to trigger the update_timer with the proper delay time. You can add the following at the end of the method:

```
refresh_time = Q_(self.experiment.properties['Scan']['refresh_time'])
self.update_timer.start(refresh_time.m_as('ms'))
```

The method to trigger the timer is `start` and takes only one argument: the delay time in milliseconds. This is one of the advantages of using quantities in python, you can forget about the details of every method you use. You convert the value given from the experiment class into `ms`, in the same way in which you have converted a time to seconds in order to use it with the Python `sleep` function. Sometimes you can pass a Quantity to a method, but it is not safe to assume that the method will know how to handle them. Every time you can, you should explicitly do it as above. The missing piece of code is the method to update the plot while the experiment is running. Remember that the scan has its own thread, but the values are stored in the experiment class, which is accessible from the main thread. The `update_scan` method will look like this:

```python
def update_scan(self):
    self.xdata = self.experiment.xdata_scan
    self.ydata = self.experiment.ydata_scan

    self.p.setData(self.xdata, self.ydata)

    if not self.experiment.running_scan:
        self.stop_scan()
```

First, you get the data from the experiment and copy it to class variables. Doing this is not mandatory, because it is only duplicating the information, but it is a strategy that can be very useful when you are getting data from a buffer. Buffers have a limited amount of memory and you empty them by reading the available values. In this case, it would mean that the data is not being stored in the experiment class, but directly in the user interface class. It is important to note the highlighted line. You are not using the `plot` method again, you are just updating the data from the plot. This is much more efficient, as was discussed earlier.

Exercise

Change the `setData` method for `self.main_plot.plot(...)`. Is the difference between one and the other appreciable?

Note that at the end of the `update_scan` method, you are also checking if the experiment is actually running or not. If the scan has finished, then you are going to trigger the `stop_scan` method. You need to do this to enable the user to trigger a new scan. Remember that you are checking the value of `experiment.running_scan` periodically. Once the scan is over, you also want to keep refreshing the plot with the same data. Therefore, you will need to update the `stop_scan` method to stop the timer:

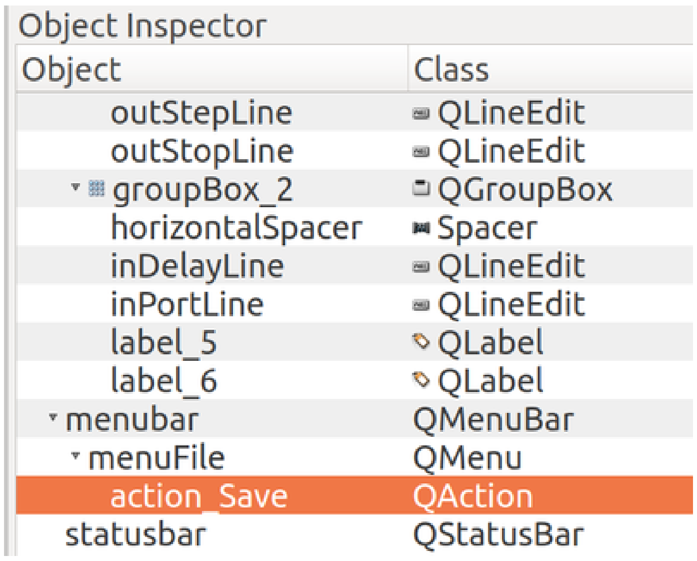
```python
self.update_timer.stop()
```

9.5 Saving Data

What you have now is a fully functional scan program, but there is an important feature missing. If you trigger a scan, you will have the data confined to your own program and you will not be able to analyze it afterward. When you triggered a scan from the command line, you have also saved the data to a file, and you should be able to do the same from within the GUI.

Exercise

Add a button to the GUI that says `Save Data`. Connect the button to a new method that saves the data to a file, in a similar way to what was done through the command line in a previous chapter.

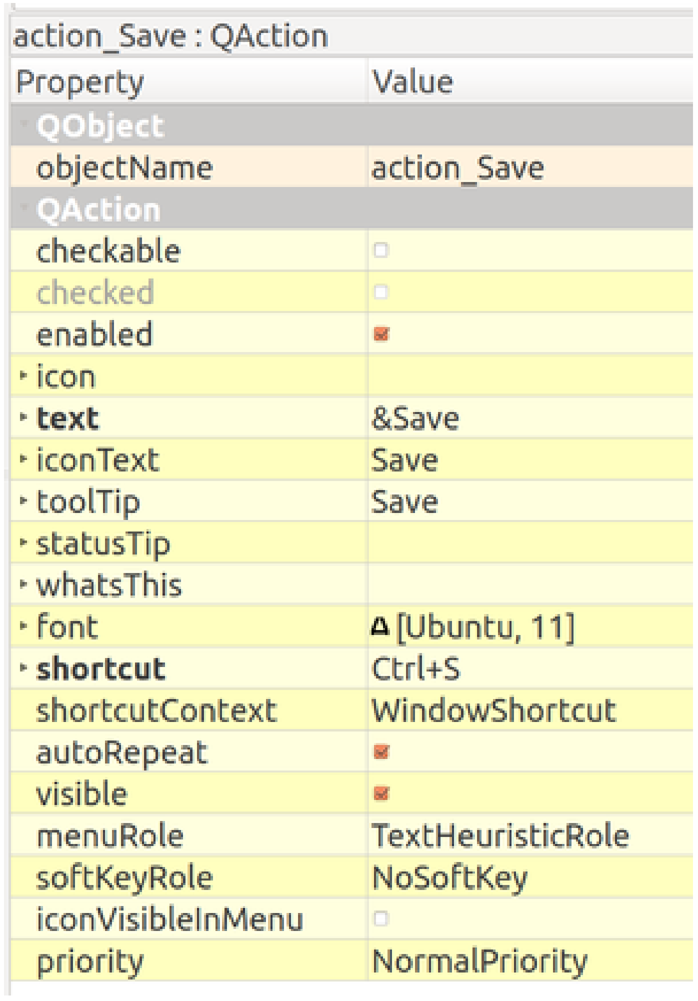


Object Inspector

Object	Class
outStepLine	QLineEdit
outStopLine	QLineEdit
groupBox_2	QGroupBox
horizontalSpacer	Spacer
inDelayLine	QLineEdit
inPortLine	QLineEdit
label_5	QLabel
label_6	QLabel
menubar	QMenuBar
menuFile	QMenu
action_Save	QAction
statusbar	QStatusBar

If you grabbed the **scan_window.ui** from the Github repository you probably found that there is a menu at the top, called `File` with a `Save` option. Having a File menu is the normal behavior of most programs in most systems, making it an intuitive place to find such an option. Working with the Menu is almost as easy as working with a button, you just need to connect the appropriate signal to the appropriate method in the class. If you open the design file in the Qt Designer, you will find an element called `action_Save` that appears as a `QtAction` instead of a button. Details are not important, you should just know that inside of menus you will find actions and not buttons. However, for the purposes of the program, they work in pretty much the same way. You have to update the `__init__` method to hook the proper signal:

```
self.action_Save.triggered.connect(self.save_data)
```


action_Save : QAction

Property	Value
QObject	
objectName	action_Save
QAction	
checkable	☐
checked	☐
enabled	✓
icon	
text	&Save
iconText	Save
toolTip	Save
statusTip	
whatsThis	
font	[Ubuntu, 11]
shortcut	Ctrl+S
shortcutContext	WindowShortcut
autoRepeat	✓
visible	✓
menuRole	TextHeuristicRole
softKeyRole	NoSoftKey
iconVisibleInMenu	☐
priority	NormalPriority

As easy as that. If you followed the previous exercise, you should have already defined a method for saving data, so you can just use it. As an extra, bear in mind two more things. When you define the name for the menus and the actions you are going to place inside, you can use the special character `&` to make it possible to select that menu by pressing Alt plus the letter the immediately follows. Therefore, if you call the menu `&File`, when you press Alt+F the menu will be opened. The same with the action, that is called `&Save`. Therefore, Alt+F opens the File menu, and if you press S it will save the data. This is handy to go quickly through more complex menus without using the mouse. You can also define a shortcut for your action, and you do so directly from within the designer, as is shown in the image. Whenever you press Ctrl+S you are going to save the data.

Saving data as you have done up to now is handy but lacks some functionality. For example, you are never asking the user where to save the data. If you are the only user of your program and you have a folder where you store all the data, it is a safe bet to default to that folder. If you are planning to have more users, it is imperative that you give the users the option to choose. Fortunately, it is incredibly easy, you can do so with only one line of code:

```
self.directory = str(QtWidgets.QFileDialog.getExistingDirectory(
                    self, "Select Directory", self.directory))
```

This will open a dialog where the user can select the directory where to save the data. For it to work, remember to initialize the variable `self.directory` within the `__init__` method. If you initialize it to an existing directory (for example `C:\textbackslash{}\textbackslash{}Data`), the dialog will start in that specific place, perhaps saving some searching time to the user. If not, just initialize it to `None` and the dialog will start in your current directory. Now that you have a directory, don't forget to use what you have already learned, i.e., don't forget to use `os.path.join`. Also, next time you save data, you will start in the same directory than last time (`self.directory`).

Exercise

Now every time the user wants to save data, a dialog appears asking for the directory. Find a way to ask for the directory only once and assume that the next time the file will be saved in the same directory.

It is also important to be able to add elements to the menu from Python. Sometimes you are using a designer file that you don't want to modify, because it is shared amongst different developers, but you still want to personalize your own program. Following the examples below, you will be able to create a new menu called `Scan`, next to the `File` menu, with two actions, one for starting and one for stopping a scan. First, you can create a new menu by doing to following:

```
menubar = self.menuBar()
self.scanMenu = menubar.addMenu('&Scan')
```

The first line is responsible for addressing the menu bar of the main window. It follows the same pattern as for when you used the layout of the central widget. The second line adds a new menu to the menu bar, called `scanMenu` and will display the text `Scan`. Again, the `&S` means that you will be able to open the menu by pressing Alt+S. Next, you have to define the proper actions to add to the menu:

```
self.start_scan_action = QtWidgets.QAction("Start Scan", self)
self.start_scan_action.setShortcut('Ctrl+Shift+S')
self.start_scan_action.setStatusTip('Start the scan')
self.start_scan_action.triggered.connect(self.start_scan)
```

In the first line, you create the action and you set its name to `'Start Scan'`. The `self` is to explicitly set the parent for the action, in your case it is the `ScanWindow`. Then you set the shortcut for the action, in this case, Ctrl+Shift+S. The status tip is something you haven't use so far, but it will help the user to understand what will happen if the action is triggered. If you look in your program, you will see that the message is being displayed at the bottom of the screen. The last line connects what happens when the action is triggered, in your case, a scan is started. The only missing thing is to add the action to the menu, like this:

```
self.scanMenu.addAction(self.start_scan_action)
```

Warning

Adding keyboard shortcuts is a great way of shortening both the development time and the interaction with the interface. It is much faster pressing Ctrl+S than going with the mouse to the menu, etc. However, it is important to document every shortcut that you add to your program. Some are intuitive and the user may try them without thinking, like the saving. However, imagine that you create a shortcut Alt+S to start a scan, Alt+F to stop it, etc. You should seriously consider adding a separate text file to your project describing the available shortcuts.

Adding elements to the menu is very handy because it allows you to add a lot of functionality without cluttering the user interface. Menus can also be nested, you could have added the scan menu to the file menu, for example. Now that you are able to start the scan, you should also be able to stop it.

Exercise

Add a new entry to the menu that allows the user to stop the scan. Pay attention to the shortcut that you specify.

For simple programs, you can still work with buttons on the screen. For more complex programs, you may need to have a clear interface, perhaps just plotting data and extra options in the menu, that doesn't get in the way.

9.6 Quitting the program

Your program is very functional, you can acquire and save data, you check that the user does not trigger more than one scan at a time. However, there is a very important piece missing. When the user quits the program, the communication with the device is suddenly interrupted, which is not the desired behavior. To prevent this, you can add an extra method for quitting. You only need to overwrite a method inherited from `QMainWindow`:

```python
def closeEvent(self, event):
    super().closeEvent(event)
```

In this case, nothing is happening because you are just relaying the method to the parent class' method. However, you can add any code you want to execute before calling `super().closeEvent`. For example, you could finalize the communication with the device:

```python
def closeEvent(self, event):
    print('Closing the window')
    super().closeEvent(event)
```

At this point is where you realize that you haven't developed a way of finishing the communication with the device from the experiment class nor from the model. However, the driver indeed has a `finalize` method. Suddenly, you realize that you missed an important feature upstream. You should be aware that in the design pattern that you are using, from the View you should interact only with the experiment model, and not directly with the driver. Therefore, to add this new functionality you need to solve the following exercises in order:

Exercise

Add a method `finalize` to the base DAQ model.

Exercise

Reimplement the method `finalize` in the `AnalogDaq` model, using the finalize method from the driver.

Exercise

Create a method in the experiment model that handles what happens when the experiment is over. For example, the communication with the DAQ should be ended with the `finalize` method. Perhaps you want to add something else, such as forcing the save of the parameter metadata or the data itself.

Exercise

Update the method `closeEvent` in the `ScanWindow` in order to use the finalizing method of your experiment class.

9.7 Conclusions

This is the most rewarding chapter of the book and, alas, the last one. You have made the jump from a working GUI to a usable interface. It is very important to point out that many of the things you have developed in this chapter were so easy because of how you structured the different classes in the previous chapter. For example, the method `do\_scan` in the experiment class was developed in such a way that it makes it very easy to add and trigger it from the GUI.

If you start a project from scratch, you will be tempted to skip some steps in order to have results faster. Trying to reach your objectives earlier is very valid, and there are many scenarios where there will be no further development of your code. In those cases, take the shortcut and deliver results as fast as you can. However, sometimes you know you are building something to last. In those cases do not jump ahead. Take the time you need to reflect on how do you see the future of the code you are writing. There are going to be different things to polish here and there, but if the basis is well developed, you are going to save a lot of time when trying to add new functionalities.

In the case of code that you thought it was never going to grow and suddenly you start using it very often, don't be afraid of doing a refactoring and transforming older scripts into classes, modules, etc. Remember always a general rule of thumb: if you are copy/pasting lines of code more than twice, then you should have done something differently. You could have written a function that takes care of your process, or a class, etc.

In this chapter, you have achieved something that very few people can say they can do. You have built an interface for a real-world experiment. You can acquire a signal, change the output, change the input, change the ranges. You can also save the data and the metadata. However, there is still much more that can be improved and developed. Of course, in the real world, once you have a working application, you can add new options when the need arises or when you have some free time.

9.8 Where to go next

At this point, you have acquired a lot of knowledge that you can put into action. The first logical step is to test what you have learned from your own experiment. You can go through the steps of the book, adapting each piece of code to your needs. Don't be afraid of reusing the pieces of code such as windows and models that can be useful for your project.

If you are willing to continue expanding the options of the GUI that you have just built, here are some ideas. First, you can build a monitor for the signal. A monitor is a plot that constantly updates the value measured, while the x-axis shows the time. You should configure some parameters, such as the refresh rate, the delay between acquisitions, and the total time to acquire the signal. You can also create a new window that holds all the configuration parameters present in the YAML file. For example, the current GUI doesn't allow you to change the current user.

If you are interested in a more fundamental project, you can think about how to deal with sensors connected to a DAQ. There is a common pattern, in which a voltage is converted into another quantity. For example, an analog output connected to a piezo stage would transform voltages into distances. In the example built in this book, you are measuring a voltage, which can be converted to a current if you know the resistance connected. Dealing with calibrations, sensors, and actuators can be achieved through new yaml files and models for the experiment.

Of course, there are many more topics that can be useful for different researchers. If you have any suggestions, doubts or have found mistakes please contact us, we are always eager to hear from our readers.

Appendices

Appendix A

Python For The Lab DAQ Device Manual

We believe that learning how to program software for a scientific laboratory can be achieved only through real-world examples. That is why the course was conceived around a small device that we are calling a General DAQ Device. In these pages, we document the behavior of the device, in a similar way to what you would normally find in the manual of any instrument in your own lab.

A.1 Capabilities

The **General DAQ Device** is a multi-purpose acquisition card that can handle digital and analog inputs and outputs. It runs on an ARM 32-bit microprocessor and drives its power from a USB connection. The device can handle one task per turn, but the outputs are persistent. This means that if you set the output of a particular port, it will remain constant until a command for changing it is issued.

Normal Analog-to-Digital conversion times are in the order of 10 microseconds and are done with a resolution of 10 bits in the range 0-3.3V. Digital to Analog conversions are done with a resolution of 12 bits in the range 0-3.3V.

The DAQ possesses 2 Analog Output channels and 10 Analog Input channels. Each can be addressed independently, however, a degree of crosstalk can be observed, especially between neighboring ports. Sensitive applications would, therefore, need to use non-consecutive ports in order to mitigate this effect.

A.2 Communication with a computer

The DAQ is able to communicate with the computer through a USB connection. However, the device has an onboard chip that converts the communication into serial, therefore it will appear listed as any other serial device.

The baud rate has to be set to 115200, and every command has to finish with the newline ASCII character. The messages generated by the device are also terminated by a newline character.

A.3 List of Commands Available

IDN: Identifies the device; returns a string with information regarding the serial number and version of the firmware. *Returns*: String with information

OUT:: Command for setting the output of an analog channel. It takes as arguments the channel **CH:{}** and the value, **{<4}**. The value has to be in the range 0-4095, while the channel has to be either 0 or 1.

Example: OUT:CH0:1024

IN: Command for reading an analog input channel. It takes one argument, **CH:{}**, between 0 and 9. *Returns*: integer in the range 0-1024

Example: IN:CH5

DI: Command for identifying the device. It blinks a built-in LED 10 times. It has no return. While the LED is blinking, the device does not accept commands. When used, an appropriate delay should be applied in the program. In total it takes around 3 seconds for the blinking sequence to complete.

Example: DI

Appendix B

Review of Basic Operations with Python

B.1 Chapter Objectives

The objective of **Python in the Lab** is to bring a developer from knowing the basics of programming to being able to develop software for controlling a complex setup. However, not all programmers have the same background and it is important to establish a common ground from which to start.

This chapter will quickly review how to start Python and how to interact with it directly from the command line. It will also review some common data structures such as lists and dictionaries. It will quickly go through for loops and conditionals. If you are already familiar with these concepts, you can safely skip this chapter.

B.2 The Interpreter

Python can be started from the command line by typing python and pressing Enter. This should start the Python interpreter and you should see something like this:

```
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.
>>>
```

You can type whatever you like into the interpreter. If it makes sense to Python, it will give you an appropriate answer. For example, you can do:

```
>>> 2+3
5
```

The first line is what you type (therefore it has the `<<<` at the beginning), while the second line is the output Python gives you (in this case, as expected, `5`). You can go ahead and play with different operations. You can multiply, subtract, divide, etc. Power of numbers can be achieved using `**`, for example, `2**.5` means the square root of 2.

B.3 Lists

From now on, the `<<<` will be suppressed in order to make it easier for you to copy the code. Python can also be used to achieve much more complicated tasks and with many different types of variables. For example, you can have a list and iterate over every element of it:

```
a = [1, 2, 3, 4]
for i in range(4):
    print(a[i])
```

As you can see, `a` is a list with 4 elements. You make a `for` loop over the four elements and you print them to screen. You should see an output like this:

```
1
2
3
4
```

Of course, you can argue that this is not handy if you don't know beforehand how many elements your list has. We can improve the code by doing it like this:

```
for i in range(len(a)):
    print(a[i])
```

There is an important point to note, especially for those who come from a Matlab kind of background. If you print the variable i inside the loop, you'll notice it starts in 0 and goes all the way to N-1. It means that the first element in a list is accessed by the index 0. Lists have another interesting behavior. The elements in them do not need to be of the same type. It is completely valid to do this:

```
a = [1, 'a', 1.1]
for i in range(len(a)):
    print(type(a[i]))
### Output ###
<class 'int'>
<class 'str'>
<class 'float'>
```

You can have lists with lists in them and many other combinations.

Exercise

Make a list in which each element is a list. Nesting two `for` loops, display all the elements of all the lists.

Lists can also be iterated over with a much simpler syntax, without the need of the index.

```
a = [1, 'a', 1.1]
for element in a:
    print(element)
```

There is also a very *pythonic* way of declaring lists with a very concise syntax:

```
a = [i for i in range(100)]
```

This will generate a list of all the numbers from 0 to 99. You can also calculate all the squares of those numbers with a small modification:

```
a = [i**2 for i in range(100)]
```

And you can make it even more complex, for example, if you want to get only the even numbers you can type:

```
a = [i for i in range(100) if i%2==0]
```

Exercise

Given a list like: `b = [1, 2, 'a', 3, 4, 'b', 5, 'c', 'd']`, create another list with only the elements of type string.

Lists are a fundamental Python structure and it is important to keep them in mind in order to follow the syntax of some programs without getting lost.

B.4 Dictionaries

Dictionaries are one of the most useful data structures of Python. They are somehow like lists, but instead of accessing them via a numerical index they are accessed via a string identifier. For example, you can generate a dictionary and access its values by doing:

```
a = {'first': 1, 'second': 2}
a['first']
```

Dictionaries, as lists, can store different types of variables in them. Pay attention to the definition and call: lists are defined using square brackets `[ ]`, while dictionaries are defined with curly brackets `{}`. However, for accessing an element the square brackets are used. It is possible therefore to do:

```
b = [1, 2, 3, '4', 5.1]
a = {'first': 1, 'second': b}
a['second']
```

The first notable advantage of using dictionaries is that it makes much clearer what data you are storing. You are giving a title to a specific value. If you want to calculate the area of a triangle:

```
t = {'base': 2, 'height': 1}
area = t['base']*t['height']/2
```

And you immediately see that even if you don't have the definition of `t`, it is very clear what you are doing. It is clearer than the following code:

```
area = t[0]*t[1]/2
```

In the case of a triangle, it doesn't really matter which element is the base and which one is the height. However, for more complex applications, altering the order can have very serious consequences. In the same fashion than with lists, it is possible to access every element within a for loop:

```
for key in a:
    print(key)
    print(a[key])
```

Now the key has a value that can be printed and used. We can also check if a specific key is present in the dictionary:

```
if 'first' in a:
    print('First is in a')
```

If you want to update several values of a dictionary, but not to replace the dictionary itself, you can use the command `update`:

```
a = {'first': 1, 'second': 2, 'third': 3}
new_values = {'first': 5, 'second': 6, 'fourth': 4}
a.update(new_values)
a['first']
a['third']
a['fourth']
```

If you pay attention you will see that not only the already existent values were updated, but a new one was created.

Exercise

Given two dictionaries,

```
a = {'first': 1, 'second': 2, 'third': 3}
```

and

```
b = {'fourth': 4, 'fifth': 5}
```

merge the second into the first one.

Of course, it is also possible to delete an element from a dictionary:

```
a = {'first': 1, 'second': 2, 'third': 3}
del a['first']
print(a['first'])
```

You'll see an error letting you know that the key `first` is not in the dictionary. So far we have always used `strings` for the keys of the dictionary, but nothing prevents you from using numbers. The following lines are perfectly valid:

```
a = {1: 2, 2:4, 3: 9}
b = {0.1: 2, 'a': 3, 1:1}
```

This, on one hand, makes dictionaries very versatile, on the other, it may make the code slightly more confusing. For example `a[1]` may be referring to either the second element of a list or the element of a dictionary with key `1`. At this point, you may wonder why you would use lists if dictionaries give you even more functionality. The short answer is memory usage; the code below will output the memory being used by a dictionary and by a list with the same information in them. The first line of the code is just importing the function we need for calculating the size of a variable.

```python
from sys import getsizeof

a = [i for i in range(100)]
b = {i:i for i in range(100)}

print(getsizeof(a))
print(getsizeof(b))
```

You should see that the size of `a` is `912\ bytes` while the size of the dictionary `b` is `4704\ bytes`. Even if you consider that the dictionary is storing not only the value but also the key, the ratio of memory usage of a dictionary to a list is more than twice.

Exercise

Write a simple for loop that prints the ratio of the memory usage of a list and of a dictionary as a function of the length of each.

Appendix C

Classes in Python

Python is an object-oriented programming (OOP) language. Object-oriented programming is a programming design that allows developers not only to define the type of data of a variable but also the operations that can act on that data. For example, a variable can be of type integer, float, string, etc. We know that we can multiply an integer to another, or divide a float by another, but that we cannot add an integer to a string. Objects allow programmers to define operations both between different objects as with themselves. For example, we can define an object `person`, add a birthday and have a function that returns the person's age.

At the beginning, it will not be clear why objects are useful, but over time it becomes impossible not to think with objects in mind. Python takes the objects ideas one step further and considers every variable an object. Even if you didn't realize, it is possible that you have already encountered some of these ideas when working with numpy arrays, for example. In this chapter, we are going to cover from the very basics of object design to slightly more advanced topics in which we can define a custom behavior for most of the common operations.

C.1 Defining a Class

Let's dive straight into how to work with classes in Python. Defining a class is as simple as doing:

```python
class Person():
    pass
```

When speaking it is very hard not to interchange the words `Class` and `Object`. The reality is that the difference between them is very subtle: an object is an instance of a class. This means that we will use classes when referring to the type of variable, while we will use object to the variable itself. It is going to become clearer later on.

In the example above, we've defined a class called `Person` that doesn't do anything (that is why it says `pass`.) We can add more functionality to this class by declaring a function that belongs to it. Create a file called **person.py** and add the following code to it:

```
class Person():
    def echo_name(self, name):
        return name
```

In Python, the functions that belong to classes are called **methods**. For using the class, we have to create a variable of type person. Back in the Python Interactive Console, you can, for example, do:

```
>>> from person import Person
>>> me = Person()
>>> me.echo_name("John Snow")
```

The first line imports the code into the interactive console. For this to work, it is important that you trigger python directly from the same folder where the file **person.py** is located. When you run the code above, you should see as output `John Snow`. There is also an important detail that was omitted this far, the presence of `self` in the declaration of the method. All the methods in python take a first input variable called self, referring to the class itself. For the time being, don't stress yourself about it, but bear in mind that when you define a new method, you should always include the `self`, but when calling the method you should never include it. You can also write methods that don't take any input, but still will have the `self` in them, for example:

```
def echo_True(self):
    return "True"
```

that can be used by doing:

```
>>> me.echo_True()
```

So far, defining a function within a class has no advantage at all. The main difference and the point where methods become handy is because they have access to all the information stored within the object itself. The `self` argument that we are passing as the first argument of the function is exactly that. For example, we can add the following two methods to our class Person:

```
def store_name(self, name):
    self.stored_name = name

def get_name(self):
    return self.stored_name
```

And then we can execute this:

```
>>> me = Person()
>>> me.store_name('John Snow')
>>> print(me.get_name())
>>> print(me.stored_name)
```

What you can see in this example is that the method `store_name` takes one argument, `name` and stores it into the class variable `stored_name`. As with methods, variables are called **properties** in the context of a class. The method `get_name` just returns the stored property. What we show in the last line is that we can access the property directly, without the need to call the `get_name` method. In the same way, we don't need to use the `store_name` method if we do:

```
>>> me.stored_name = 'Jane Doe'
>>> print(me.get_name())
```

One of the advantages of the attributes of classes is that they can be of any type, even other classes. Imagine that you have acquired a time trace of an analog sensor and you have also recorded the temperature of the room when the measurement started. You can easily store that information in an object:

```
measurement.temperature = '20 degrees'
measurement.timetrace = np.array([...])
```

What you have so far is a vague idea of how classes behave, and maybe you are starting to imagine some places where you can use a class to make your daily life easier and your code more reusable. However, this is just the tip of the iceberg. Classes are very powerful tools.

C.2 Initializing classes

Instantiating a class is the moment in which we call the class and pass it to a variable. In the previous example, the instantiation of the class happened at the line reading `me = Person()`. You may have noticed that the property `stored_name` does not exist in the object until we assign a value to it. This can give very serious headaches if someone calls the method `get_name` before actually having a name stored (you can give it a try to see what happens!) Therefore it is very useful to run a default method when the class is first called. This method is called `__init__`, and you can use it like this:

```
class Person():
    def __init__(self):
        self.stored_name = ""

    [...]
```

If you go ahead and run the `get_name` without actually storing a name beforehand, now there will be no error, just an empty string being returned. While initializing you can also force the execution of other methods, for example:

```
def __init__(self):
    self.store_name('')
```

Will have the same final effect. It is however common (and smart) practice, to declare all the variables of your class at the beginning, inside your `__init__`. In this way, you don't depend on specific methods being called to create the variables.

As with any other method, you can have an `__init__` method with more arguments than just `self`. For example you can define it like this:

```
def __init__(self, name):
    self.stored_name = name
```

Now the way you instantiate the class is different, you will have to do it like this:

```
me = Person('John Snow')
print(me.get_name())
```

When you do this, your previous code will stop working, because now you have to set the `name` explicitly. If there is any other code that does `Person()` will fail. The proper way of altering the functioning of a method is to add a default value in case no explicit value is passed. The `__init__` would become:

```
def __init__(self, name=''):
    self.stored_name = name
```

With this modification, if you don't explicitly specify a name when instantiating the class, it will default to `''`, i.e., an empty string.

Exercise

Improve the `get_name` method in order to print a warning message in case the name was not set

C.3 Defining class properties

So far, if you wanted to have properties available right after the instantiation of a class, you had to include them in the `__init__` method. However, this is not the only possibility. You can define properties that belong to the class itself. Doing it is as simple as declaring them before the `__init__` method. For example, we could do this:

```
class Person():
    birthday = '2010-10-10'
    def __init__(self, name=''):
        [...]
```

If you use the new `Person` class, you will have a property called `birthday` available, but with some interesting behavior. Let's see. First, let's start as always:

```
>>> from person import Person
>>> guy = Person('John Snow')
>>> print(guy.birthday)
2010-10-10
```

What you see above is that it doesn't matter if you define the birthday within the `__init__` method or before, when you instantiate the class, you access the property in the same way. The main difference is what happens before instantiating the class:

```
>>> from person import Person
>>> print(Person.birthday)
2010-10-10
>>> Person.birthday = '2011-11-11'
>>> new_guy = Person('Cersei Lannister')
>>> print(new_guy.birthday)
2011-11-11
```

What you can see in the code above is that you can access class properties before you instantiate anything. That is why they are class and not object properties. Subtleties apart, once you change the class property, in the example above the birthday, next time we create an object with that class it will receive the new property. In the beginning, it is hard to understand why it is useful, but one day you will need it and it will save you a lot of time.

C.4 Inheritance

One of the advantages of working with classes in Python is that it allows you to use the code from other developers and expand or change its behavior without modifying the original code. The best would be to see it in action. So far we have a class called `Person`, which is general but not too useful. Let's assume we want to define a new class, called `Teacher`, that has the same properties as a `Person` (i.e., name and birthday) plus it is able to teach a class. You can add the following code to the file **person.py**:

```
class Teacher(Person):
    def __init__(self, course):
        self.course = course

    def get_course(self):
        return self.course

    def set_course(self, new_course):
        self.course = new_course
```

Note that in the definition of the new `Teacher` class, we have added already `Person`. In Python jargon, this means that the class `Teacher` is a child of the class `Person`, or viceversa, that `Person` is the parent of `Teacher`. This is called **inheritance** and is not only very common in Python programs, it is one of the characteristics that makes Python so versatile. You can use the class `Teacher` in the same way as you have used the class `Person`:

```
>>> from person import Teacher
>>> me = Teacher('math')
>>> print(me.get_course)
math
>>> print(me.birthday)
2010-10-10
```

However, if you try to use the teacher's name it is going to fail:

```
>>> print(me.get_name())
[...]
AttributeError: 'Teacher' object has no attribute 'stored_name'
```

The reason behind this error is that `get_name` returns `stored_name` in the class Person. However, the property `stored_name` is created when running the `__init__` method of Person, which didn't happen. You could have changed the code above slightly to make it work:

```
>>> from person import Teacher
>>> me = Teacher('math')
>>> me.store_name('J.J.R.T.')
>>> print(me.get_course)
math
>>> print(me.get_name())
J.J.R.T.
```

However, there is also another approach to avoid the error. You could simply run the `__init__` method of the parent class (i.e. the base class), you need to add the follwing:

```
class Teacher(Person):
    def __init__(self, course):
        super().__init__()
        self.course = course
    [...]
```

When you use `super()`, you are going to have access directly to the class from which you are inheriting. In the example above, you explicitly called the `__init__` method of the parent class. If you try again to run the method `me.get_name()`, you will see that no error appears, but also that nothing is printed to screen. This is because you triggered the `super().__init__()` without any arguments and therefore the name defaulted to the empty string.

Exercise

Improve the `Teacher` class in order to be able to specify a name to it when instantiating, for example, you would like to do this: `Teacher('Red Sparrow', 'Math')`

C.5 Finer details of classes

With what you have learned up to here, you can achieve a lot of things, it is just a matter of thinking how to connect different methods when it is useful to inherit. Without doubts, it will help you to understand the code developed by others. There are, however, some details that are worth mentioning, because you can improve how your classes look and behave.

C.5.1 Printing objects

Let's see, for example, what happens if you print an object:

```
>>> from person import Person
>>> guy = Person('John Snow')
>>> print(guy)
<__main__.Student object at 0x7f0fcd52c7b8>
```

The output of printing `guy` is quite ugly and is not particularly useful. Fortunately, you can control what appears on the screen. You have to update the `Person` class. Add the following method to the end:

```
def __str__(self):
    return "Person class with name {}".format(self.stored_name)
```

If you run the code above, you will get the following:

```
>>> print(guy)
Person class with name John Snow
```

You can get very creative. It is also important to point out that the method `__str__` will be used also when you want to transform an object into a string, for example like this:

```
>>> class_str = str(guy)
>>> print(class_str)
Person class with name John Snow
```

Which also works if you do this:

```
>>> print('My class is {}.'.format(guy))
```

Something that is important to point out is that this method is inherited. Therefore, if you, instead of printing a `Person`, print a `Student`, you will see the same output, which may or may not be the desired behavior.

C.5.2 Defining complex properties

When you are developing complex classes, sometimes you would like to alter the behavior of assigning values to an attribute. For example, you would like to change the age of a person when you store the year of birth:

```
>>> person.year_of_birth = 1980
>>> print(person.age)
38
```

There is a way of doing this in Python which can be easily implemented even if you don't fully understand the syntax. Working again in the class `Person`, we can do the following:

```
class Person():
    def __init__(self, name=None):
        self.stored_name = name
        self._year_of_birth = 0
        self.age = 0

    @property
    def year_of_birth(self):
        return self._year_of_birth

    @year_of_birth.setter
    def year_of_birth(self, year)
        self.age = 2018 - year
        self._year_of_birth = year
```

Which can be used like this:

```
>>> from people import Person
>>> me = Person('Me')
>>> me.age
```

```
0
>>> me.year_of_birth = 1980
>>> me.age
32
```

What is happening is that Python gives you control over everything, including what does the `=` do when you assign a value to an attribute of a class. The first time you create a `@property`, you need to specify a function that returns a value. In the case above, we are returning `self._year_of_birth`. Just doing that will allow you to use `me.year_of_birth` as an attribute, but it will fail if you try to change its value. This is called a read-only property. If you are working in the lab, it is useful to define methods as read-only properties when you can't change the value. For example, a method for reading the serial number would be read-only.

If you want to change the value of a property, you have to define a new method. This method is going to be called a *setter*. That is why you can see the line `@year_of_birth.setter`. The method takes an argument that triggers two actions. On the one hand, it updates the age, on the other, it stores the year in an attribute. It takes a while to get used to, but it can be very handy. It takes a bit more of time to develop than with simple methods, but it simplifies a lot the rest of the programs that build upon the class.