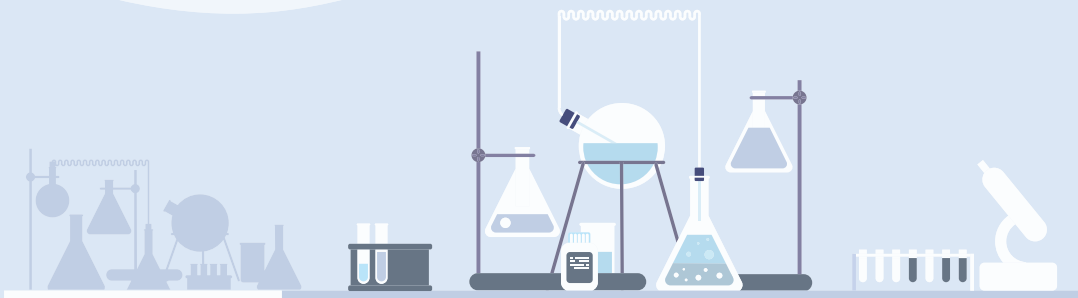
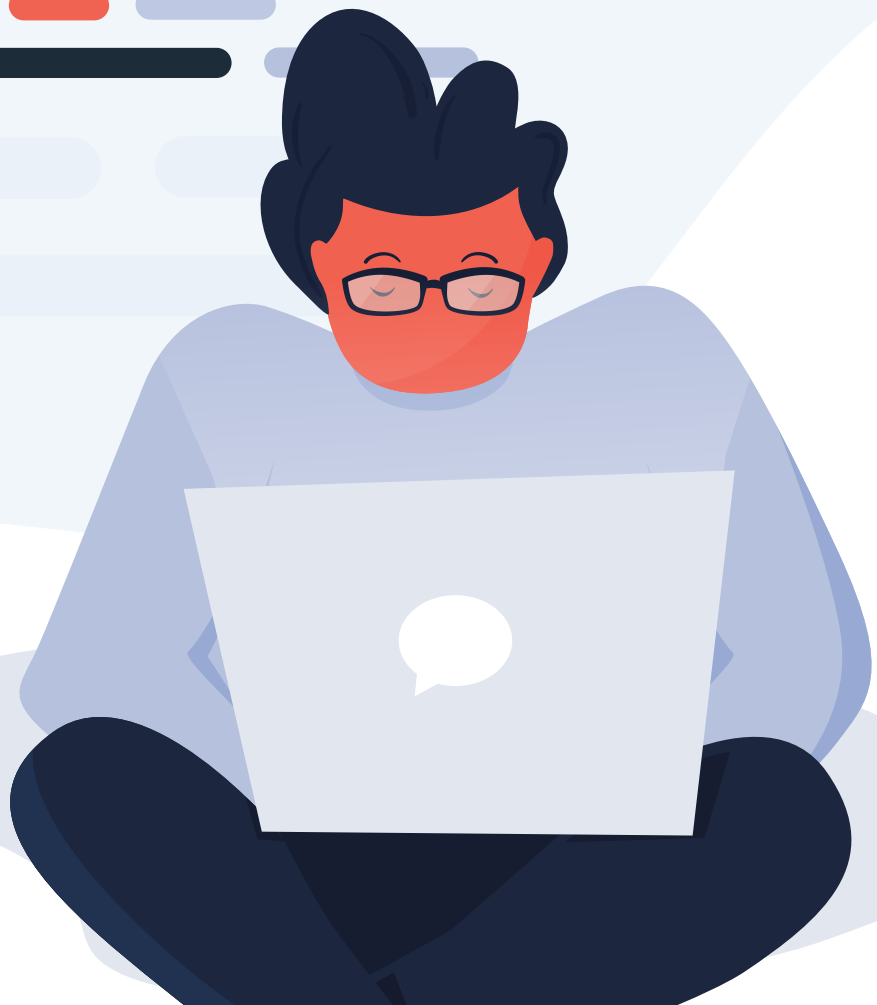
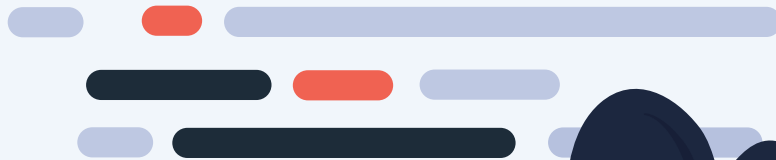


Dr. Aquiles Carattino



# Python *for the lab*



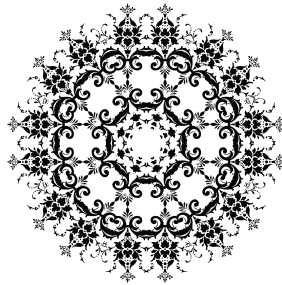
# **PYTHON FOR THE LAB**

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

**BY**

**AQUILES CARATTINO**

**PHD IN PHYSICS, SOFTWARE DEVELOPER**



**AMSTERDAM**

**Compiled on April 27, 2020**

# Contents

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

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

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



# Chapter 1

## Introduction

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

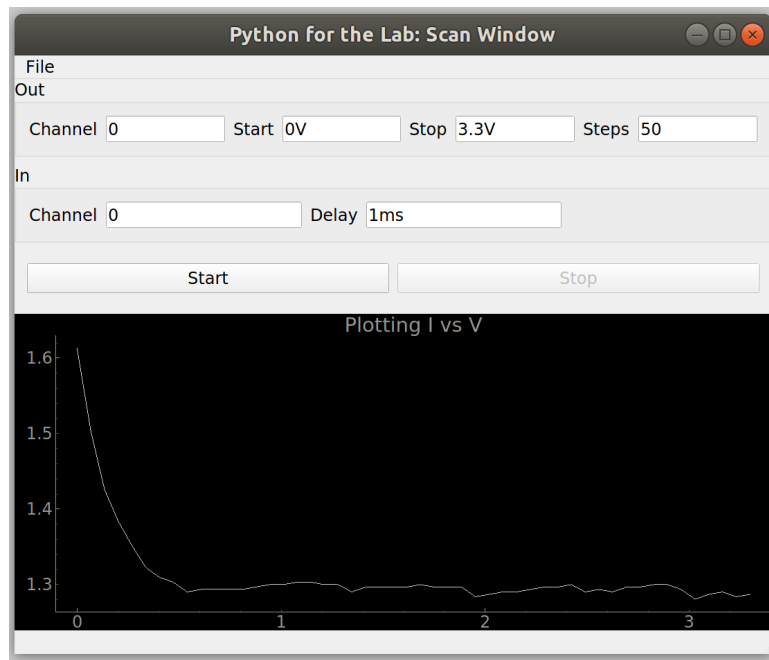
This book is task-oriented, meaning that it focusses on how things can be done and not on much theory on how programming works in general. This approach can lead to some generalizations that may not be correct in all scenarios. I ask your forgiveness in those cases and your cooperation: if you find anything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

Together with the book, there is a website<sup>1</sup> where you can find extra information, anecdotes, and examples that didn't fit in here. Remember that the website and its forum are the proper places to communicate with fellow Python For The Lab readers. If you are stuck with the exercises or you have questions that we don't answer in the book, don't hesitate to shout in the forum. Continuous feedback is the best way to improve this book.

---

<sup>1</sup><https://www.pythonforthelab.com>

## 1.1 What are you going to learn



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

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

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

## 1.2 Who Can Read this Book

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



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

## 1.3 Why building your software

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

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

## 1.4 PFTL DAQ Device

We have developed a device nicknamed PFTL DAQ that works as a data acquisition board. The instructor provides these boards during the workshops, but if you got this book online and would like to buy one of PFTL DAQ's, please contact us<sup>2</sup>. The devices are open source/open hardware, they are based on the Arduino DUE, and you can find the instructions for building one on our website. If you have access to any other acquisition card, with a bit of tinkering, you will be able to adapt the course contents to your needs.

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

## 1.5 Why Python?

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

---

<sup>2</sup>[courses@pythonforthelab.com](mailto:courses@pythonforthelab.com)

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

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

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

## 1.6 The Onion Principle

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

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

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

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

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

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

## 1.7 Where to get the code

The code that we develop through the book is freely available on Github<sup>3</sup>. The code is organized by chapters, to make it more accessible while reading. There is also an extra folder with a version of the program that goes beyond what the book covers. For example, the code includes documentation and an installation script. In this way, the readers can have an idea of the possible directions to take for their software.

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

## 1.8 Organizing a Python for the Lab Workshop

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

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

---

<sup>3</sup><https://github.com/PFTL/py4lab>



# Chapter 2

## Setting Up The Development Environment

### 2.1 Introduction

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

### 2.2 Python or Anaconda

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

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

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

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

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

## 2.3 Installing Anaconda

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

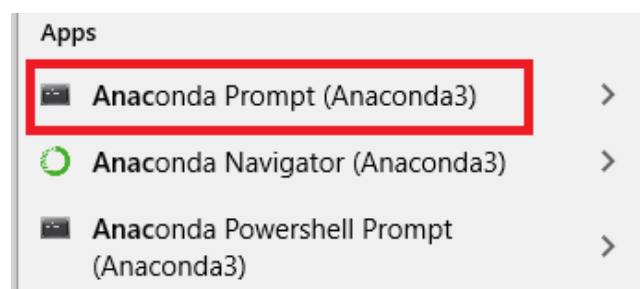


### Note

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

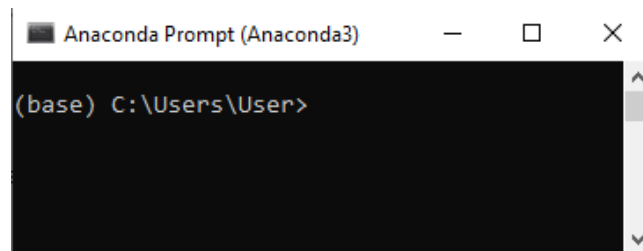
### 2.3.1 Using Anaconda

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



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

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



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

```
conda list
```

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

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

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

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

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

```
conda install pyserial
```

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

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

## 2.3.2 Conda Environments

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

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

And then we activate it:

```
conda activate myenv
```

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

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

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

```
conda install python=3.7
```

### Python Versions

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

After installing Python, you can start it by running:

```
python
```

And the output will be something like this:

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

To exit, just type:

```
exit()
```

To follow the book, you will need these packages:

- `numpy` -> For working with numerical arrays



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

Which we can install by running:

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

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

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

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

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

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

```
conda deactivate
```

## Quicker Environment Creation

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

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

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

## Remove an Environment

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

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

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

## 2.4 Installing Pure Python

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

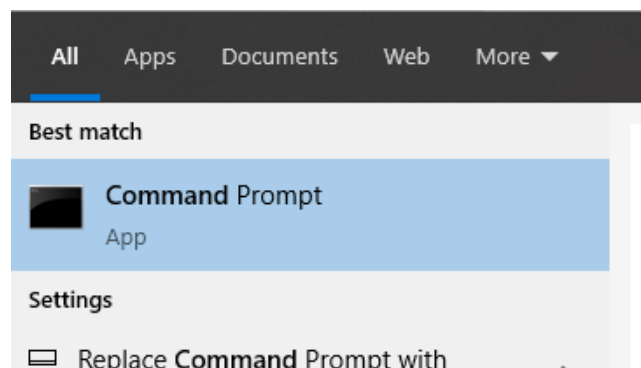
### 2.4.1 Python Installation on Windows

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

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

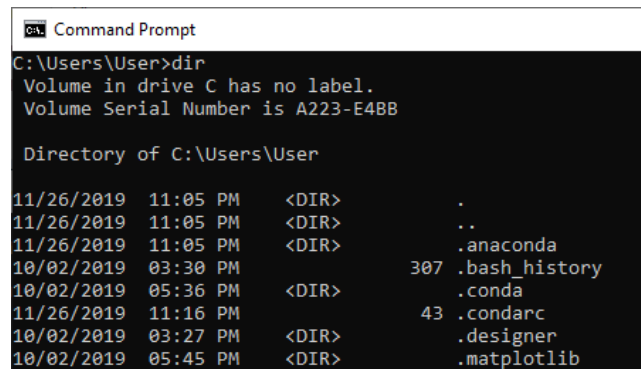
### Testing Your Installation

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



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

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

A screenshot of a Windows Command Prompt window. The title bar says "Command Prompt". The command prompt shows the current directory as C:\Users\User. It displays the output of the 'dir' command, showing the directory listing for C:\Users\User. The output includes the date and time for each file, the file type (e.g., <DIR> for directories), and the file name. The files listed are ., .., .anaconda, .bash\_history, .conda, .condarc, .designer, and .matplotlib.

```
Command Prompt
C:\Users\User>dir
Volume in drive C has no label.
Volume Serial Number is A223-E4BB

Directory of C:\Users\User

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

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

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

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

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

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

## 2.4.2 Adding Python to the PATH on Windows

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

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

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

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

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

### 2.4.3 Installation on Linux

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

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

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

```
sudo apt install python3
```

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

### 2.4.4 Installing Python Packages

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

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

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



### Info

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

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

```
pip install numpy
```

Windows users should instead type:

```
pip.exe install numpy
```



### Before Continuing

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

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

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

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

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

## 2.4.5 Virtual Environment

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

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

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

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

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

### Warning

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

## Virtual Environment on Windows

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

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

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

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

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

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

Once you have finished working in the environment, type:

```
deactivate
```

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

```
workon Testing
```

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

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

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

```
pip freeze
```

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

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



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

```
pip install virtualenvwrapper-powershell
```

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



```
Set-ExecutionPolicy RemoteSigned
```

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

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

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

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

```
.\activate
```

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

## Virtual Environment on Linux

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

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

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

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

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

```
which python3
```

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

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

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

To close the Virtual Environment you have to type:

```
deactivate
```

To work in the virtual environment again, just do:



```
workon Testing
```

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

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

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

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

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

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

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

```
pip freeze
```

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

## 2.5 Qt Designer

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

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

### 2.5.1 Installing on Windows

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

```
pip install pyqt5-tools
```

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

## 2.5.2 Installing on Linux

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

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

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

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

## 2.6 Editors

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

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

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

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

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



# Chapter 3

## Writing the First Driver

### 3.1 Objectives

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

### 3.2 Introduction

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

To have an idea of how commands look like, you can check the manuals of devices such as oscilloscopes or function generators. Both Tektronics<sup>1</sup> and Agilent have complete sets of instructions. If you search through their websites, you find plenty of examples. A command that you can send to a device may look like this:

```
*IDN?
```

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

The devices that exchange information with the computer in this way are called **message-based** devices. Some of this type of device are oscilloscopes, lasers, function generators, lock-ins, and many more. The PFTL DAQ device to work with this book also enters into this category. If you got the book online and not as part of a workshop, you can build your device or contact us, and we may be able to offer you one already programmed<sup>2</sup>.

---

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

<sup>2</sup>[courses@pythonforthelab.com](mailto:courses@pythonforthelab.com)



## Device Drivers

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

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

### 3.2.1 Scope of the Chapter

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

## 3.3 Communicating with the Device

To communicate with the PFTL DAQ<sup>3</sup> device, we are going to use a package called `PySerial`, which you should have already installed if you followed chapter 2. The first thing we can do is to list all the devices connected to the computer to identify the one in which we are interested. Plug your device via the USB port of your computer. We need to connect the PFTL DAQ device through the micro USB port closest to the power jack, also known as *programming port*. Then, the following command in a terminal (be sure you are within the environment in which you have installed the required packages):

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



### Warning

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

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

---

<sup>3</sup>PFTL is the shorthand notation for Python For the Lab

COM3

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

```
/dev/ttyACM0
```

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

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

```
python
```

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

```
>>> import serial
```



## Interpreter Characters

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

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

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

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

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

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

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

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

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

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

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

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

Many devices (but not all) follow a standard called SCPI<sup>4</sup>. The standard makes devices easy to exchange, because the commands for all oscilloscopes are the same, for all function generators are the same, and so forth. Moreover, the SCPI standard follows a structure that makes messages easy to understand and modular. A more powerful oscilloscope, for example, has commands not available to a more basic device, but the common features are controlled by the same messages.

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

### 3.3.1 Organizing Files and Folders

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

---

<sup>4</sup>[https://en.wikipedia.org/wiki/Standard\\_Commands\\_for\\_Programmable\\_Instruments](https://en.wikipedia.org/wiki/Standard_Commands_for_Programmable_Instruments)



## 3.4 Basic Python Script

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

```
import serial

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

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

device.close()
```

Now we can run the file:

```
python communicate_with_device.py
```



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

### What happens when you run the file?

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

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

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

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

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

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

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

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

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

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

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

```
import serial
from time import sleep

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

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

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

## Reading an Analog Value

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

```
import serial
from time import sleep

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

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

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

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

### ? Exercise

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

### ? Exercise

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

### ? Exercise

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

### ? Exercise

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

### ! Important note about ports

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

### ? Exercise

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

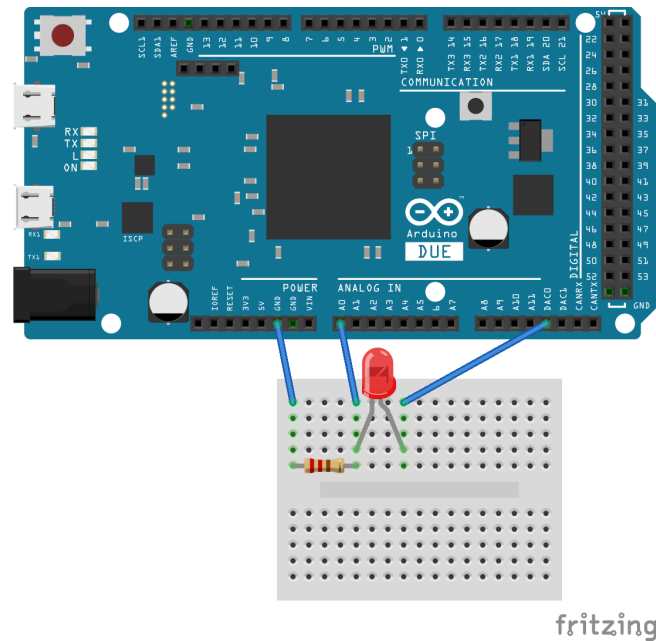


Figure 3.1: Schematic of the connections to perform the experiment

### 3.5 Preparing the Experiment

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

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

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

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

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

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

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

#### ? Exercise

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

## 3.6 Going Higher Level

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

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

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

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

```
import serial
from time import sleep

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

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

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

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

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

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

### ? Exercise

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

### ? Exercise

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

## Reading and Setting Values

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

### ? Exercise

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

### ? Exercise

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

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

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

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

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

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

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

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

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

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

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

And we can use it as follows:

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

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

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

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

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

## ? Exercise

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

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

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

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

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

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

## Proper Values Instead of Bytes

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

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

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



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

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

### ? Exercise

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

## 3.6.1 Abstracting Repetitive Patterns

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

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

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

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

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

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

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

    sleep(1)
```

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

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

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

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

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

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

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

And the same we can do for the other methods:

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

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

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

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

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

## ? Exercise

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

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

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

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

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

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

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

## 3.7 Doing something in the Real World

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

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

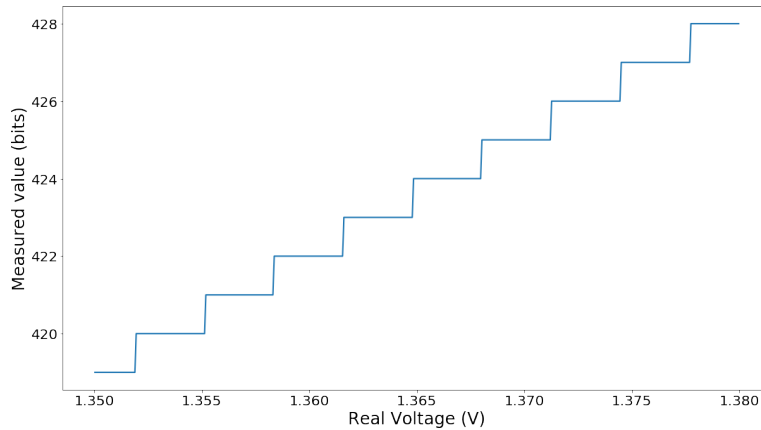
With this simple code, we can switch on and off the LED 10 times, and we print to screen the values that we are reading when it is on or off. There are two things to note: first, we are switching it ON by using a value of 4000. We have selected it because it is high enough to switch the LED on, but it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

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

### 3.7.1 Analog to Digital, Digital to Analog

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

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



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

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

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

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

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

### ? Exercise

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

## 3.8 Doing an experiment

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

### ? Exercise

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

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

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

### ? Exercise

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

### ? Exercise

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

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

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

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

## 3.9 Using PyVISA

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

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

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

```
pip install pyvisa
```

Or with `conda`:

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

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

```
pip install pyvisa-py
```

or with `conda`:

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

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

ourselves (and the same is true for USB, GPIB, or any other communication standard.) The documentation of pyvisa-py<sup>5</sup> has handy information.

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

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

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

Pay attention to the `query` method that we use to get the serial number from the device. We didn't develop it. PyVISA already took care of defining query for us. Not only PyVISA takes care of the query method, but they have plenty of options that we can use, such as transforming the output according to some rules or establishing the write termination. If we were to follow the pyVISA path, we could start by reading their documentation<sup>6</sup>.

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

## 3.10 Introducing Lantz

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

We may want to establish some limits, for example, to the analog output values. Imagine that we have a device that can handle up to 2.5 V. If we set the analog output to 3 V, we would burn it. Fortunately, there are some packages written especially to address this kind of problem. We are going to mention only one because it is a project with which we collaborate: Lantz<sup>7</sup>. You can install it by running:

---

<sup>5</sup><https://pyvisa-py.readthedocs.io>

<sup>6</sup><https://pyvisa.readthedocs.io>

<sup>7</sup><https://github.com/lantzproject>



```
pip install lantzdev
```

## About Lantz

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

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

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

```
from time import sleep

from lantz import MessageBasedDriver, Feat

class MyDevice(MessageBasedDriver):

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

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

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

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

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

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

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

## ? Exercise

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

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

```
output0 = None

[...]

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

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

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

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

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

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

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

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

```
_output = [None, None]

[...]

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

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

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

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

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

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

### Exercise

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

## 3.11 Conclusions

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

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

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

## 3.12 Addendum 1: Unicode Encoding

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

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

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

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

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

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

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

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

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

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

```
var_ñ = 1
```

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



# Chapter 4

## Model-View-Controller for Science

### 4.1 Introduction

Developing a computer program is much more than having a few scripts that we can run when we need them. The software should take care of a lot of concerns, such as the limits of the devices, should be flexible enough that enables you to change what measurements you are doing without spending months. More importantly, a program should be extensible in the long run, not just by yourself but also by future colleagues and, potentially, by anyone who finds your application online.

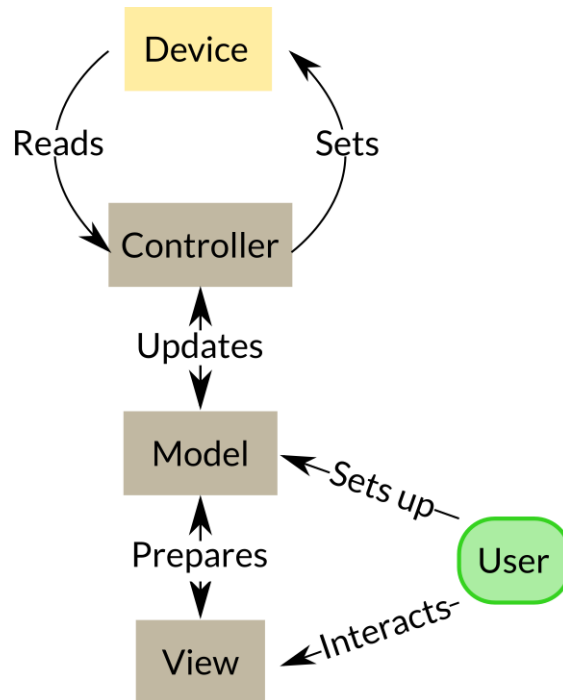
Therefore, when we develop software, we want to keep in mind the following programmer's *mantras*:

- 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 (e.g., oscilloscopes of different brands)
- The code developed in one context has to be available in other contexts (i.e., in other experiments)

The first line does not pose a challenge to be understood. When we talk about solutions developed by others, we mean that often someone already wrote a driver for a device, or they wrote a measurement script. Therefore, it should be easy to get other's code and use them in our projects. Exchanging hardware is something that is not valued until it happens. In most labs, there is always a legacy device that sooner or later breaks down, and we need to replace it. Or you move to a different lab and need to continue with your experiments with different hardware. There are patterns that we can follow to allow a simple exchange of devices. Finally, when we talk about context, we mean that sometimes experiments are very different, but the logic behind it is very similar. We measure the I-V curve of a diode, but it is, by no means, any different from doing a 1-D scan on a confocal microscope, or tuning the wavelength of a laser.

The mantras are no rules. They are just points on which you have to reflect on realizing whether you are departing from the path you wanted to follow when you started. In the following sections, we are going to explore a design pattern for software that has many benefits when developing scientific software for controlling experiments. It is called **The Model-View-Controller for Science**

## 4.2 The MVCs design pattern



A design pattern is nothing more than a set of rules that determine where we can place different parts of the code and how they are going to interact with each other. One of such patterns is called the Model-View-Controller, or MVC for short. When you work with devices in the lab, there is an extra layer that most computer programs lack, which is the interaction with the real world through specific devices. That is why we decided to nickname the pattern MVCs, with the *s* for *science*. Let's see what each component of the MVC is.

A *Controller* for our purposes can also be called a *driver*, which is responsible for communicating with devices. It can be a Python class we developed ourselves, such as the one we did in the previous chapter, but it can also be a Python package developed by someone else. The latter scenario is the case when manufacturers provide the drivers themselves, such as PyPylons from Basler, or the NI-DAQmx bindings for Python. The driver has to reflect the capabilities of the device, nothing less and nothing more. For example, if a device can acquire just a data point at a time, the driver shouldn't include a function for acquiring an array of data using a loop. We briefly discussed this in the previous chapter. Whatever belongs to the logic that a user imposes belongs to the Model component.

The *Model* is where all the logic is defined. In the models, we are going to determine how we are going to use a device for our experiment. A clear example would be the introduction of units. The Device class from the previous chapter takes only integer values as arguments of the methods. If we would like to transform that information into voltages, we could do it in the model.

Moreover, in the experiment itself, we measure voltage, but we can convert it into a current with Ohm's law. This behavior is particular to our experiment, and thus the option shouldn't be hardcoded in the driver. The place to include this information is the Model. The main advantage of splitting *Controllers* and *Models* is that it becomes simple to upgrade or replace a device. You need to update the *Model* to reflect the new options of the device, but the logic of the experiment is left intact.

There is a second type of model, which is the *Experiment Model*, in which we link different devices to perform a measurement. Or we use a single device, but we add the features that



an experiment needs, for example, saving data, plotting, analyzing, transforming units. With straightforward cases, the boundary between the device model and the experiment model can be blurry. Still, when you are dealing with several devices or more complex flows, it becomes much clearer. At the end of the book, we give a reference to some projects we have worked on, which can be a good source of inspiration.

The *View* is 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 set, for example, the start, stop, and step of the experiment. This information is passed to the model to acquire the data, save it, and plot it. We must note, however, that the user interacts through the View with the Model, but never directly with the Controller. It is also essential to keep in mind that we implement all the logic in the Model. For example, if we save the data to files, the procedure to create new filenames should be specified in the Model and not in the View.

For our project, the controller is what we discussed on Chapter 3, the model for the device will be discussed on Chapter 5, the model for the experiment on Chapter 6, and the view will be discussed in Chapters 8 and 9. As you can see, there are still many things to cover in the book. It is important to be patient, because we are going to grow a solution slowly, based on need, and solving the mistakes that may appear as we go, and not just following a path blindly.



### Why Splitting the Code

For people who are new to developing code for the lab, it may be hard to understand why and how to split *Controller* and *Model*. When you have only one device that you use for only one goal, and that is as simple as the device we are using in this book, the differences between model and Controller are very thin. However, when you wish to include code developed by others or when you want to share your code, it is crucial to 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 when doing just one particular experiment.

The meaning of *Model*, *View* and *Controller* changes depending on each developer or community. People developing a web application are not dealing with devices in the real world as developers in the lab do. Therefore, the MVC pattern definition can change from one field to another. The details are not that important; once we establish a structure, if we follow it, everyone else will be able to understand quickly what the code is doing and where. Once we understand what each different component is, we will very quickly understand where we need to change the code to solve a bug or add new functionality.

## 4.3 Structure of The Program

In the program we are developing in this book, we follow the MVC design pattern quite literally. This means that we have to create three folders called *Model*, *View* and *Controller*. In the previous chapter, we have already developed the driver for the device. Go ahead and move the file `pftl_daq.py` into the Controller folder.

## ? Exercise

Create a file **analog\_daq.py** in the Models. Inside the file, define a class called `AnalogDaq` what methods do you think that belongs to the device Model?

In our definition of Model, it is vital to make a further distinction. On the one hand, we have models for the devices we use. In those models, we define things such as units, or how to initialize the device. However, experiments often require to perform complex tasks in which we need to synchronize several instruments. When we perform a measurement, we need to save the data or load the configuration from a file.

## ? Exercise

Create a file in the Models folder, called **experiment.py**. Define a class called `Experiment` and add 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 of an analog output signal. The methods can be empty, don't worry about making it work, but just about the layout. You have to start thinking about the parameters that you need and the order in which you can call every method. For example, to save the data, you need to specify a folder first.

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 thought that sometimes the LED will be plugged into the output channel number 1, sometimes to channel number 0. You don't want to change the code every time you change where you connected the LED. The same happens, for example, with the channel you want to monitor, or the time delay between steps. We include all this behavior in the View that we develop in the last chapters of the book.

Now that you have started to split the code into different folders and files, it is essential to discuss how you can make programs that use the code available in separate files. That is called *importing* and is the focus of the next section.

## 4.4 Importing modules in Python

In the previous chapter, we 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 is important to realize that the import process was different in both cases. *Numpy* is a complex package, with many modules that we can use. The same is true for *time*. However, in the lines above, we have imported only the module *sleep* from package *time*. If we want to use it, we can do:

```
sleep(1)
```

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

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

If we know we only want to use `random` from *Numpy*, we can also import and use it like this:

```
from numpy.random import random  
  
random(1)
```

We may wonder why we would import all of *numpy* if you are just using one of its functions. The name *random* is not defined solely by *numpy*. Python also provides its *random* module. We can import it like this:

```
import random
```

If we needed both *random* functions in the same program, we would have a clash. How would we be sure we are using *Numpy*'s and not Python's function? We may, for example, define our *random* function, and we would like to be able to choose which one to use and, more importantly, we want to avoid generating an unexpected behavior because of redefining functions without realizing it.

When working with our code, we can import different modules in the same way. Open a terminal and navigate to the root folder of the project, i.e., the folder that contains the *Model*, *View*, and *Controller* folders. Start the Python interpreter, and then type:

```
>>> from Controller.pftl_daq import Device
```

Now we have the controller available to use. We can do the following:

```
>>> dev = Device('/dev/ttyACM0')  
>>> dev.initialize()
```

However, there is something strange happening if we run the code above. As soon as we import `Device`, it starts communicating with the device, outputs the identification number, and switches on and off the LED. This behavior is, of course, something we don't want, and a very common pitfall for Python developers. Of course, we don't want to trigger a measurement just because we imported the module to our program.

To avoid running code when importing, we must change the file **pftl\_daq.py**. We can add one line of code at the end of the file, and some indentation, to make it look like this:

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

```
volts = dev.get_analog_input(0)
print(f'Measured {volts}')
sleep(.5)
```

To see the changes, we first must close Python running the `exit()` command, and then opening it again. Python does not import twice the same modules, and therefore, it won't realize that the file with our Controller changed. After we restart Python, we can try again importing the Controller. Now things are going to look fine, and we can use the `Device` as we intended. We can leave the space at the bottom of files to hold examples of how to use the code above. If we ever find the `Device` around, and we don't remember what we were supposed to do with it, we can always go to the bottom of the file and see how it works.

Every time we encounter a behavior which is not easy to understand, the best idea is to transform it into simpler components and explore them one by one. In this case, the importing process may be a bit confusing. To understand it a bit more about, especially how the `if __name__` works we can create a new file called **dummy\_controller.py** in the *Controller* folder, with the following lines of code:

```
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__')
    dummy_example()
```

From the Terminal, we can just type `python Controller.dummy_controller.py`. The output should be:

```
This is the dummy Controller
This is printed only from __main__
This is a function in the dummy Controller
```

What we see is that the entire code got executed.

## ? Exercise

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

Things are going to be different when we import the file. In the Python interpreter we can do the following:

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

First, we notice that the code at the end never gets executed. This means that the `if` statement is not `True`. This block is handy when we want to have code that works standalone (when we execute it directly, for example), but we don't want to execute those lines if we import it. In the case of the real Controller, we wanted to leave some examples at the end to show how we can use it, but when we are importing a class, we don't want it to start communicating with the device.

We can also see that the line `"This is the dummy Controller"` appears only once. Python knows that we have already imported the module `dummy_controller`, and it doesn't execute again the lines that we already imported. Therefore, we have to be aware that it is not a matter of using the `from` in the importing procedure. We can close Python, start it again and revert the order in which we import, and the print is still there the first time but not the second. It means that Python is smart when importing modules, and won't fetch the elements it already has available, even if we are importing in a slightly different way.



## Naming Conventions

We should establish some naming conventions to avoid confusion later on. In Python, any file that defines variables, functions, or classes 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<sup>1</sup> has an excellent chapter covering a lot of the ideas 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.

There is one final remark that is worth mentioning about packages. Right now, we have three folders in our project: `Model`, `View`, and `Controller`. But what would happen if we add a new folder called, for example, `serial`? Next time we try to do `import serial`, how would Python know if it is supposed to import the PySerial library or our package? It can also happen the other way around, perhaps we don't know there is something already called `Controller` in Python, and we don't care about it, we want to import our modules.

For Python to understand that a folder is a *package*, we should add an empty file called `__init__.py`. This file is the way of letting Python know that a folder is more than a plain directory, and thus it should be treated like that. Therefore, we must add the empty init files in the three folders we have created so far. If you are using a Python IDE, such as Pycharm, you notice that these files get created automatically if you chose 'new package' while trying to add a folder.



## Exercise

Create a folder called **random** and inside create a new file called **test.py**. Define a function inside the file. It is not important what it does, and then you save it. Start Python and see what happens if you do `from random import test`. Quit the interpreter and add an empty file called `__init__.py` inside the **random** folder. Try to import **test** again and see what happens.

<sup>1</sup><https://docs.python.org/3.6/tutorial/modules.html>

The init files of packages allow us to specify more complex behaviors when importing, but that goes beyond the scope of this book. Going through other packages is an excellent way of understanding how developers have decided to structure their code.

## 4.5 The PATH variable

There is still a huge difference between our code and the way a library, such as *Numpy*, can be imported. *Numpy* can be imported regardless of where we have started the Python interpreter. We can go to any folder in the Terminal, start Python and import *numpy*. However, we can import our package only if we are in its directory. If we are sitting on a different folder and try to import the Controller, we get an error like the following:

```
>>> from Controller.pftl_daq import Device
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'controller'
```

Python searches for packages in specific locations, and once it finds one, it stops searching. *Numpy* is located in one of the folders that Python uses, but Python is not aware of our package yet. One way to let Python know where our package is located, is by adding the folder to a system variable called *PYTHONPATH*.

On Windows, we can follow the steps explained in Section 2.4.2, when we were dealing with the details of adding environment variables after installing Python. The only difference is that we should use the variable *PYTHONPATH* instead of just *PATH*. On Linux, it is enough to run the following command:

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

We have to change `/path/to/PFTL` with the full path to the folder where we are keeping the code. After doing that, we can type `from Controller import *` wherever you are in your computer, and Python finds the appropriate folder. On Linux, the change is not permanent, and we need to rerun the line next time we start the Terminal. We can modify environment variables permanently, but this goes too much into the details of the operating systems.

If you are using an IDE to develop and run the code, you may have noticed that there is no need to add anything to the Python path. It is one of the advantages of using robust programs to develop software. They take care of many things for us. However, at some point, it is also essential to be able to run the program independently of the editor software.

## 4.6 The Final Layout

At this stage, we should have a clear separation of the code into Model, Controller, and View. Most of them are, for the time being, empty folders. However, we are not limited to having only three folders in our project. Most likely, we want to provide some examples of how to use the code or some documentation.

However, if we create extra folders next to the three main ones, the structure of the program starts to be polluted. It won't be clear what is part of the program, and what is a user-specific setting. Therefore, it is a common practice to make a folder to hold all the program and, next to it,

we create an extra folder to contain non-essential elements. In our case, the folder structure would look like this:

```
├── Docs/
├── Examples/
└── PythonForTheLab/
    ├── Controller/
    │   ├── __init__.py
    │   └── pftl_daq_01.py
    ├── Model/
    │   └── __init__.py
    └── View/
        └── __init__.py
        __init__.py
```

There are three folders at the top level: *Docs*, *Examples* and *PythonForTheLab*. The last one is holding the *Model*, *View* and *Controller* with all the code that you have developed up to now. The *PythonForTheLab* also requires an `__init__.py` file, because it is our main package. This folder structure would allow us to do the following:

```
>>> from PythonForTheLab.Controller import pftl_daq
```

This code is clearer than importing the *Controller* directly, because it also allows us to, in principle, have different programs at the same time and import only the elements we need from each one. For example, we could have one big project with many devices and complex measurements, and a smaller program to do more specialized tasks that only require *some* of the devices.

## 4.7 Conclusions

Following a pattern when programming makes the code much easier to understand, to maintain, and to explain to others. Patterns, however, are not set in stone. Sometimes there is no clear divide between what we should develop where. The important thing is to be consistent. For example, if at some point it is decided that controllers should handle real-world units, then all controllers should do it. If, on the other hand, we agreed that models should handle units, then all models should do it, regardless of what the specific controllers do.

Following a pattern is useful not only when there are many developers involved in the same project, but also when the same person is working on different projects, or if they change labs, or experiments. As time passes, we start accumulating a collection of tools, and being able to transfer them from one experiment to another makes it much faster to have a measurement running without reinventing the wheel over and over again.

Another essential aspect for many scientists is that solutions should be developed as quickly as possible to test out new ideas. And this is how we laid out this book. First, we quickly manage to communicate with the device. We saw we can acquire data, switch on and off the LED. If the measurement would be a one-off task, then we would have finished. But if we would like to start changing parameters, or using the code for another experiment, then we must keep going. Expanding the code is what we are going to do next. In these sections, we have explored a sustainable way of following the *Onion Principle*. Factoring also concerns about the speed at which we can implement a solution for the lab.

The strategies proposed in this chapter do not come naturally to every developer, and even if you know them, you can try to find shortcuts. The truth is that no one does an experiment only once. The key to better science is reproducibility, and the clearer the code that enabled you to acquire data, the easier it is to repeat a measurement, even by someone else. We can only lay out the foundations for what we consider a robust solution. If you find a different path, you are always free to follow it, but never miss the bigger picture from your sight.



# Chapter 5

## Writing a Model for the Device

### 5.1 Introduction

The secret to successfully develop a model for a device is to think. We need to know what we expect from the device, how are we planning to use it. The first time we start thinking about models, it can become complicated because we need to anticipate our future needs. For example, the PFTL DAQ device doesn't handle units. It would be great if the Model would allow us to specify the output voltage instead of converting it to an integer because it is what the driver uses. This requirement seems trivial and probably is the first one that comes to mind. Later, once you start using the program, you can see that some other useful options were missing, and you could have saved much time if you would have thought about them.

There is no magical recipe to teach precisely how to develop models for devices. Each device and each experiment is unique; the best we can do is to focus on the task at hand. We can extrapolate the rest to other devices and experiments. Once you understand the role that models play, you can use them for very different purposes, without needing to re-write the entire program. With a simple device and a simple experiment, the gain of having models separated from controllers may not be immediate. Still, as soon as the complexity grows, the value becomes apparent.

#### Device Manuals

It is impossible to overestimate the importance of reading the manuals of your devices. Hardware in the lab is not the same as consumer hardware. Things can break, signals may not make sense. Be always sure to understand the limits under which each component operates.

### 5.2 Device Model

We must first thing how do we want to interact with that particular device. Of course, we would like to initialize it, set a voltage, read a voltage, and finalize the device. But we don't want just to repeat what the Controller can do. When we initialize or finalize the device, we want to be sure the output voltages are at 0 V. In this way, we can ensure that no current flows through the LED unless we explicitly want it. We also want to be able to use values in volts when setting an output, and getting values in volts when reading a voltage.

We can develop a skeleton of the Model. We can use empty methods to have an idea of what we need to develop and the arguments and outputs of each method. Let's start by creating a file **analog\_daq.py** in the *Model* folder, we can then add the following code:

```
class AnalogDaq:
    def __init__(self, port):
        pass

    def initialize(self):
        pass

    def get_voltage(self, channel):
        pass

    def set_voltage(self, channel, volts):
        pass

    def finalize(self):
        pass
```

Now we can start step by step. We start very similarly to how we started the Controller; the `AnalogDaq` class takes `port` as an argument for initializing. The main difference is that we don't use `PySerial` directly, but we use the Controller. We can start improving our code, like this:

```
from PythonForTheLab.Controller.pftl_daq import Device

class AnalogDaq:
    def __init__(self, port):
        self.port = port
        self.driver = Device(self.port)

    def initialize(self):
        self.driver.initialize()
        self.set_voltage(0, 0)
        self.set_voltage(1, 0)
```

We initialize the class by storing the `port` and by creating a `self.driver` attribute. Remember that the `Device` has a separate method for initializing. The `initialize` method now not only initializes the the driver itself, but also sets the output voltages to 0. We haven't developed a way of setting voltaes, yet, but we see the flow. The same works for the `finalize` method:

```
def finalize(self):
    self.set_voltage(0, 0)
    self.set_voltage(1, 0)
    self.driver.finalize()
```

We first set the voltages to 0, and then we finalize the Controller. This is a clear example of our own logic imposed on the device. In some cases, we don't want to set the voltage to 0 when closing the communication. Perhaps we are just switching on a laser, and we want it to stay on even if we switch off the computer, or we are using piezo stages and is not recommended to suddenly shake them by setting a different voltage, it is better just to leave a voltage applied to them. That is why adding these features to the Controller would imply violating the separation of models and controllers. What we do with the voltages is part of the logic, not of the device itself. Now that we have this code, we can also add an example to the end of the file, to show how to use the Model:

```

if __name__ == "__main__":
    daq = AnalogDaq('/dev/ttyACM0')
    daq.initialize()
    print(input_volts)
    daq.finalize()

```

The last missing bits are the methods for getting and setting a voltage. What we do in these steps was discussed in Section 3.7.1. To set a voltage, we first need to transform a number in the range 0 – 3.3 to an integer in the range 0 – 4095 and then we apply it:

```

def set_voltage(self, channel, volts):
    voltage_bits = volts*4095/3.3
    self.driver.set_analog_output(channel, voltage_bits)

```

And we can do the same for the get method:

```

def get_voltage(self, channel):
    voltage_bits = self.driver.get_analog_input(channel)
    voltage = voltage_bits*3.3/1023
    return voltage

```

And that is all that is needed. We can now run the code and see that we are reading voltages and setting voltages. Because of how the experiment works, the values we get at the analog input are going to be very small, in the order of few tens of millivolts, but enough to be detected by the PFTL DAQ.

## 5.3 Base Model

In this book, we are working with only one device, and therefore we are using only one model. But if we want to make our program compatible with more devices, we need to start developing models for each new device. Since we have one model, it would look reasonable to copy it and adapt the methods based on what the new drivers allow us to do. Another option is to create a base class that all other models inherit. In this way, we know that all the methods are defined, perhaps they don't do anything, but at least they are there.

What we do in this section is not a requirement to keep going, but it is important to show the pattern because sooner or later, when the program grows, it becomes a handy approach. Create a file called **base\_daq.py** inside the Controller folder, we can add the following code to it:

```

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

    def initialize(self):
        pass

    def get_voltage(self, channel):
        pass

    def set_voltage(self, channel, volts):
        pass

```

```
def finalize(self):  
    pass
```

This class doesn't do anything by itself. It is only the schematics of what a model should contain. We added `pass` after every definition to take care of functions in which nothing happens. We can see that we also specify the arguments that each method takes: `initialize` takes a port, setting a value takes channel and value, and so forth. In programming, this is also called an API, or Application Programming Interface. The base class defines the interface that all the DAQ models use. There is an `initialize` method, a `get` and `set` analog and a `finalize`. Just by looking at this simple example, we already know how things are going to work and what do we need to do to make them work.

As an example, let's create a dummy DAQ that can generate random values when requested, and since it is not connected to any real device, it doesn't do anything else. We can add the following code to **dummy\_daq.py** in the Model folder:

```
from random import random  
from PythonForTheLab.Model.base_daq import DAQBase  
  
class DummyDaq(DAQBase):  
    def get_analog_value(self, channel):  
        return random()
```

And if we copy the example code from our real daq, things are still going to work fine:

```
if __name__ == "__main__":  
    daq = DummyDaq('/dev/ttyACM0')  
    daq.initialize()  
    voltage = 3  
    daq.set_voltage(0, voltage)  
    input_volts = daq.get_voltage(0)  
    print(input_volts)  
    daq.finalize()
```

Of course, when we set a voltage, initialize, or finalize the Model, nothing happens, but when we ask for a value, we get one. You see that the way of using this class is the same as the real Model, and it took us only 3 lines of code to develop. Perhaps in the future you move to a more complex DAQ, such as an oscilloscope. If you maintain the same names for the methods of the Model, everything keeps working in the same way.

### Exercise

Update the real Model to inherit from the base class

## 5.4 Adding real units to the code

The Model for the PFTL DAQ device is working great, but it has one problem. It allows us to set the output in volts and reads a value in volts. But if we ever make the mistake of supplying the value in millivolts, the program won't work as expected. In real cases, it is tough to remember the units that every method should take. Sometimes you have very different outputs and inputs, with each

one taking different units. Imagine you want to make a periodic signal, perhaps the device asks for the frequency, perhaps for the period.

In Python we can overcome the limitations of working with plain numbers by using a package called *Pint*, that allows us to work with *real* units. Let's quickly see how *Pint* can be used with a simple example:

```
>>> import pint
>>> ur = pint.UnitRegistry()
>>> meter = ur('meter')
>>> b = 5*meter
>>> type(b)
<class 'pint.quantity.build_quantity_class.<locals>.Quantity'>
>>> print(b)
5 meter
>>> c = b.to('inch')
>>> print(c)
196.8503937007874 inch
```

First, we import the package and start the *Unit Registry*. In principle, *Pint* allows us 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 assign the value of 5 meters to `b`. In this case, `b` is of type `Quantity`. Therefore it is not just a number but a number and a unit attached to it. *Pint* allows us to convert between units, and this is how we create the variable `c`, which is 5 meters converted to inches. And things can get very interesting:

```
>>> b == c
True
```

Even if the numeric value of `b` and `c` is different, they are still equal to each other, exactly as we would have imagined. We can also work with more complex units:

```
>>> 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
```

So far we have always been transforming between units of the same type, i.e. a length in meters to a length in inches, etc. But *Pint* can also handle combined units such as what happens with voltage, current and resistance:

```
>>> 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
```

The snippet above shows you that Pint can understand the relationship between Amperes, Ohms, and Volts. There is one more feature that is important to point out: Pint can parse strings to separate the units from the numbers. For example, we can do the following:

```
>>> current = ur('5 A')
>>> resistance = ur('10 ohm')
```

Being able to parse strings so easily is going to make our life much easier when we will be dealing with user input. Now we have seen how to handle *real* units on our code. However, the device still requires us to set the output using plain numbers. In the context of Pint, just the number, without the units is called the *magnitude*. To get the magnitude out of a quantity, we can do the following:

```
>>> current = ur('5 A')
>>> current_mag = current.m
>>> print(current)
5 ampere
>>> print(current_mag)
5
>>> current_ma = current.m_as('mA')
>>> print(current_ma)
5000.0
```

We start with a quantity called `current` of 5 A. If we just append the `.m` to the variable, we get the magnitude in whatever unit it is already expressed. If we want to be sure to get the magnitude in a specific unit, we use the command `.m_as()`. In our case, we will need to transform the user input to an integer, and we will not need to assume it is in volts, we can transform it to volts before converting it to an integer. The `set_voltage` method would look like this:

```
def set_voltage(self, channel, volts):
    value_volts = volts.m_as('V')
    value_int = round(value_volts / 3.3 * 4095)
    self.driver.set_analog_output(channel, value_int)
```

We transform the value to volts and get only the magnitude. Then we transform that value to bits, using the `round` function to get an integer after the operation. We use that rounded value to set the output on the device. Our program is now very flexible since the user can provide the output value in whatever units she pleases, provided that Pint can transform them into volts.

## ? Exercise

Update the method `get_voltage` so it generates an in volts. Pay attention to the fact that you need to import Pint and create the unit registry before you define your class to be able to use it.

We should also update the method for getting a voltage to return a voltage and not a plain number. The code below only shows the parts that have changed or added, not the entire class:

```
import pint

ur = pint.UnitRegistry()

[...]

def get_voltage(self, channel):
    voltage_bits = self.driver.get_analog_input(channel)
    voltage = voltage_bits * ur('3.3V')/1023
    return voltage
```

However, we also need to update the `initialize` and `finalize` methods in order to use units and not plain numbers:

```
def initialize(self):
    self.driver.initialize()
    self.set_voltage(0, ur('0V'))
    self.set_voltage(1, ur('0V'))

def finalize(self):
    self.set_voltage(0, ur('0V'))
    self.set_voltage(1, ur('0V'))
    self.driver.finalize()
```

The class is complete and we need to update the example code at the bottom of the file in order to use the real units:

```
if __name__ == "__main__":
    daq = AnalogDaq('/dev/ttyACM0')
    daq.initialize()
    voltage = ur('3000mV')
    daq.set_voltage(0, voltage)
    input_volts = daq.get_voltage(0)
    print(input_volts)
    daq.finalize()
```

If you are hesitant about the impact that different unit systems can have, there is a great example involving a multi-million dollar satellite<sup>1</sup>. The Mars Climate Orbiter fell from its orbit because engineers from the US failed at using the established units of measure in their software, resulting in a mix of metric and imperial systems.

## 5.5 Testing the DAQ Model

At this point, we have a very functional program. We can handle units, we have split the logic of the units from the driver, meaning that we can easily share our code with colleagues or the rest of the world. It is time to test our program. One of the reasons we created the Examples folder was to be able to add extra python files that don't belong to our core program. In the Examples folder, create a file called **test\_daq.py**, and we can start using the Model to do some measurements:

---

<sup>1</sup>You can check the <http://articles.latimes.com/1999/oct/01/news/mn-17288>

```

import numpy as np
import pint

from PythonForTheLab.Model.analog_daq import AnalogDaq

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

daq = AnalogDaq('/dev/ttyACM0') # <-- Remember to change the port
daq.initialize()
# 11 Values with units in a numpy array... 0, 0.3, 0.6, etc.
volt_range = np.linspace(0, 3, 11) * V
currents = [] # Empty list to store the values

for volt in volt_range:
    daq.set_voltage(0, volt)
    currents.append(daq.get_voltage(0))

print(currents)

```

We can run the code above, but we will notice that we are printing currents with units of volts. This can be very confusing for someone looking at our results, or even for ourselves. We have to remember that we can transform volts to amperes by deviding them with the resistance we are using. If we have a 100 Ohm resistance, we can do the following:

```

for volt in volt_range:
    daq.set_voltage(0, volt)
    measured_voltage = daq.get_voltage(0)
    current = measured_voltage/ur('100ohm')
    currents.append(current)

```

If we run the code with the changes above, we will get the following error:

```

[...]
ValueError: Cannot operate with Quantity and Quantity of different registries.

```

The error is descriptive, but hard to understand without knowing the underlying principles of Pint. The unit registry is a collection of rules that allows us to transform from one quantity to another. But these rules belong to a unit registry. In principle, two distinct unit registries hold rules for different sets of units. It means that we can't convert units across unit registries. We need to use only one registry throughout the program. Right now, we are creating the registry in two different places: The device model and the example.

Since units belong to the entire program, it could be a good idea to define the unit registry at the root. In other words, we can create it directly in the `__init__.py` file that we placed in the PythonForTheLab folder:

```

import pint

ur = pint.UnitRegistry()

```

Every time we want to use units and the unit registry, we can do the following:



```
from PythonForTheLab import ur
```

### ? Exercise

Improve DAQ model to use the central unit registry and not one defined locally.

### ? Exercise

Modify the example that we developed for testing the DAQ model, so it uses the central unit registry, and see that it works as expected.

After completing the exercises above, we should be able to run the example and get the values we wanted. We are getting currents in amperes, we are setting voltages in volts. We are still missing some details, such as saving data, but the core of the measurement is already there.

## 5.6 Appending to the PATH at runtime

In Section 4.5, we have seen how to add the root folder of the project to the computer's PYTHON-PATH. It allows Python to find our program and allows us to import the packages and modules very easily. However, altering environment variables in different Operating Systems is not only cumbersome, but it can also lead to unwanted results. For example, we may be overwriting something important if there is any name clash between the program we develop and some other library on the computer.

Therefore, we can go a different route and add the folder to the path directly from within Python. This change is not permanent, and it is in place only while the program runs, but no further. First, we need to learn how to identify the folder we want to add to the path. For this, Python offers a module called `os`. The code below looks cumbersome, but we explain it after. We can add the following line at the beginning of the `test_daq.py` file:

```
import os

base_dir = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

First, `__file__` variable is a way of letting Python know we are interested in the current file, which in this case is `test_daq.py`. We can start from the inside of the function. First, we get the absolute path to the file, including all the folders to get there. Then, we grab the directory that holds the file, which would be the *Examples* folder. And then, we grab the directory that contains the examples, which in our case is the root folder or `base_dir` as we called it.

Next, we need to add the `base_dir` to the path. For this, we use another package called `sys`, which is a wrapper for the operating system. It means that this package adapts according to which operating system we use to run the program. To add the folder, we only need to do the following:

```
import sys

sys.path.append(base_dir)
```

And that is it. It doesn't matter if we modify the `PYTHONPATH` variable anymore, we can always run the `test_daq.py` file.

### The PATH

Since appending to path only works while the program runs, if we try to run the package files independently, Python does not know where to find the modules. The test files we created in Examples are what are called entry points, and the program should be run directly from them

## 5.7 Real World Example

The usefulness of models at this stage may still be obscure. We may be tempted to define units and transformation in the Controller itself. We know we are the only ones using it, just to do one experiment. It may be fine when you start, but at some point, the code grows because the experiment is progressing and getting more complex. As an example, at Python for the Lab, we have developed software for controlling a microscope using a camera. However, there were several cameras available, some more expensive and powerful, and they were all shared between people and experiments.

Therefore, having a flexible way of using different cameras for the same experiment became mandatory. Sometimes we would use a Hamamatsu, sometimes a Basler, sometimes a Photonics Science. However, each camera had an incredibly different way of working. First, Hamamatsu didn't provide any drivers written in Python. Photonics Science shared an internal tool they used, and Basler as an entire package called `PyPylon` to control their cameras. The controller layer, therefore, was not developed by us but was given.

At the model level, however, we made sure that all cameras would work in the same way. They all have the same method for setting the exposure or changing the region of interest. Therefore, the only thing that we needed to change to run an experiment with one or the other camera was changing what Model we were importing. The rest of the code would stay the same. If you want to see the real code, you can head to the repository on of the UUTrack project<sup>2</sup>.

## 5.8 Conclusions

In this chapter, we have seen what does *Model* mean in the MVC pattern. We focus on adding features to how the device works, such as switching off the outputs when we finalize using the device. We have also included real-world units by using **Pint** and learned some of its quirks regarding the unit registry.

We have also covered how to append folders to the path through Python. It allows us to run all the import statements that we want, without having to alter the environment variables of the operating system manually. It is handy because, on the one hand, it runs unaltered in Linux,

---

<sup>2</sup><https://github.com/uetke/UUTrack>

Windows, and Mac. On the other, we don't make any permanent changes to the configuration of the computer. It is a possibility that at some point we have two projects with the same name, because we use them for two different but very similar experiments, for example, but we want to be sure we are importing the correct one.



# Chapter 6

## Writing The Experiment Model

### 6.1 Introduction

In the **MVCs** design pattern, the *Model* is where we should place the logic and the decisions we make to perform a measurement. In the previous chapter, we started specifying the logic of how we are going to use the DAQ device. We decided, for example, that the device should start and finish with the outputs set to 0 V, and that it should handle Pint units for setting and reading values. Even though it was a great start, there are still steps missing to perform a real experiment.

We are still missing, for example, the possibility to perform a scan in a given range of voltages, there is no way of saving data nor the parameters used to generate it. We can develop these tools in a script, and that is fine to get started. At some point, however, we want to be systematic, and we don't want to keep editing a script every time we must perform a measurement. We want to save data consistently, or we want to enable other people to run their experiments based on our routines.

At this moment is when the *onion principle* starts making sense. We are slowly building in complexity, one layer at a time. We started with the driver, built some logic on top of it through a device model, and now we can keep growing the complexity by defining an Experiment model. There are many different steps needed to perform a reproducible measurement, and we cover them one by one in this chapter. One important aspect we are missing, on top of the ones mentioned earlier is, for example, transforming the voltages we measure to a current, so we can give sense to the **I** on the I-V curve we are trying to measure.



#### Splitting into Modules

With a simple device and experiment such as the one we use in this book, sometimes it is hard to put a limit on what should go in the device model and what into the experiment model. There are no strict rules. We should always reflect on what we believe is useful in the long run. Once we are comfortable working with classes in Python, splitting the code into reusable, smaller modules does not lead to much more typing.

There are two extra advantages of developing an experiment model, which become apparent only after working on these topics long enough. On the one hand, the models are going to lead to reproducible experiments. If we keep track of the parameters, we can go back precisely to the same conditions in which we have performed a specific measurement. On the other hand, having a

well-structured model is going to make it very straightforward to build a Graphical User Interface (GUI) on top of it, but let's not get ahead of ourselves.

## 6.2 The Skeleton of an Experiment Model

Every time we want to start developing a model, it is handy to start from an empty skeleton. It helps us know what parts of the code we need to develop, what things we don't know how to do yet and need to learn. It also allows us to have a quick glimpse of how we expect our program to be used. We did it for the device model in the previous chapter, and we can do the same for the experiment in this chapter. First, we create a file called **experiment.py** inside the *Model* folder, and then we think about the steps needed to perform a measurement:

```
class Experiment:
    def __init__(self, config_file):
        pass

    def load_config(self):
        pass

    def load_daq(self):
        pass

    def do_scan(self):
        pass

    def save_data(self):
        pass

    def finalize(self):
        pass
```

Most of the code should be self-explanatory, but some steps are worth mentioning. First, we include a `load_config` method, which will rely on the `config_file` we specify at the `__init__`. Having a separate config file is useful not only to change parameters between measurements, but it helps us with our code. Especially when the number of things we need to remember grows, it is always handy to have a file where to look at how we named things.

We have also included a `load_daq` method. So far, we have only one DAQ to use, but we go a bit ahead of time and assume that at some point, there may be more than one, and therefore we should have a way of loading one or the other. It can also be a case of premature optimization, in which we anticipate a future that never comes. Nevertheless, it is pedagogically useful to define it there.

The rest of the methods may seem intuitive enough. If we are missing extra pieces, we can always come back and add them. Now it is time to start developing each one of the building blocks.

## 6.3 The Configuration File

### 6.3.1 Working with YAML files

Before we keep developing code, we need to stop for a second to think about what do we want our program to do. We need to think about what inputs we need for our program. For example, we

need to know the port to which the device is connected. We also know that we need to define an output and input channel, a range for the scan, a delay between data points. Before going into the details, let's see how to store all these parameters into a text file easily.

In the *Examples* folder, create a file called **experiment.yml**. We can create the file with any text editor, including the one we are using for editing Python files. The only difference is the extension *yml*. We are going to use this file to hold all the parameters of the experiment. The format for the file is called YAML, which has a straightforward structure; it looks like this:

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

YAML is very simple to read both by a person and by the computer. It has just a few rules. The most important one is that the indentation is 2 spaces. In the example above, there is a main element called **Experiment**; everything that is indented compared to that element belongs to it. To read the file, we are going to use a package called PyYAML, which we installed in Chapter 2. We can create a file called **test\_yaml.py** also in the Examples folder to understand how to use these files:

```
import yaml

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

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

We begin by opening the file using `open`. We use the `with` command because it is convenient for working with files and other resources that have the same pattern of opening/doing/closing. `Yaml` is then responsible for interpreting the information contained in the file. We store this information in a variable `e`, which turns out to be a dictionary.

To get the elements stored in dictionaries, we use keys. In the example above, there is a main key called `'Experiment'`, and the sub-keys `'name'`, `'range'` and `'list'`. If we want to use one of those elements, we can type `e['Experiment']['name']`, for example. The code prints out each element, separated by a horizontal line. `Yaml` imported the file directly as a dictionary, but some of the elements are special. We can see that `'name'` is a string, but `'range'` and `'list'` are not. `YAML` automatically detects what kind of information we are storing, making our job easier if we already know what we want to store.

## ? Exercise

What type of variables has `yaml` generated for `'range'` and the `'list'`? (Remember that you can use `type(var)` to know the type of the variable.

## ? Exercise

YAML also supports numeric information. Create a new element and assign it a value of `1` or `3.14`. What kind of variable has YAML created in such cases?

Of course, YAML can be used not only to read properties but also to save information generated in a program. Instead of loading data, we can use `yaml.dump` method to save it. For example, we can define a dictionary with the following information:

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

If we want to save it to a file, we can do the following:

```
with open('data.yaml', 'w') as f:  
    f.write(yaml.dump(d, default_flow_style=False))
```

We are using the `'w'` option to open the file, which means that every time we run the code, it overwrites the file, and we lose the previous contents. After running the code, we can open the file **data.yaml** with any text editor and see that the contents are very similar to the one we created ourselves earlier. One of the advantages of the YAML format is that files are straightforward to read, and don't require much typing.

## ? Exercise

Read back the contents of **data.yaml** and check that they are the same you saved.

## ? Exercise

Modify some values in **data.yaml** directly with your text editor and see that those changes are reflected when you reread the file.

## ? 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?



### ? Exercise: Advanced

Save a numpy array using yaml. Then activate a virtual environment in which PyYAML is available but not numpy. Try to load the contents of the file. What happens?

Now that we know how to work with YAML files, it is time to start thinking about the experiment again. We need to stop and think about what do we need to know to perform an experiment.

### ? Exercise

Create a new `experiment.yml` file in the *Examples* folder. Use what you have learned about YAML files to write a file that contains all the information that you need to perform an experiment and interpret its data. For example, knowing *who* performed an experiment may be important.

## 6.3.2 Loading the Config file

We have decided to use YAML files because of their simplicity, and because it is easy to map them to dictionaries in Python. The main advantage of having a separated config file is not only that it allows us to run different measurements changing parameters as we desire, but that it also helps us keep our code well organized. We always know how to get the information we need, just by looking at the config file.

First, we need to decide what we are going to include in the configuration file. The information is not static, and it may happen that after working for a while, we realize there is some important parameter missing or that something we thought was important is not necessary anymore. Every time we find ourselves deciding a value, that information should go to the config file. For example, if we need to decide in which port the DAQ is connected, that goes to the config file. We are going to use the **experiment.yml** file in the *Examples* folder to store all this information. Below, we show how the config file could look like. We have included extra options that we haven't discussed yet, such as the user performing the experiment, which can be relevant for bookkeeping:

```
User:
  name: Aquiles

DAQ:
  name: AnalogDaq
  port: /dev/ttyACM0

Scan:
  start: 0V
  stop: 3.2V
  step: 400mV
  channel_out: 0
  channel_in: 0
  delay: 100ms

Saving:
  filename: data.dat # Files won't be overwritten, but renamed as data_001.dat,
  ↪ etc.
```

We haven't used a primary key such as *experiment* because it does not add anything, and forces us to type more. If for some reason, we needed to store parameters for two different experiments on the same file, then we could add two top-level keys such as `Experiment_1`, `Experiment_2`. The parameters above are enough to get started and then slowly keep adding or modifying when we need it. There are some glaring omissions, meant to trigger discussions further down the road. If you have already spotted them, you have done an excellent job, but don't stress yourself if you haven't.

We have to update the `Experiment` class to load the file. We are storing the filename within the class, so all our methods can be triggered without arguments. But this is just a choice we've made:

```
import yaml

class Experiment:
    def __init__(self, config_file):
        self.config_file = config_file

    def load_config(self):
        with open(self.config_file, 'r') as f:
            data = yaml.load(f, Loader=yaml.FullLoader)
        self.config = data
```

We have used the same code we showed in the previous Section but as part of a method. When we load the configuration file, it is stored as an attribute of the `Experiment` class, called `self.config`. Now that the experiment is starting to have shape, we can create a file to show how to use it and check whether we are doing things properly. In the *Examples* folder, we can create a file called **run\_experiment.py**, and add the following:

```
import sys
import os

base_dir = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
sys.path.append(base_dir)

from PythonForTheLab.Model.experiment import Experiment

experiment = Experiment('experiment.yml')
experiment.load_config()
print(experiment.config)
```

We use the strategy explained in Section 4.5 to let Python know where to find our program. The rest is easy to understand; we start an experiment, load the config file, and print the parameters that we have loaded. So far, we are not doing much, but it is a good start.

Now we have a consistent interface for loading the configuration file into our programs. It gives us a great deal of flexibility and makes the code nicely reusable and extendable.

## ? Exercise

When the function `load_config` loads the configuration from a file, verify that some essential parameters are defined. For example, verify that there is a user associated with

an experiment. If something is missing, print an error message or raise an Exception.

## 6.4 Loading the DAQ

To load the DAQ into our experiment, we can start with the simplest approach. We import the model for the daq, and we initialize it using the `load_daq` method. We can update the experiment model (we omitted some parts of the code for brevity):

```
from PythonForTheLab.Model.analog_daq import AnalogDaq

[...]

def load_daq(self):
    self.daq = AnalogDaq(self.config['DAQ']['port'])
    self.daq.initialize()
```

The `load_daq` method uses the information stored in the configuration file to instantiate and initialize the daq model. Having a config file in YAML format makes it very easy to navigate and find where is located the information of the port for the daq. The main key is `DAQ`, and the sub-key is `port`. In this way, there are many fewer things we need to keep in our heads. We can always go back to the file and read what we need to know. We have also decided that we initialize the device right after loading it. We could have developed a separate method, but at this stage and for this experiment, there is no real need.

Since we are taking care of loading and initializing the DAQ, we can also develop the finalize method, which takes just one line:

```
def finalize(self):
    self.daq.finalize()
```

We can update the `run_experiment.py` file to reflect the additions we have just made to the Experiment:

```
[...]
experiment.load_daq()
print(experiment.daq)
experiment.finalize()
```

When we run the code above, we see that the output of `print(experiment.daq)` is not pretty, and it is hard to understand. We can update the device model to make the string that appears on screen nicer. We edit the `analog_daq.py` file to include the following method in the `AnalogDaq` class:

```
[...]

class AnalogDaq:
    [...]

    def __str__(self):
        return "Analog Daq"
```

The method `__str__` is called a *dunder* method, because it has the double underscore before and after its name. These are magic methods in classes that allow us to change the behavior at a

much lower level. The *str* method is responsible for letting Python know how to transform an object to a string. It happens, for example, when we use the `print` function. If we rerun the experiment, we see that the output is clearer than before.

## String Representation

Adding a string representation to the classes we build is an excellent addition but not mandatory. We should be careful not to derail on details that don't bring us close to the goal of performing a measurement.

### 6.4.1 The Dummy DAQ

In the previous chapter, we took a small detour when we discussed the creation of a base class for the device models. In Section 5.3, we also discussed creating a dummy device, a fake model that outputs random numbers when requested. It means that we have two different DAQ devices we can use, one real and one fake. Therefore, we can improve the Experiment model to accommodate the possibility of using one or the other. We are going to follow a non-standard approach, which does not comply with the general principles of Python, but that it is nevertheless convenient. In the config file, we included the name of the DAQ, and it is time to use this information. There will be two possibilities, either `AnalogDaq`, or `DummyDaq`:

```
def load_daq(self):
    name = self.config['DAQ']['name']
    port = self.config['DAQ']['port']
    if name == 'DummyDaq':
        from PythonForTheLab.Model.dummy_daq import DummyDaq
        self.daq = DummyDaq(port)

    elif name == 'AnalogDaq':
        from PythonForTheLab.Model.analog_daq import AnalogDaq
        self.daq = AnalogDaq(port)

    else:
        raise Exception('The daq specified is not yet supported')

    self.daq.initialize()
```

The `load_daq` method is now much more powerful than before, and we are doing something that may seem strange at first sight. We are importing Python modules not at the top of the file, but deep inside a method. It is not standard, and in some contexts, it is discouraged, but it is crucial to understand why we have decided to follow this approach. When working with real devices, the models may depend on drivers that are not installed on the computer. If we import the device model at the top of the experiment file, we may get an error because of a device we do not intend to use.

On the other hand, this behavior is discouraged because if there is a problem in one of the modules, we won't notice it until we are running the program, and that can be a waste of our time and, more importantly, of data. For example, `AnalogDaq` depends on having a proper controller available. If we make the import at the top of the class, and the controller is not in place, we get an error even before we instantiate the experiment class. If we plan to use only the dummy model, then we don't care about this. The balance between safety, best practices, and ease of development

is sometimes hard to choose. What is important to remember is that Python is an incredibly flexible language.

We have also included a final clause in our if-statement to raise an exception if the specified model is not one of the two with which we know how to work. The advantage of having models that specify the same API is that after we load and instantiate them, they both work in the same way. Therefore the rest of our code is completely independent of which model we are using.

### ? Exercise

In Section 5.4 we included units for the `AnalogDaq` class, but we didn't add units for the `DummyDaq`. It is an inconsistency since both models are going to generate different types of output. Update the dummy model to generate random values including units of volts.

## 6.5 Doing a Scan

The core of the experiment model is being able to perform a scan, changing the voltages on the output, and reading the voltages in the input. This kind of measurement is widespread in a lot of different experiments, not only in electronics. That is the reason we decided to call it `scan`, which is a fairly generic name.

### ? Exercise

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

To perform a scan, we use the parameters that we have defined in the `Scan` Section of the config file. Since we already performed this type of measurement either from the command line or from example files, it is easy to adapt to what we did to the method in the `Experiment` class:

```
from PythonForTheLab import ur
[...]
```

```
def do_scan(self):
    start = ur(self.config['Scan']['start'])
    stop = ur(self.config['Scan']['stop'])
    step = ur(self.config['Scan']['step'])
```

After the first few lines, we need to stop and think. After we introduced units to our model, we didn't try to perform a scan, and therefore we need to see how Pint works. We know that `start`, `stop`, and `step` all have units of volts or related. However, if we try to use the `start`, `stop`, and `step` values as they are, we will fail. Note the Python `range` function, nor Numpy's `arange` know how to deal with quantities. However, Pint allows us to transform a quantity to a plain number in given units, by using the `m_as` method. Therefore, if we want to have a range of values over which to perform the scan, we can do this:

```
scan_range = np.arange(start.m_as('V'), stop.m_as('V'), step.m_as('V'))
```

If we explore the `scan_range` variable, we will see that it includes the numbers starting with `start`, going in increments of `step`, but it does not include the `stop` value. This is not a bug, but just how `arange` works. The documentation clearly states that `arange` generates values in the semi-open interval `[start, stop)`. If we want to include the last point or not, is debatable. One possible strategy would be to do this:

```
scan_range = np.arange(start.m_as('V'), stop.m_as('V')+step.m_as('V'),  
↳ step.m_as('V'))
```

But this has an associated risk. What happens if `start` is 0 V, `stop` is 3.3 V and `step` is 0.5 V? The last value in the range would be higher than 3.3 V. It means that forcing the `stop` to be larger only works if the `step` divides the range in an integer number of intervals.

We could use numpy's `linspace`, which allows us to generate equally spaced values if we provide a `start`, `stop` and the number of points we want. This seems like a viable solution. We can do the following:

```
num_points = int((stop.m_as('V')-start.m_as('V'))/step.m_as('V'))+1  
scan_range = np.linspace(start.m_as('V'), stop.m_as('V'), num_points)
```

We added a `+1` to the number of points to accommodate the last value. For example, if we wanted all the integer numbers from 0 to 10, we should use 11 data points. Again, this works fine if the number of points is an integer value, but as soon as the `step` we specified does not divide the interval in an integer number of points, we would have a problem. With the same values used before, we would get a scan range like this:

```
array([0. , 0.55, 1.1 , 1.65, 2.2 , 2.75, 3.3 ])
```

Which is not spaced by 0.5 V, but it respects the `start` and `stop` values. The last option to which we can resource is to change the config file. We thought that defining the `step` was a good idea, but perhaps it is a better idea to use the number of steps we want in our scan instead of the `step` size. Therefore, we have to update **experiment.yml** to include the number of steps and not the `step` size:

```
Scan:  
  start: 0V  
  stop: 3.3V  
  num_steps: 100  
  channel_out: 0  
  channel_in: 0  
  delay: 100ms
```

Before we proceed with the experiment class, we see that the array that numpy generates has no units, and this makes sense because we stripped the units from the parameters of the scan. But this can be easily solved by multiplying the array with the proper units, like the code below shows:

```
from time import sleep  
import numpy as np  
[...]  
  
def do_scan(self):
```

```

start = ur(self.config['Scan']['start']).m_as('V')
stop = ur(self.config['Scan']['stop']).m_as('V')
num_steps = int(self.config['Scan']['num_steps'])
delay = ur(self.config['Scan']['delay'])
scan_range = np.linspace(start, stop, num_steps) * ur('V')
scan_data = np.zeros(num_steps)
i = 0
for volt in scan_range:
    self.daq.set_voltage(self.config['Scan']['channel_out'], volt)
    measured_voltage = self.daq.get_voltage(self.config['Scan']['channel_in'])
    scan_data[i] = measured_voltage
    i += 1
    sleep(delay.m_as('s'))

```

The `do_scan` method is very complete now. We use the number of steps specified in the config file, we are also using the delay, and therefore we import `sleep` at the top of the file. Then we go through all the voltages. We also defined a variable called `scan_data` to hold the values as we measure them. It is a big achievement, and we need to reflect it the file we were using for testing the Experiment class. We can update **run\_experiment.py**:

```

[...]
experiment.do_scan()
experiment.finalize()

```

When we run the experiment, we encounter a problem:

```

...

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: setting an array element with a sequence.

```

It is because of how Pint and numpy work together. We can't simply put into a numpy array a quantity. However, solving this problem is very straightforward. In the same way we defined the `scan_range` as an array with units, we can define `scan_data` to also include units:

```

scan_data = np.zeros(num_steps) * ur('V')

```

And we can go ahead now and run the experiment without errors. After we do the scan we can finalize the experiment, but there is no way for us to actually look at the data that we acquired. To make the data available to the outside world, or to other methods in the same class, we can define attributes instead of just variables that get destroyed when the method finishes. The `do_scan` method can then look like this:

```

def do_scan(self):
    start = ur(self.config['Scan']['start']).m_as('V')
    stop = ur(self.config['Scan']['stop']).m_as('V')
    num_steps = int(self.config['Scan']['num_steps'])
    delay = ur(self.config['Scan']['delay'])
    self.scan_range = np.linspace(start, stop, num_steps) * ur('V')
    self.scan_data = np.zeros(num_steps) * ur('V')
    i = 0
    for volt in self.scan_range:
        self.daq.set_voltage(self.config['Scan']['channel_out'], volt)

```

```

measured_voltage = self.daq.get_voltage(self.config['Scan']['channel_in'])
self.scan_data[i] = measured_voltage
i += 1
sleep(delay.m_as('s'))

```

We altered both `scan_range` and `scan_data`, so we can now go back to the `run_experiment.py` file and print the values we acquired:

```

[...]

experiment.do_scan()
print(experiment.scan_range)
print(experiment.scan_data)
experiment.finalize()

```

Finally, we can see the data we acquired after the scan runs. It would allow us to do plenty of things like saving, plotting, analyzing. But we should not go ahead of ourselves at this stage. We had laid out a plan for the `Experiment` class, and we should follow it as much as we can. The strategy of adding `self.` before a variable is beneficial, and this is the real power of objects. The idea is that we should use objects when we need to maintain *state*. It is not just a function that runs and returns a value, but it is updating the *state* of the experiment, or the device.

Now we can proceed further to the last missing step: saving data.

## 6.6 Saving Data to a File

After we perform a scan, we must save both the data and the *metadata*. With metadata we mean information on the parameters used to perform an experiment and that would allow us or anybody else to repeat the measurement. Let's start by the beginning, just saving data to a file, we are going to use the built-in functions of numpy for this, because we have to save two arrays: `scan_data` and `scan_range`. And since we already know that Pint and Numpy interact in special ways, we anticipate the errors that may appear:

```

def save_data(self):
    data = np.vstack([self.scan_range, self.scan_data]).T
    header = "Scan range in 'V', Scan Data in 'V'"
    filename = self.config['Saving']['filename']
    np.savetxt(filename, data.m_as('V'), header=header)

```

The method above works. We can go ahead and run the experiment, and it generates a file with two columns (that is the reason for the `vstack` and the `.T`) and a header specifying that the data is in units of Volts. However, if we run the experiment for a second time, the data gets overwritten. Moreover, we are saving the data to the same folder where we run the experiment, and this is a terrible idea, we would like to save the data in a dedicated place. Therefore, we need to update the config file first, and then the saving method. The folder we are using for saving data is only an example, and one should change it according to their needs:

```

Saving:
filename: data.dat # Files won't be overwritten, but renamed as data_001.dat,
↳ etc.
folder: /home/aquiles/Data

```



Something handy when saving data is to organize it by dates. We can create a folder with the date of the measurement inside the *Data* folder. First, we see snippets to understand what we need, and then we add the modifications to the method itself. To get the date of today as a string, we can combine the `datetime` package and the formatting of strings:

```
>>> from datetime import datetime
>>> print(datetime.today())
2020-04-13 12:22:34.895038
>>> print(f'{datetime.today():%Y-%m-%d} ')
2020-04-13
```

The first part is there, we know how to get the folder name based on today's date. Now we have to create the folder if it does not exist:

```
>>> import os
>>> data_folder = '/home/aquiles/Data'
>>> today_folder = f'{datetime.today():%Y-%m-%d} '
>>> saving_folder = os.path.join(data_folder, today_folder)
>>> os.path.isdir(saving_folder)
False
>>> os.makedirs(saving_folder)
>>> os.path.isdir(saving_folder)
True
```

Using the package `os` allows us to take care of the common problems that appear when dealing with directories. For example, Linux uses `/` to separate the structure, while Windows uses `.`. Also, we may define the folder with a trailing `/` or not. `os` takes care of all this complexity for us. We join the data folder and today's folder and check if it exists, if it doesn't, we create it. The `makedirs` also creates the missing parent directories. For example, if the *Data* folder does not exist, the program creates it.

Next, we have to be sure we will not overwrite the files when we save new scans. Ideally, we would like the files to look like `data_001.dat`, `data_002.dat`, etc. Using what we specified in the config file, we can achieve something like it:

```
>>> filename = 'data.dat'
>>> base_name = filename.split('.')[0]
>>> ext = filename.split('.')[-1]
>>> i = 1
>>> new_filename = f'{base_name}_{i:04d}.{ext}'
>>> print(new_filename)
data_0001.dat
```

It seems like much work just to format the name, but once we have it, we can use it in all our experiments. We only need to know which is the first available name and save data there. We are now ready to save the data, and also the metadata:

```
import os
from datetime import datetime
[...]

def save_data(self):
    data_folder = self.config['Saving']['folder']
    today_folder = f'{datetime.today():%Y-%m-%d} '
    saving_folder = os.path.join(data_folder, today_folder)
```

```

if not os.path.isdir(saving_folder):
    os.makedirs(saving_folder)

data = np.vstack([self.scan_range, self.scan_data]).T
header = "Scan range in 'V', Scan Data in 'V'"

filename = self.config['Saving']['filename']
base_name = filename.split('.')[0]
ext = filename.split('.')[-1]
i = 1
while os.path.isfile(os.path.join(saving_folder,
    ↪ f'{base_name}_{i:04d}.{ext}')):
    i += 1
data_file = os.path.join(saving_folder, f'{base_name}_{i:04d}.{ext}')
metadata_file = os.path.join(saving_folder,
    ↪ f'{base_name}_{i:04d}_metadata.yml')
np.savetxt(data_file, data.m_as('V'), header=header)
with open(metadata_file, 'w') as f:
    f.write(yaml.dump(self.config, default_flow_style=False))

```

The `save_data` method may be the most complex in the program, but it is also powerful. We start with a base folder and create a folder with the current date, with the format Year-Month-Day. We format the data to make it easy to store as two columns on a text file, and we also add a header to explain what we are storing. We extract the base part of the filename and the extension as two separate variables, and we format the filename in such a way that it can have a number appended to it. The syntax `i:04d` is the secret to formatting numbers with an appropriate number of 0 in front. We use a while loop to increase the counter until the first file is available, and use that value to store the data. We also create another file with the same number and a similar name, to store the metadata. In our case, the metadata is nothing more than the `config` dictionary.

## Saving Metadata

A convenient by-product of saving the config as a YAML file is that we can use it as the config file for the next experiment. If we want to repeat a measurement from the past, we just need to point the experiment to the config we saved from that date.

We can update the `run_experiment.py` file, and perform a complete measurement:

```

import sys
import os

base_dir = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
sys.path.append(base_dir)

from PythonForTheLab.model.experiment import Experiment

experiment = Experiment('experiment.yml')
experiment.load_config()

experiment.load_daq()

experiment.do_scan()

```

```
experiment.save_data()
experiment.finalize()
```

Except for the part of appending the folder to the system path, we can run a relatively complex experiment in less than 10 lines of code, including saving data and metadata. Being able to reduce a complex problem to a relatively simple flow is thanks to the effort we put into defining the proper methods in the experiment and device models.

## 6.7 Conclusions

With this chapter, we have finished the core code for a functioning experiment. Defining an Experiment model is one of the best strategies to simplify the work needed to perform a measurement. Once the complicated bits of code, such as doing a scan or saving data are in place, our running script just needs one line of code. The model takes care of rest automatically. If you share the code with a colleague and you show them the `run_experiment` script, they will be able to perform their measurements in no time.

We still left some more things that we should do in our experiment. For example, we are still acquiring voltages instead of currents. We are not going to explicitly solve this problem because it is an excellent exercise to start thinking by yourself.

### ? Exercise

Update the config file to include information on the resistance that we are using. With that value, update the `do_scan` method to acquire amperes (or milliamperes) instead of volts. Finally, you need to update the save method to accommodate for the changes, not only in the header but also in how we transform the arrays to unitless before saving them.

Some other topics which are still relevant and that we are going to cover on the next chapter include how to change the parameters of the scan directly from within the running script. It would allow us to, for example, scan the voltages with varying delays to understand if there is some form of hysteresis or scans with an increased resolution around specific features. We have also neglected what would happen if someone forgets to load the config file or the daq before starting a scan, for example.

The final missing bit is how to stop the experiment while it is running. If we see something is going wrong, there is no way of deciding to stop it without losing data. There are a lot of different approaches to solving this. The one we are going to explore is based on *Threads*, a handy tool for developing acquisition software, but that may have a somewhat high entry barrier for newcomers to Python.



# Chapter 7

## Run an experiment

### 7.1 Introduction

In the previous chapter, we have seen how to develop an experiment model and how to use it from a simple script. In this chapter, we are going to explore how to bring it to the next level. So far, we can run the experiment only with the parameters specified on the config file, but we are not limited to them. Moreover, once the scan starts, there is no way of stopping it unless we completely stop the program. That leads to missing data and is not particularly helpful.

Following the *Onion Principle*, it is now time to bring the experiment model to the next level.

### 7.2 Running an Experiment

In the previous chapter, we have seen how to run an experiment from a script. Each of the steps needed is triggered one after the other. We called that script **run\_experiment.py** and had the following code (skipping some less relevant lines):

```
from PythonForTheLab.Model import Experiment

experiment = Experiment('experiment.yml')
experiment.load_config()

experiment.load_daq()

experiment.do_scan()
experiment.save_data()
experiment.finalize()
```

It is relatively straightforward, but not the only thing that we can do with the experiment. We can access the attributes of the experiment class while we are performing a measurement. For example, we could show the data once the scan finishes:

```
experiment.do_scan()
print(experiment.scan_data)
```

But we could also change the parameters of the scan after loading the config file. Sharing information with classes in Python is always two ways: we can read from them, but we can also write to them. Let's see what happens when we change the `config` attribute of the experiment before triggering the scan. We can do something like the following:

```
[...]

experiment.config['Scan']['num_steps'] = 5
experiment.do_scan()
experiment.save_data()
experiment.config['Scan']['num_steps'] = 10
experiment.do_scan()
experiment.save_data()
experiment.finalize()
```

We can change any of the values stored in the `config` of the experiment at any time. We can change them from the running script, for example. But we can also change them from the Python interpreter. For people familiar with Jupyter notebooks, for example, the experiments can be run directly from within them. There are some extra benefits of using notebooks, such as the possibility of documenting the work, but we leave the discussion for the Advanced Python for the Lab book.



## BPython

Running from the Python interpreter usually is cumbersome. There is another version of the interpreter called **bpython** which has auto-complete functions, as well as it allows you to edit code before running it. We don't encourage people to run experiments from the interpreter, but some times it is the handiest solution to change values when desired.

In the code above, we have changed the value of the number of steps before triggering a scan. First, we have set it to 5 steps and then to 10 steps. We can check the metadata, and see that this information was saved. If we open the data file, we would also see a different number of data points. This flow proposes exciting possibilities. For example, if we would like to see if there is any form of hysteresis on the measurements, we could vary the delay between data points.



## Exercise

Write a for-loop that changes the delay between data points after each scan. Remember to save the data so you can explore the differences

If we start developing complex logic in the running script, it is fair to wonder if it is better to put all that logic into the experiment class. As always, when we do things only once, knowing they'll be one-off tasks, then it is quicker to run from a script. If, however, that one-off becomes an essential part of our experiments, or we believe that others could benefit from doing the same measurement, then we should implement the code in the class. As always, common sense is the best ally for a scientific developer.

One of the important things to note is that when we work with units, we leave the values as a string, and only transform them into real quantities when we need them. If we explore the `do_scan` method, we would see that we start by running the parameters through the unit registry. It means that if we want to change the values of, for example, the start, stop, or delay, we only need to change a string:

```
experiment.config['Scan']['start'] = '2.2V'
experiment.config['Scan']['delay'] = '200ms'
```

## 7.3 Plotting Scan Data

We have learned how to do several things with the device, but we are not doing anything with the data after we save it. Being able to see what we are doing through a plot seems a lovely addition to our workflow. For plotting, we are going to use a library called PyQtGraph, which was developed mainly to generate fast data visualizations. Exactly what we need when we are dealing with experiments. Usually, people a bit more familiar with Python get acquainted with matplotlib, which is an excellent tool for paper-ready plots, but a bit slow for real-time visualization. In any case, at this stage, we don't care about speed, therefore if you are more confident with another tool, feel free to adapt the code to use them. In any case, PyQtGraph comes back later, when we build the user interface.

Making a simple plot with PyQtGraph is very easy. We can add the following after we perform the scan, in the **run\_experiment.py** script file:

```
import pyqtgraph as pg

[...]

pg.plot(experiment.scan_range, experiment.scan_data)
```

The caveat with this solution is that we must run the program slightly differently:

```
python -i run_experiment.py
```

We must append the `-i` or the plot will close right after it appears on screen. It is simple, but it works. PyQtGraph also gives us some mouse interactions out of the box. We can drag the image, we can zoom in and out with the mouse wheel, we can also transform the scale of the axes by right-clicking. We can make the plot a bit better looking by adding labels and a title:

```
PlotWidget = pg.plot(title="Plotting I vs V")
PlotWidget.setLabel('bottom', f"Channel:
↳ {experiment.config['Scan']['channel_out']}\"", units = "V")
PlotWidget.setLabel('left', f"Channel: {experiment.config['Scan']['channel_in']}\"",
↳ units = "V")
PlotWidget.plot(experiment.scan_range, experiment.scan_data)
```

Plotting with PyQtGraph is relatively simple. However, we can only plot data after the scan finishes. The `do_scan` method takes relatively long to complete. When a method, or a function, work in this way, they are called blocking functions. The program can't continue until they finish. If we would like to monitor the progress of the experiment while running, we need to find a way of making the scan in a non-blocking way.

## 7.4 Running the scan in a nonblocking way

Methods or functions like `do_scan` take a long time to run. Imagine if we would like to acquire a movie for one hour, the function would take one hour to complete. If we would be running a

very complex simulation or data analysis process, functions can also take very long to complete. However, both scenarios are different from each other. Sometimes functions take long to execute because they are computationally expensive. They need many cycles of the processor to finish. Sometimes they take longer to execute because the process is slow. When we do a scan, the program waits in a `python` sleep or waits for the device to read a signal. Waiting is not computationally expensive, which means that our computer can perform other tasks at the same time if we know how to do it.

Python has different options for achieving what we are after. Still, the one that gives the best results, not only in terms of flexibility but also in terms of simplicity to implement, is the *multithreading* module<sup>1</sup>.

### 7.4.1 Threads in Python

All threads have two ends. A computer program always starts going through a thread in the same direction, until it reaches its end. Sometimes, along the thread, a task blocks the progress, and the program halts there for some time. What Python allows us to do is to start other threads at any point in time. The more threads we have, the more entangled the program would be. Python, however, does not run the threads at the same time. There is a tool called the Global Interpreter Lock (or GIL), which prevents two things to happen simultaneously.

What Python does is run small pieces of each thread one after the other. Someone once compared it to a super-efficient secretary, who pushes jobs according to who has free time and who has tasks to perform. It means that two computations won't happen exactly at the same time, but if one thread is, for example, waiting on a `sleep`, Python can go and let another thread run. Whenever we start Python, we are starting a thread, also called *main thread*, that lives through our program. From that thread we can start a second one, as we do in the code below:

```
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 we run it, we get the following output:

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

It is essential to understand what is happening in the code above, so we can go step by step, dissecting it. If we run the function `func` on its own, we see that the numbers appear one by one.

---

<sup>1</sup>Another library which is gaining popularity is AsyncIO. It is, however, harder to implement efficiently for our purposes



We would also notice that while the function is running, nothing else would happen. However, in the code above, we can see a `'There'` printed in between the output of the function. It means that we managed to trigger the function, and it didn't stop the execution of the rest of the program.

When we used `threading.Thread`, we were creating a new child thread. The `target` is whatever function we want to run on that thread. We can also pass arguments, as we did, using `args=(3, )`. Once we created the thread, we have to start it by doing `t.start()`. Note that the `target` is `func` and not `func()`. If we would use `func()`, we would be using the result of the function, and not the function itself. Therefore, there wouldn't be much to gain from the thread.

## ✖ Functions or Function Calls

It is very important to distinguish the proper function from its output. We can put into running on a separate thread a function, and therefore we use `target=func`. If we call the function, we would be sending its output to a thread, which is not useful. If things don't work, check whether there is an extra pair of `()` after `func`.

Another common pitfall is forgetting the `start()`. When we create a new thread, the function is not called, but it waits until the `start()` signal is triggered. That is why we first see `'Here'` being printed, then some steps. With such a simple example, it is always a good idea to change the parameters to see how they affect the output printed to screen, what happens first, second.

We can also have several threads running the same function at the same time. In the example below you can see how easy it is:

```
first_t = threading.Thread(target=func, args=(3, ))
second_t = threading.Thread(target=func, args=(3, ))
print('Here')
first_t.start()
second_t.start()
sleep(1)
first_t.join()
second_t.join()
print('There')
```

Starting two threads is as simple as starting one. We have added one extra detail to our code, we have used `join()` to wait for the threads to complete. Now we can see that the program prints `'There'` after the function finishes. Using `join()` is usually a good idea in order not to finish the program and leave some orphaned threads.

## i Threads don't run in parallel

The fact that things can happen more or less at the same time does not mean our code is running in parallel. Threads have been around for a very, very long time. Even in single-core computers, we were able to switch from one program to another. We could get e-mails while browsing the internet. It was the operating system taking care of executing bits of each program to keep them all up to date. Parallelizing means running computations at the same time. In multi-core processors like the ones available in most computers and cell

phones today, we can split tasks and run them at the same time in different cores. For really running on different cores, Python offers the *multiprocessing* package. However, multiprocessing is challenging. The Advanced Python for the Lab covers in detail how to leverage it for more extensive and more data-consuming experiments.

## 7.5 Threads for the experiment model

We have seen how to run a straightforward function on a different thread. But we can also see that more complex functions do not present a challenge. Going back to the experiment model, we can already test what we have learned by running the `do_scan` method on its thread. Since the method doesn't take any arguments, the code would simply look like this:

```
t = threading.Thread(target=experiment.do_scan)
t.start()
```

While the scan is running in its thread, we can do other things in the main thread, such as plotting. We need to refresh the plot while the acquisition is happening, and therefore we need to update the plot within a loop. Combining what we have seen so far, we can update the `run_experiment.py` file:

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

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

PlotWidget = pg.plot(title="Plotting I vs V")
PlotWidget.setLabel('bottom', f"Channel:
↳ {experiment.config['Scan']['channel_out']}\"", units = "V")
PlotWidget.setLabel('left', f"Channel: {experiment.config['Scan']['channel_in']}\"",
↳ units = "V")

while t.is_alive():
    PlotWidget.plot(experiment.scan_range, experiment.scan_data, clear=True)
    pg.QtGui.QApplication.processEvents()
    sleep(.1)

experiment.finalize()
```

The beginning of the code is the same. The only difference is that we trigger the scan inside its thread. After starting it, we create a plot exactly in the same way we did before. The important part is the `while` loop. First, we check whether the thread is still running with the `is_alive()` method. If the thread is not running, it means the scan has finished. Then, within the loop, we update the plot with the data available in the experiment class. It is crucial to note that even if the `do_scan` method is running in a different thread, its data is accessible from the main thread. The

extra line with the `QApplication` is necessary to make things work, but not important, and since we are not going to follow this approach, we do not discuss it further.

Exchanging information between threads is a topic that we need to handle with care. Right now, we have two threads: the main thread, in which we define the experiment and the plot, and we have a child thread, in which the scan is happening. However, for a scan to happen, the thread needs to have a copy of the experiment itself. It turns out that it is not just a copy, but the experiment is the same object on both threads. That is why, if the child thread modifies the values of `scan_data`, for example, the main thread can see them. But data is not the only thing shared. The experiment holds communication with the PFTL DAQ device. This communication is open from both the main and the child threads. Nothing prevents us from triggering two scans at the same time:

```
scan1 = threading.Thread(target=experiment.do_scan)
scan2 = threading.Thread(target=experiment.do_scan)
scan1.start()
scan2.start()
```

If we do something like this, we see the inherent problem of working with threads carelessly. Each scan has a for-loop, in which the output voltage changes to a given value, and then it reads a voltage. In the best-case scenario, when one thread sets the value, the other changes it, and the voltage read corresponds to the second one. In the worst-case scenario, the information transmitted to and from the device gets split. It means that the bytes transmitted to and from the device can end up intercalated. It corrupts the data and can lead to crashes or the device going beyond the specified range of voltages.

We are not going to enter into the details of all the possible solutions to prevent these problems from appearing. The threading package has many tools to make our programs *thread-safe*. However, we also have to find a balance between how complex we want to make our solution and how much we can trust that we won't make these kinds of mistakes. We use a straightforward approach that prevents us from triggering a second scan if one is already running. That is a likely scenario if we don't realize a scan is taking place. The solution is relatively easy: when the scan is running, we set a variable to `True`. Every time we want to start a new scan, we check the variable and prevent the program from going further if a scan is already happening. We must edit the `Experiment` class:

```
class Experiment:
    def __init__(self, config_file):
        self.is_running = False # Variable to check if the scan is running
        [...]

    def do_scan(self):
        if self.is_running:
            print('Scan already running')
            return
        self.is_running = True
        [...]
        for volt in self.scan_range:
            [...]
        self.is_running = False
```

We have removed all the code that didn't change to keep it shorter. It is very important to define the `self.is_running` attribute in the `__init__` because if we don't, the program crashes the first time we try to run a scan. Then, we check if the scan is already running. If it is, we print a message and stop the execution by using a `return`. This pattern is convenient to avoid using a very long if-else block. Then we switch the attribute right before starting the loop and back to false

when it finishes. It should be enough to prevent two threads from running a scan at the same time. We can go ahead and re-run the script to see that this time we get a nice message warning us that a scan is already taking place.

## ✖ Thread safety

People experienced with threads may dislike the solution above, and they are right to do so. There is a chance that both threads check if the scan is running one precisely after the other, and then both see it is not. Then both threads set the safety variable to `True`, and we face the same issues as before. This situation can happen if we trigger two threads to do the scan one right after the other. In practical terms, however, we trigger the scan by clicking on a button, or by typing on the Python interpreter, and this is never quicker than checking whether a variable is true or not. However, as programs grow in complexity, these concerns can become real issues<sup>a</sup>

<sup>a</sup>We have written an extensive tutorial on threading on our website, feel free to check it out to learn more.

There is only one extra feature that our `do_scan` method is missing: the ability to stop whenever we want. We have seen that the main thread can read data stored in the experiment class even if a child thread generates this data. But also, the child thread can see the changes to the attributes of the experiment class. Therefore, we can use an attribute, similar to the `is_running`, that signals that we want to stop the scan. We can modify the `do_scan` again:

```
def do_scan(self):
    [...]
    self.keep_running = True
    for volt in self.scan_range:
        if not self.keep_running:
            break
```

When we start the scan, `keep_running` is set to `True` because we want to keep running the scan. Then, in every iteration, we check whether this variable changed or not. If it is `False`, then the loop would stop. We can see how this would work in the `run_experiment.py` script:

```
[...]
scan1 = threading.Thread(target=experiment.do_scan)
scan1.start()
sleep(2)
experiment.keep_running = False
print('Experiment finished')
experiment.finalize()
```

The example is relatively straightforward. We start the scan, wait for two seconds, and then we stop it. It is not a particularly useful situation, but it is crucial for cases when we want to be in control and not lose data nor damage the equipment.

## ? Exercise

Use the input from the keyboard to stop the scan. The best approach is to wait while the thread is alive, such as we did for plotting. Then, you can use a try/except block, using the KeyboardInterrupt exception.

## 7.6 Improving the Experiment Class

Through the book, we have always come back to the *Onion Principle*. Every time we find something that we believe can be useful in the future, we don't leave it as an example on a Python script, but we try to implement it in a robust way. It is what we did with threads. We have seen that we can run the scan without blocking the program. But to achieve it, we have to remember how to work with threads. A better idea would be to implement the threads directly in the Experiment class.

In this case, we can write a new method that takes care of starting the scan in a separate thread and another method just for stopping. Having specific methods is a perfect way of not having to remember what attributes do what. Let's create the methods for starting and stopping the scan, directly on the Experiment class:

```
import threading
[...]
```

```
def start_scan(self):
    self.scan_thread = threading.Thread(target=self.do_scan)
    self.scan_thread.start()

def stop_scan(self):
    self.keep_running = False
```

And now we can update the **run\_experiment.py** script to make it look much better:

```
experiment.start_scan()
while experiment.is_running:
    print('Experiment Running')
    sleep(1)
experiment.finalize()
```

This code is clean and easy to understand. One of the advantages is that it also gives us the freedom to decide whether we want to run the scan on its thread or not. There is only one more detail and is that if we finalize the experiment while the scan is running, then we may face some issues trying to read from a closed device. It is better to update the finalize method:

```
def finalize(self):
    print('Finalizing Experiment')
    self.stop_scan()
    while self.is_running:
        sleep(.1)
    [...]
```

We stop the scan, and then we wait until we are sure it has finished. It is important because the delay between data points may be significant, or because acquiring data takes long, such as what happens with long exposure times of cameras. Once we know the scan stopped, then we continue to close the communication.

### 7.6.1 Threads and Jupyter Notebooks

Running experiments on Jupyter Notebooks can be a perfect solution to combine the generation of data, its documentation, and its analysis in only one tool. If appropriately used, Jupyter notebooks can be an excellent resource for the experimentalist. However, real-time plotting of data within notebooks is virtually impossible. Building interactive tools on top of a notebook are very complicated, and that is why we chose to follow a different path for the last chapters.

However, for people who are already using Jupyter, it is worth mentioning that how we developed the Experiment and Device models makes them readily available to be incorporated into a notebook. Moreover, the use of threads directly built in the class allows us to run the scan in one cell and simultaneously plot the data on another cell. It can help prototype programs very quickly and can also be the starting point for plugging an experiment directly to the data analysis pipeline.

## 7.7 Conclusions

This chapter aims at polishing some of the details that we were missing to be able to perform a scan in a robust way. The most exciting aspect of the chapter is the inclusion of threads to be able to run a scan and still maintain control of the program. We quickly saw how to plot while the scan runs, and took a look at the problems that can appear when we run multiple threads. We explored some new strategies to prevent a second scan from starting, and for stopping the scan without the risk of losing data.

Threads open a lot of different possibilities for Python developers, not only for lab applications. They are, however, a complex topic that we need to take seriously. The pattern we decided to follow in this chapter, exchanging information between main and child threads using attributes of a class is straightforward, but also prone to problems. We believe that in the context of controlling an experiment, there is rarely the need to go beyond what we have done. It does not mean that we shouldn't keep an eye in case problems arise.

# Chapter 8

## Getting Started with Graphical User Interfaces

### 8.1 Introduction

Building User Interfaces may seem more complicated than what it is. Once you get the fundamentals, you can see that it is possible to put something together very quickly, especially if we already have some code to which to relate. In the previous chapter, we saw that we can control an experiment from the command line. It is possible to ask ourselves why going through the trouble of a user interface. In some cases, it can be very not only handy to control the parameters of an experiment and to monitor the output in a window especially designed, but it can become necessary to monitor the progress in real-time to make decisions while the experiment runs.

There are several options for building GUIs with Python. And in the community, there is no clear consensus on what is the best path to follow. For scientific applications, however, the only library which is powerful enough to achieve what we want to achieve is 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. Qt itself is a Finnish company with a long trajectory. It was part of Nokia for a while, and now they are publicly traded in the Helsinki exchange. It means that Qt is going to be around for a long time.

In this chapter, we are going to give the first steps for building a user interface using Qt, and the Python wrapper called *PyQt*, that we installed in Chapter 2. At Python, for the Lab, we have traditionally adopted PyQt, but in the past years, Qt itself took over the project called PySide, which is another wrapper for Qt. They are licensed under different open-source terms, and both are excellent. However, the structure of the PySide2 package is different from the PyQt package, and, for consistency, we keep using PyQt.



#### **Qt, PyQt, and PySide licensing**

If you are planning to release commercial software, or if you are packaging Qt, PyQt or PySide2 into your application, you should explore the different licensing options available.

## 8.2 Simple Window and Buttons

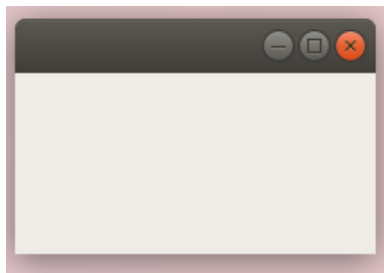
Now it is finally time to start using the empty folder from the M-V-C design pattern: the **View**. We start by learning how to create simple windows directly with Qt and proceed to a fully-featured user interface for our experiment. We start just with scripts, and we slowly grow in complexity.

The best way to get started with Qt is with a quick example. We can create a new file, **simple\_window.py** in the *Examples* folder, with this code:

```
from PyQt5.QtWidgets import QApplication, QMainWindow

app = QApplication([])
win = QMainWindow()
win.show()
app.exec()
```

We will come back to the code above over and over again. In the beginning, it is hard to remember, but once we do it often enough, it sticks. After importing, we create a `QApplication`, a `QMainWindow`, we show it, and we run the app. This code should produce a very simple window that looks like the image below:



The style matches the operating system where it runs. It is a simple, empty window. However, we can already start understanding how Qt works. A user interface is a program that keeps running in a loop. When we click and drag to resize a window, for example, there is always a program responsible for knowing how to do it. In Qt, this never-ending loop is the `QApplication`. Whatever window we want to create needs to belong to an application, and that is why the first thing we did was defining `app`.

In the following line, we define a new object, called `win`, which is a `QMainWindow`. As the name suggests, the main windows are the core of the user interface. From the main window, we can open dialogs, other windows, but the main window is central to our program. After creating it, we show it. The last line is where the application loop starts. The `app.exec()` command is blocking. Therefore nothing that comes after is executed until we finish with the user interface.

### ? Exercise

To understand a bit better what is going on with the user interface, you are encouraged to try different things. For example, what happens if you don't show the window or add a few print statements to see when they get executed. You can also try to define the window before the application.

Having an empty window is not particularly useful, so we can start adding elements to it. First, we add a title to the window, like this:



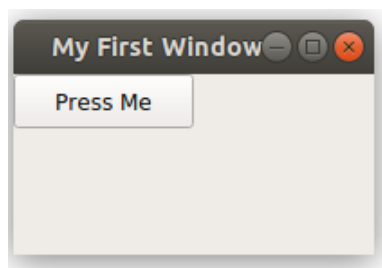
```
win.setWindowTitle('My First Window')
```

Very slowly, our program starts taking shape and looking more professional. We can also add an interactive element, such as a button. We can define one like this:

```
from PyQt5.QtWidgets import QApplication, QMainWindow, QPushButton

app = QApplication([])
win = QMainWindow()
win.setWindowTitle('My First Window')
button = QPushButton('Press Me', win)
win.show()
app.exec()
```

Which will produce a small window, like this:



Notice that when we defined the button, we added a second argument, `win`. Qt has a hierarchical structure, where each element is called a *widget*. We have imported three widgets so far: the application, the window, and the button. All widgets live inside the application loop, but we have to establish the relationship between them. By passing the window as the second argument, we are explicitly saying that the button belongs to the window.

### ? Exercise

Remove the `win` from the definition of the button, and see what happens

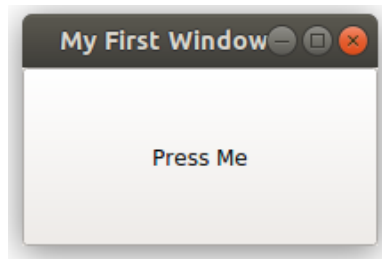
### ? Exercise

Alter the order and make the button the parent of the main window, does this work?

A big part of working with Qt is finding out how to relate different widgets to each other, how to position them. `QMainWindows` are special because they must hold widgets within them. That is why they specify a method to determine which Widget is the most important for the window, or in Qt jargon, which Widget is the central Widget. We can explicitly declare it:

```
button = QPushButton('Press Me')
win.setCentralWidget(button)
```

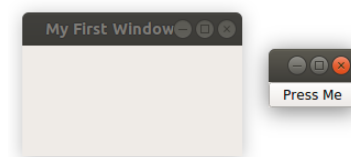
We removed the `win` from the declaration of the button, but if it's there it doesn't change the behavior. The window now looks somewhat different:



You can try resizing the window, and you see that the button scales. By declaring the button as the central Widget of the window, we made the relationship even stronger. The last possibility which is worth mentioning before moving forward is that we can also show the button independently from the window, like this:

```
win.setWindowTitle('My First Window')
button = QPushButton('Press Me')
win.show()
button.show()
```

In this case, the window and the button are two independent components, like we show in the image below. For the program to finish, we must close both the button and the window.



Qt offers a great deal of flexibility, which doesn't mean we need actually to use it. Having buttons floating around the screen does not sound like a good idea, but it is a possibility in case we ever need it.

Now that we have a button on a window, we are craving to do something with it.

## 8.3 Signals and Slots

Qt offers a programming pattern known as *Signals* and *Slots*. The core idea is that different actions on a user interface trigger a signal. For example, moving the mouse over an element triggers a signal. This signal is then caught by *slots*, which do something with the information provided. When we move the mouse over a button (this is also known as hovering), its background changes color. It is a clear example of the signal/slot paradigm.

Every Widget that we can place on screen has a myriad of signals. From interactions with the mouse to changes in shape or size triggered by reshaping the window, to time-based signals. It does not mean we need to know them all, nor that we use them all. But once we understand the pattern, we know where to go and find what we are after. The button has one signal called, very eloquently, `clicked`. To use it, we must define a function that can be called every time the signal fires. This function is the *slot*. We can expand our example code like this:

```
def button_clicked():
    print('Button Clicked')

[...]
button = QPushButton('Press Me')
button.clicked.connect(button_clicked)
win.setCentralWidget(button)
```

We can see that every time we click the button, the program prints a message to the screen. It is very important to note that we used `button_clicked` and not `button_clicked()`. It is the same that we discussed in Section 7.4.1 when discussing multithreading. We must use the function itself as a slot, and not the outcome of the function.

### 8.3.1 Start a Scan

With what we have done so far, triggering a scan from the user interface becomes almost trivial. For the time being, we can keep working on the *Examples* folder, but this time let's create a new file called **start\_gui.py**. We only need that when we press the button, a scan starts. We need to mix what we already have in **run\_experiment.py** with what we have done above. It can look like this:

```
from PyQt5.QtWidgets import QApplication, QMainWindow, QPushButton

from PythonForTheLab.Model.experiment import Experiment

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

app = QApplication([])
win = QMainWindow()
win.setWindowTitle('My First Window')
button = QPushButton('Start Scan')

button.clicked.connect(experiment.do_scan)

win.setCentralWidget(button)
win.show()
app.exec()

experiment.finalize()
```

The code above is a merge between what we did in the previous chapter and this one. We define the experiment as always, and the window and button as we have just learned. However, the line that does all the magic is this one:

```
button.clicked.connect(experiment.do_scan)
```

We can try the program. When we click the button, a scan starts. However, there is something else happening. The window freezes, we are not able to reshape it, close it. In some cases, especially on Windows, the program crashes, and we get a message saying whether we want to report the issue.

## ? Exercise

Can you guess why the window freezes?

When we described the flow of a Qt program, we talked about a loop taking care of the interactions within the program. However, if we trigger a scan using `do_scan`, we are going to block that loop. Both Qt and Python are single-threaded applications by default, and when one blocks, the other blocks as well. And by talking about single-threaded applications, we gave a hint to how this can be solved.

At the end of the last chapter, we developed a different method called `start_scan` that creates a separate thread to hold the scanning, effectively releasing the main thread to do other tasks. We can change just one line of code and achieve a very different behavior:

```
button.clicked.connect(experiment.start_scan)
```

We have developed a somewhat functional program. We have a window with a button from which we can control our experiment. It is already quite an excellent achievement. We also got some extra features out of the box, such as preventing the user from triggering two scans at the same time.

Sometimes, the easiness of developing this kind of solution misguides the readers. It was so easy to achieve what we achieved so far because we spent a lot of time and effort developing a proper *experiment class*. The threading, the checks to prevent two scans, and some extra things that keep appearing in this and next chapter are thanks to a well-designed model.

## 8.4 Extending the Main Window

We have seen how to get started by creating the main window and adding a button to it. However, we can also start seeing that if we try to add more elements, the code is going to become more and more convoluted. It would be a nice addition if the window we design here could be used for different purposes as well. As we have already seen many times, a good idea when we want to make blocks of code reusable is to convert them into classes. Qt is ideally suited for this because every Widget they provide is an object with a special inheritance tree.

In the folder *View* we can create a file called **main\_window.py**, and we can add the following code:

```
from PyQt5.QtWidgets import QMainWindow

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('My First Window')
```

Before discussing what we have done, we can quickly go back to **start\_gui.py** and change the following two lines of code:

```
win = QMainWindow()
win.setWindowTitle('My First Window')
```

with this one:

```
win = MainWindow()
```

We should also remember to change the imports at the top of the file by these:

```
from PyQt5.QtWidgets import QApplication
from PythonForTheLab.View.main_window import MainWindow
```

If we run the code, we see that it behaves as it was behaving previously. In our code, we create a new class called `MainWindow`, which in turn inherits from `QMainWindow`. It is always important to call `super()` because that runs the init method from the `QMainWindow` itself, setting up all the parameters, signals, properties that we need to generate a window. There is, however, a difference with a plain `QMainWindow`, we specify its title. Effectively, we have now extended the pure `QMainWindow` class to include a title by default.



### Naming conventions

It is a personal preference when I start developing a program that has only one main window to name it `MainWindow`, removing the preceding `Q`. It can lead to mistakes if we overlook the small difference in both names. Depending on taste, an alternative is to call the windows by what they are supposed to do, such as `ScanWindow`. It depends on the reader's preferences.

We can add the button, and the slot, like this:

```
from PyQt5.QtWidgets import QMainWindow, QPushButton

class MainWindow(QMainWindow):
    def __init__(self, parent=None):
        super().__init__(parent=parent)
        self.setWindowTitle('My First Window')
        self.button = QPushButton('Press Me')
        self.setCentralWidget(self.button)

        self.button.clicked.connect(self.button_clicked)

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

We can test this code again and see that we have recovered what we had before. Every time we press on the button, a message appears on the screen. We can also go one step further and start thinking about how to work with the experiment itself. The window is not aware of any experiments, but we would like to be able to trigger a scan if we press the button. Therefore, the experiment has to come from outside of the class and be stored within.

We have already done something like this. When we developed the driver in Section 3.6, we could send the port number to the class through the `__init__` method. We did the same for the device model in Section 5.2, and for the experiment model in Section 6.2. In all those cases, we were using simple strings, but we are not limited to them. Arguments of methods, or any function for the matter, can be complex objects as well.

We can adapt the `MainWindow` to accept an experiment as argument, store it as an attribute and use it when we need to. The code would look like this:

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

    [...]
    def button_clicked(self):
        self.experiment.start_scan()
        print('Scan Started')
```

The changes to the code were minimal but significant. We have included `experiment=None` in the init. We have provided a default value for the experiment because this allows us to run the program even if we have no experiment defined. It is useful if we want to test how the window looks like quickly. However, as soon as we press the button, the program crashes. We have to update the `start_gui.py` script to accommodate for the changes:

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

app = QApplication([])
window = MainWindow(experiment)
window.show()
app.exec()

experiment.finalize()
```

We pass `experiment` directly to the window, and it takes care of the rest. We can now safely trigger a scan from the user interface. What is important to note is that once we reach this step, all the rest happens directly on the view. The script that we use to open the window stays unaltered. Even in much more complex programs, we use the same pattern<sup>1</sup>.

## 8.5 Adding Layouts

So far, our window holds only one button, and we set that button to be the central Widget of the window. This makes it virtually impossible to add any other button or object. Therefore, it is time to start sophisticating our user interface<sup>2</sup>. The basic building blocks in Qt are `QWidgets`, and Qt allows us to place widgets inside of widgets at our will. We also know that Main Windows require a central widget. Therefore, we can create a widget that holds two buttons: start and stop, and that Widget is the central Widget of the window. We use this opportunity to clean up the names we have used, to make them more descriptive as well, it is important to pay attention to all the changes made:

---

<sup>1</sup>See, for example, how PyNTA starts its user interface: <https://bit.ly/WindowExperiment>

<sup>2</sup>In this chapter we decided to go lower level, programming every feature, but in the next chapter we will see how to do it with the QtDesigner software, which will speed up the process

```

from PyQt5.QtWidgets import QMainWindow, QPushButton, QWidget

class MainWindow(QMainWindow):
    def __init__(self, experiment=None):
        super().__init__()
        self.experiment = experiment
        self.setWindowTitle('Scan Window')

        self.button_widgets = QWidget()
        self.start_button = QPushButton('Start', self.button_widgets)
        self.stop_button = QPushButton('Stop', self.button_widgets)

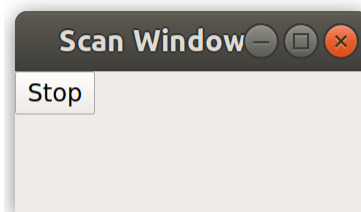
        self.setCentralWidget(self.button_widgets)

        self.start_button.clicked.connect(self.start_scan)

    def start_scan(self):
        self.experiment.start_scan()
        print('Scan Started')

```

If we run the program again, we will see a Window like the one below:



The stop button is visible, but not the start button. It happens because Qt has no way of knowing where we want to add the buttons and place them in the same position. The one that gets added later is on top.

## Exercise

Change the order in which we define the buttons and see that one or the other gets on top. If the button that is below has a much longer text, you can see it beneath the top one.

We could specify explicit coordinates for the positions of the buttons, but there is a much simpler approach using layouts. In Qt, there are 4 basic layout types: Horizontal, Vertical, Grid, and Form. With the first two, each time we add a widget, it is added either below or to the right. With the grid layout, we can control the position, width, and height based on a grid we define. The form defines two columns, ideally to hold some labels and inputs. We see more about layouts in the following chapter. For the time being, if we want to add two buttons, we can choose a horizontal layout. The code of the `MainWindow` takes a few extra lines to set everything up properly:

```

from PyQt5.QtWidgets import QHBoxLayout
[...]
self.button_widgets = QWidget()

```

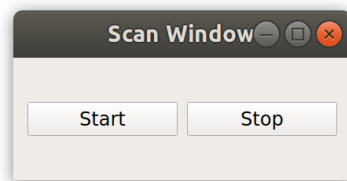
```

self.start_button = QPushButton('Start')
self.stop_button = QPushButton('Stop')
layout = QHBoxLayout(self.button_widgets)
layout.addWidget(self.start_button)
layout.addWidget(self.stop_button)

self.setCentralWidget(self.button_widgets)

```

In Qt, the horizontal layout is called `QHBoxLayout`, and we apply it to the `buttons_widget`. Then, we add the start and stop to the layout, instead of directly to the widget. This window will look much better:



If we resize it, we see that the buttons take the entire width, and they are always centered. It is already a good improvement compared to the simple window with which we started. Before we finish this section, these two exercises are a good way of practicing the skills acquired so far:

### ? Exercise

Change `QHBoxLayout` by `QVBoxLayout` to see the buttons stacked vertically.

### ? Exercise

Connect the stop button to a method that stops the scan

## 8.6 Plotting Data

We finish this section by adding a plot of the data in real-time. We already did something similar in Section 7.3. Parts of the code are very similar. Our window slowly starts getting more complex, with more elements. So far, we have two buttons stacked horizontally, but we would like to show the plot beneath the buttons, not next to them. One of the most natural solutions is to start stacking widgets, instead of making the `buttons_widget` the central Widget, we can make another one that contains the buttons and the plot.

```

import pyqtgraph as pg
from PyQt5.QtWidgets import (QMainWindow,
                             QPushButton,
                             QWidget,
                             QHBoxLayout,
                             QVBoxLayout, )

```



```

class MainWindow(QMainWindow):
    def __init__(self, experiment=None):
        super().__init__()
        self.experiment = experiment
        self.setWindowTitle('Scan Window')

        self.central_widget = QWidget()
        self.button_widgets = QWidget()
        self.start_button = QPushButton('Start')
        self.stop_button = QPushButton('Stop')
        self.plot_widget = pg.PlotWidget(title="Plotting I vs V")
        self.plot = self.plot_widget.plot([0], [0])

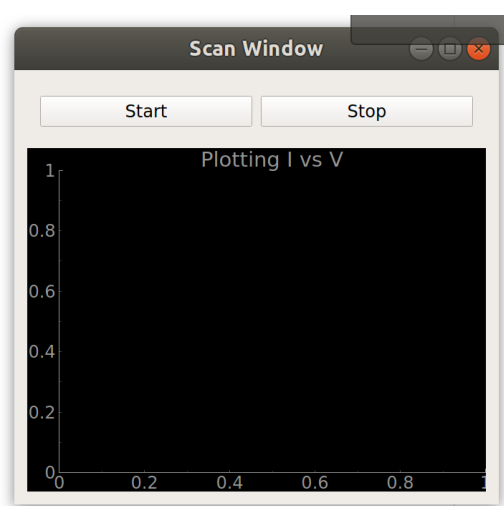
        layout = QHBoxLayout(self.button_widgets)
        layout.addWidget(self.start_button)
        layout.addWidget(self.stop_button)

        central_layout = QVBoxLayout(self.central_widget)
        central_layout.addWidget(self.button_widgets)
        central_layout.addWidget(self.plot_widget)

        self.setCentralWidget(self.central_widget)

```

Some remarks about the code before we run it. We are using `()` for the import because it makes it easier to stack the modules instead of having a very long line that becomes hard to read. In the window, we define three widgets now, `central_widget`, `button_widgets`, and `plot_widget`. The plot widget is very similar to what we did in the previous chapter. The only difference is that we store the Widget itself and the plot separately, and we explain why later. We didn't touch the buttons, but instead of adding them as the central Widget, we add them to a higher-order widget. By stacking the buttons horizontally between themselves, but vertically to the plot, we get a window that looks like this:



We are halfway through with what we wanted. If we resize the window, the plot changes, taking all the space available, even if we expand the window vertically, there is no gray area around the buttons. How the surrounding elements control the shape of a widget is one of the properties that can be specified.



## Qt options

We make several remarks about possibilities with Qt that we don't explore further. We just want to point out that every single thing that happens on a user interface was decided, and can be changed. Not only the aspect, such as colors but also how different elements relate to each other and change shapes when the container window changes.

To plot the data we acquire, we just need to update the plot periodically. Qt offers a special object called `QTimer` that also specifies signals. With timers, we can trigger periodic actions without interrupting the rest of the program. We also need to develop a method that can update the plot. The Main Window code looks like this:

```
from PyQt5.QtCore import QTimer
[...]

class MainWindow(QMainWindow):
    def __init__(self, experiment=None):
        [...]
        self.timer = QTimer()
        self.timer.timeout.connect(self.update_plot)
        self.timer.start(50)

    def update_plot(self):
        self.plot.setData(self.experiment.scan_range, self.experiment.scan_data)
```

The timer is relatively easy to understand. It is an object that triggers a signal, `timeout` periodically. We connect that signal to the method `update_plot`. When we start the timer, we need to specify the time interval in milliseconds, therefore 50 ms means a refresh rate of 20 Hz. The `update_plot` method is different from what we did in the previous chapter. Instead of using `plot`, we are using `setData`. There are two reasons for it. First, if we use `plot()`, we create a new plot on top of the existing one. We wouldn't be refreshing the data but drawing on top of it. After a while, especially if the parameters or results change, we would see several lines overlapping. The second reason is speed. `plot()` is a relatively slow method because several things need to be set up, such as the axes, labels, ticks. By using `setData` PyQtGraph automatically reuses the elements available.

However, if we try to run the code, we will get a problem:

```
AttributeError: 'Experiment' object has no attribute 'scan_range'
```



## Exercise

Find out why are we getting this error even though we didn't find it when running a more straightforward script in the previous chapter

When the window starts, we automatically start the timer, which, in turn, tries to update the plot. However, the experiment class does not have any `scan_range` nor `scan_data` until the experiment starts running. A bypass to the problem would be to start the timer after we have started

the scan, but this is very unreliable. Best-practices in Python indicate that we should always define attributes in classes in the `__init__` method. It means that as soon as we create the object, the attributes exist, even if with place-holder values.

When we developed the *Experiment* class, we completely neglected this practice. We added `self.` whenever we needed to have data available through the class and also from outside of it. We leave the definition of most of the attributes to the reader, but we show how to solve the problem with the scan. Going back to the experiment model, we need to add the following:

```
class Experiment:
    def __init__(self, config_file):
        [...]
        self.scan_range = np.array([0]) * ur('V')
        self.scan_data = np.array([0]) * ur('V')
```

We decided to define both attributes as numpy arrays holding only one value: 0 V. If we try to plot these results, we get a single point at the origin. It may raise other questions, such as whether it is better to have a 0 or a `None` value, because 0 V could be a valid measured value. It is left to the sensitivity of the reader to judge what is best in their specific case. For our purposes, this is enough to get the window running and showing a plot of the data in real-time once the scan starts.

### ? Exercise

Every attribute in any class should be defined in the init of that class. Go through all the models, and see whether there are attributes used but not defined at instantiation.

## 8.6.1 Refresh Rate and Number of Data Points

When we follow the strategy of using a timer for refreshing the plot, we can be tempted to increase the refresh rate to make the animations more appealing, but we have to be careful with this. On the one hand, if we are generating data at, let's say, 1 Hz, doesn't matter how fast we refresh the plot, it won't change faster than once per second.

Let's assume we are acquiring data much faster than once per second, perhaps at hundreds or thousands of new points per second. We have to consider how fast the screen of the computer can redraw the elements on it. Most screens work at 30 Hz, some may go to 60 Hz. Therefore, if we try to update the plot faster than that, we just waste computer power on something that the screen never can show us.

There is one additional limitation that is our own eyes. We can't process images faster than at 30fps. Already at 50 Hz, we don't see the lights in our room blinking. If we are not interested in video quality for the update of our plots, we can safely go down to 20 Hz, and the images still look fluid.

### ? Exercise

Instead of plotting data from the device, you can update the plot with points that oscillate in time and see up to which point the refresh rate affects the quality of what you are showing.

There is one more thing to consider beyond the refresh rate, which is the number of points we

are plotting. Most screens have a few thousand pixels in each direction. A very common resolution is  $1920 \times 1440$  pixels. If we acquire 10000 data points and try to show them on the screen, they have to be reduced almost 5 times to fit the number of pixels available on the screen. In this reduction process, we can lose many details. If we use downsampling, for example, and we are looking for a narrow peak, the chances of it appearing on the image can be very little.

We have to be aware of the number of data pixels that we try to show not only on user interfaces but also when we are preparing plots for printing or inserting into a PDF. The number of dots a printer can generate is normally specified as dots per inch, or dpi. Even at 600 dpi, an image with a width of 8 cm (standard 1-column figure on a paper) will have under 2000 dots in its horizontal direction. And, of course, a reader behind a computer screen is limited to its pixels.

## 8.7 Conclusions

In this chapter, we have started building a user interface for the experiment. We explored how to get started with Qt, PyQt, how to use buttons to trigger actions using signals and slots. We also saw how to connect a basic user interface to the experiment model. It showed us the advantages of having an experiment that already runs measurements in its threads.

We also saw how to extend the basic building blocks of Qt, such as QMainWindow, by subclassing it and adding the elements we needed. We saw how to build widgets with more widgets inside, how to lay them out on more complex patterns. Finally, we added simple plotting capabilities to the window, refreshing whatever data the experiment is acquiring in real-time.

This chapter typically generates much satisfaction for people who are developing user interfaces for the first time. On the other hand, we have done much work, and we got a window that does not look nearly as nice as the windows with which we are familiar from other programs. On the one hand, this helps us understand how much effort is behind every window we see. On the other, it pushes us to go one step further.

In the next chapter, we start seeing how to improve the design of our User Interfaces by using a program called Qt Designer.

# Chapter 9

## User Input and Designing

### 9.1 Introduction

We started building a GUI by programming every aspect of the interface. We built a `MainWindow` class, added some buttons and a plot. A logical next step would be to add a way to change the parameters that build up the scan. We would like to be able to change the `start`, `stop`, and `num_points` values directly from the user interface and not from a config file that gets read-only at the beginning.

We could continue adding elements to the program as we did in the previous chapter, but that is time-consuming. In this chapter, we are going to introduce a program called **Qt Designer** that is targeted precisely at speeding up the design of user interfaces. We see not only how to make the program better looking, but we are also going to see what happens when we let users input data, what problems may arise, and we show the way to keep improving based on what we achieve by the end of the chapter.

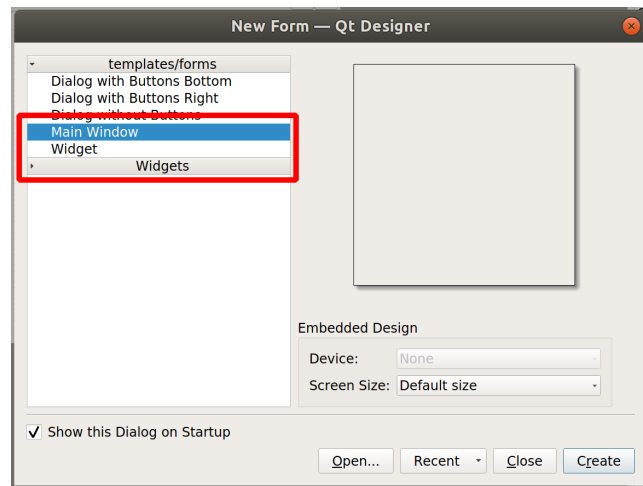
### 9.2 Getting Started with Qt Designer

We explained how to install Qt Designer in Section 2.5. Different operating systems and different Python versions have a slightly different way of opening the program.

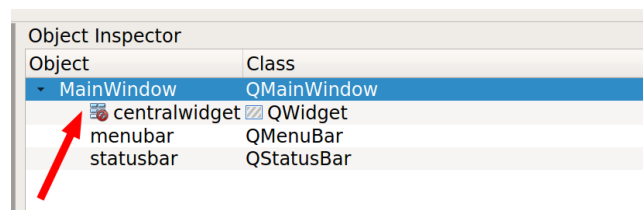
If you are using **Anaconda on Windows**, you only have to open the start menu and type *Designer*, press `enter` once it finds it. **Anaconda on Linux** is similar, open a terminal, either from the base environment or the environment used for this book, type `designer`, press `enter`, and a window should open.

If you are using **plain Python on Windows**, and you installed `pyqt5-tools`, you only need to start the *Command Prompt*, activate the environment where you work, type `designer.exe`, and press `enter`. If you are using **plain Python on Linux**, you installed the Designer as part of the package `qttools5-dev-tools`. In this case, the Designer is an actual application that you can find within the installed apps in your distribution.

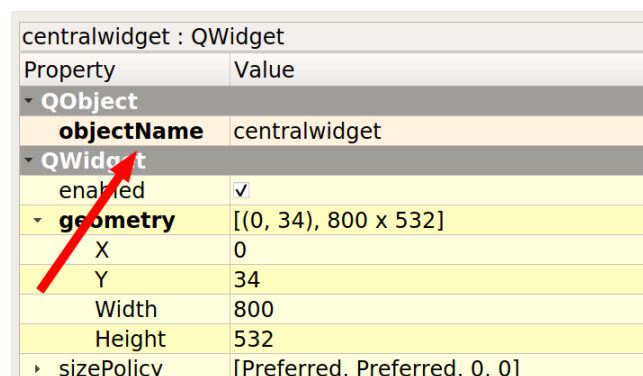
The Designer welcomes us with a screen like the one below. In between the template/forms options, we can already see two familiar options: Widget and Main Window.



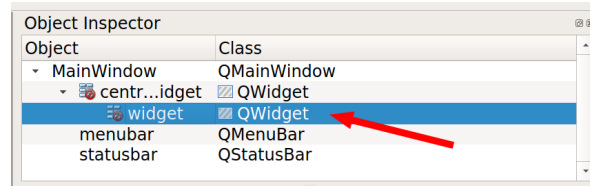
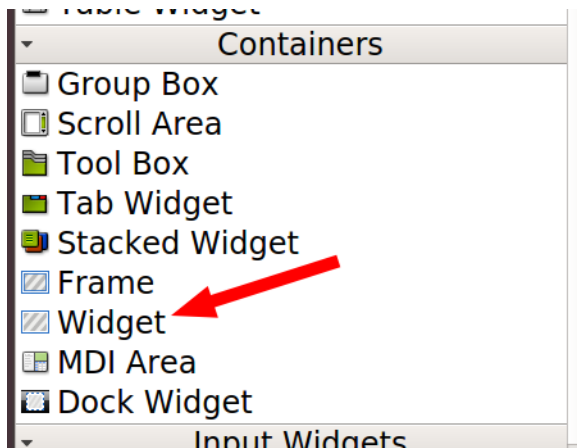
We start by recreating the window we developed in the previous chapter. We start by selecting **Main Window**, and clicking create. The Designer opens the working area with an empty main window. That space is our canvas to start adding elements. In the previous chapter, we created a central widget explicitly. The Designer already did this for us. We can see it on the object inspector at the right sidebar:



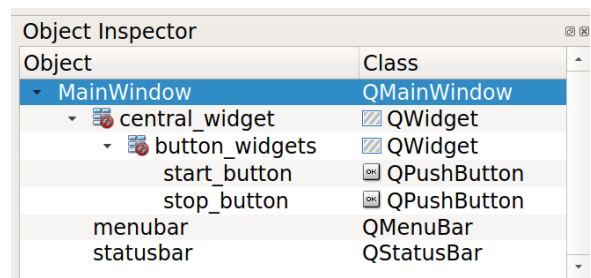
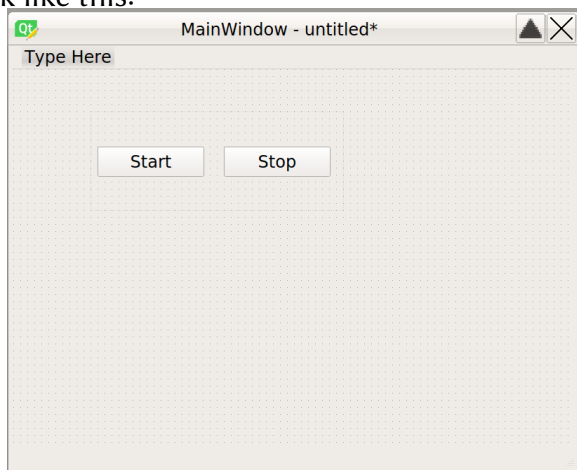
Note that the Designer named the central widget `centralwidget` instead of `central_widget`. Naming classes, variables, methods, and functions is completely free in Python, but some conventions make code easier to understand at first sight. One is naming classes with the Camel Case convention, such as `MainWindow`, and attributes in lower-case with words separated by underscores. We can change the name of the central widget by clicking on it and changing the `objectName` name property:



Now we have the central widget with the same name we used in our Python class. We can add the buttons widget by dragging and dropping an empty widget to the window, that can be found in the containers group of elements. After we add the widget, we can also see it appears in the object inspector on the right sidebar. We can change its name to `button_widgets`.



To add the start and stop buttons, we can just drag and drop two `QPushButton`s to the window. We have to be sure we drop them inside the buttons widget we created earlier. To change the text that appears on the button, we can double click on it and edit the text. We can make one button with the text *Start*, and the other with the text *Stop*. The text on the button is not the name of it. We must change the name of the buttons, and to repeat the same structure of the previous chapter, we are going to call them `start_button` and `stop_button`. The window and the structure should look like this:



## Screenshots

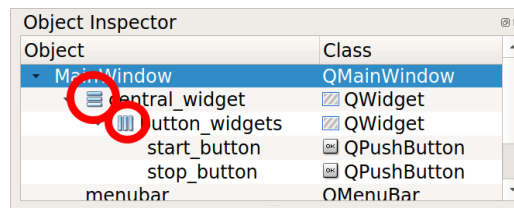
Explaining how to use a designing program through screen captures requires much patience from the reader. We try to highlight the checkpoints that allow comparing what the reader does with what we show here. Practice and a critical view is the best that can the reader can do at this stage.

We have a window with the buttons, but we are missing the layout to make it look as good as we had in the previous chapter. The Designer has a toolbar dedicated exclusively to the layouts; you can find it at the top of the main window space; it looks like this:



To apply a layout to a widget, we can click first on the widget inside the Object Inspector and then click on the desired layout. We can apply a vertical layout to the `central_widget` and

a horizontal layout to the `button_widgets`. When we apply a layout to widgets, it shows as a different icon on the Object Inspector.



We are ready to save the window. Let's create a new folder called **GUI** inside the *View* folder, and we can call the file `main_window.ui`. Once we save the designer file, we are ready to go back to Python to use this window. We can go back to `main_window.py`, and we can start editing the `MainWindow` class. Since some of the elements are now defined directly on the designer file, we remove them from the class:

```
import os
from PyQt5 import uic
[...]

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

        base_dir = os.path.dirname(os.path.abspath(__file__))
        ui_file = os.path.join(base_dir, 'GUI', 'main_window.ui')
        uic.loadUi(ui_file, self)

        self.experiment = experiment

        self.plot_widget = pg.PlotWidget()
        self.plot = self.plot_widget.plot([0], [0])
        layout = self.central_widget.layout()
        layout.addWidget(self.plot_widget)
[...]
```

Besides the new imports, the fundamental change to the `MainWindow` class is that we load the file using `uic.loadUi`. When we try to import files in Python, we always have to be careful with determining the path from which we are importing. We already encountered the syntax of `dirname` when we were adding the root folder to the path in Section 5.6. The idea is the same, but we are interested in the folder where the current Python file is located.

The command `uic.loadUi` takes two arguments; the first is the file we want to load, and the second is the object to which it is applied. We use the `self` to indicate that we want to apply the layout to the entire `MainWindow`. If we develop a more modular program, such as defining individual widgets, we could apply the Designer file just to the widget in which we are interested.

The rest of the window stays the same. We only removed the definition of `central_widget` and the buttons. Since we have the layout of the central widget specified in the Designer, we need to access it to be able to append the plot widget. It is what this line is doing:

```
layout = self.central_widget.layout()
```

We can go ahead and run the program, and we see that the window appears as it did before, without changes, even though we are now pulling the elements from the Designer file.



### 9.2.1 Compiling or not Compiling ui files

Most tutorials online add one step after creating the Designer's file. They normally suggest transforming the `.ui` files into a `.py` file. There is a program able to do it for us, that can be triggered from the command line:

```
pyuic5 main_window.ui -o compiled_window.py
```

We can explore both files and see the differences. If we open the `ui` file with a text editor, we find out it is plain text and formatted following a standard called *XML*, or extended markup language. This format is very similar to how websites are built, *HTML* stands for hypertext markup language. Going through the file, we find the place where we defined the buttons, surrounded by some more information, such as the layout, and the text of the button.

If we open the generated Python file, we find the same information but formatted in the same way we have done it in the previous chapter. We can go to the declaration of the start button, and we find a line almost identical to the one we used in the previous chapter

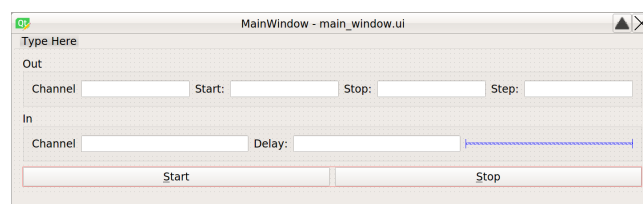
```
self.start_button = QtWidgets.QPushButton(self.button_widget)
```

The question is whether we **need** to compile the `ui` files to Python files or not. Qt was developed with C++ programmers in mind, and for a C++ program, it is essential to have each variable defined with a specific type before compiling the code. Qt offers a program, `uic` to transform `ui` files to C++ compatible files. Most Python tutorials built on that experience and kept the recommendation. But for Python, this step is not necessary at all.

There is one caveat, however. If we don't transform the file to a Python file, editors won't be able to know which attributes are available on the windows and widgets. The editor won't have any idea whether there is a `start_button` or not defined. It means that to be sure, we should keep opening the designer program and see how we named the buttons and elements. But this is the only drawback that non-compiling has.

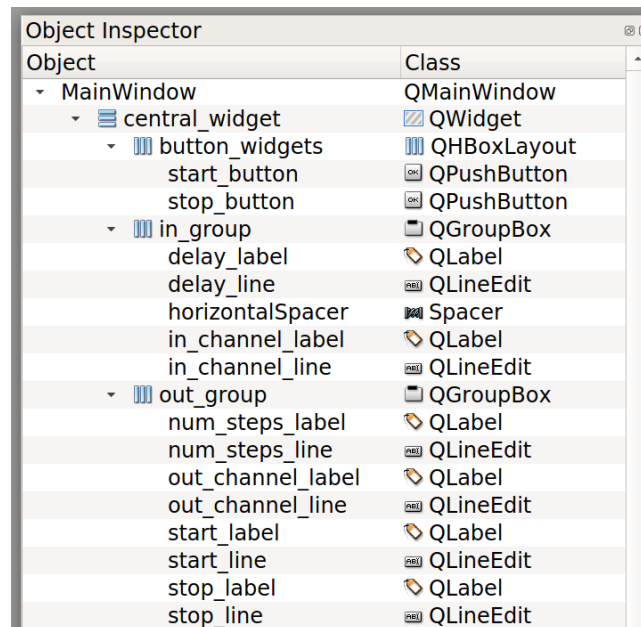
In our experience, modifying the designer files and seeing the changes as soon as we restart the program outweighs the problems of not being able to auto-complete. If we are systematic with the naming conventions, we won't face many issues. And if we do encounter issues with attributes not defined, we know that the root of the problem is that we used one name in the Designer and a different one in Python.

## 9.3 Adding User Input



It is time to bring our program to the next level by adding the possibility of entering the values of the scan before triggering it. The goal is to have a window that looks like the image above. We structured the window in three rows, one for selecting the range of the scan and the channel, one for the input and delay between points and finally the buttons. Below it, we append the plot, as we have done before.

First, we start by designing the window. We do not cover every single detail as we did in the previous section, because it would become too cumbersome to follow. Therefore we leave to the discretion of the reader whether they want to try it by themselves or get the designer file from the online repository. The easiest way to reproduce the window is by looking at the structure we should obtain after placing all the elements:



Object	Class
MainWindow	QMainWindow
central_widget	QWidget
button_widgets	QHBoxLayout
start_button	QPushButton
stop_button	QPushButton
in_group	QGroupBox
delay_label	QLabel
delay_line	QLineEdit
horizontalSpacer	Spacer
in_channel_label	QLabel
in_channel_line	QLineEdit
out_group	QGroupBox
num_steps_label	QLabel
num_steps_line	QLineEdit
out_channel_label	QLabel
out_channel_line	QLineEdit
start_label	QLabel
start_line	QLineEdit
stop_label	QLabel
stop_line	QLineEdit

The central widget and the buttons are there, as always, but we added two more widgets now, they are of a particular type called `QGroupBox`, meant for grouping elements together. They also have a title that appears at the top left, and become quite handy to understand what we need to edit before starting a scan. To handle input from the user, we chose `QLineEdit` widgets. These widgets allow the user to input any type of text, precisely as we need.

The rest of the elements are to give consistency to the user experience. Before each line edit, we included a label to show what information is supposed to go in each box. In the bottom group, we also used a `Spacer`, to push the elements to the left. If we don't do this, the boxes for editing would take the entire space and wouldn't look cute. It is, however, a purely aesthetic decision.

The most important thing to pay attention to is the names of each input line because they are going to be fundamental for the Python code. What we normally use is a distinctive name followed by the type of element with which we are dealing. That is why have the `start_button`, but also the `stop_line`, `delay_label`, etc. In this way, we can avoid confusion between the starting value of the scan and the button that is supposed to trigger it.

### Naming Consistency

Consistency with the names is very important. An incredibly common bug is to change the names and then use the wrong value for the experiment. If we were to swap the in and out channels, the error might go unnoticed until we use two different values. By that point, it is tough to understand if the problem is within the program or within the experiment.

After improving the window design, we can run again the program and we will see that the window has a different aspect, with inputs, labels and groups. However, they can't do much yet. If

we trigger a scan, the plot will update, but it will always be with the values set in the config file. First, let's learn how to populate the different elements with the values we start when we load the config file. We must edit the `__init__` of the `MainWindow` class to take care of the values:

```
self.start_line.setText(self.experiment.config['Scan']['start'])
self.stop_line.setText(self.experiment.config['Scan']['stop'])
self.num_steps_line.setText(str(self.experiment.config['Scan']['num_steps']))
```

The `QLineEdit` objects have a method called `setText` that allows us to change what is displayed. For the start and stop values, there are no problems, because their values are already strings (remember they include the units), but the number of steps is an integer. PyQt is a very thin wrapper and won't try to convert a number to a string. We need to force it ourselves, adding the `str` function.

### ? Exercise

We have shown how to add the values of start, stop, and number of steps, but we are still missing the input and output channels, and the delay between points. Add them following the example above.

We know how to set the values to the lines, we can change them, but the we still use the ones defined in the config file. We already saw in Section 7.2 that thanks to our strategy of having a robust experiment class, we can change the values of the parameters after the experiment was defined. We can do the same within the user interface. Let's improve the `start_scan` method of the `MainWindow`:

```
def start_scan(self):
    start = self.start_line.text()
    stop = self.stop_line.text()
    num_steps = int(self.num_steps_line.text())

    self.experiment.config['Scan'].update(
        {'start': start,
         'stop': stop,
         'num_steps': num_steps}
    )
    self.experiment.start_scan()
```

In exactly the same way we used `setText` to set the text on the `QLineEdit` objects, we can use `text()` to retrieve what is written. For start and stop there are no problems, but for the number of steps we need to have an integer, not a string. Once we got the values, we update the `experiment.config`, specifically the group of properties that belong to `Scan`. Using `update` on a dictionary is a shorter way of doing:

```
self.experiment.config['Scan']['start'] = start
self.experiment.config['Scan']['stop'] = stop
self.experiment.config['Scan']['num_steps'] = num_steps
```

It is important to remember that in the experiment, when we trigger the `start_scan`, the model takes care of creating the scan range directly from the values stored in the config dictionary. If we supply values without units, the experiment complains.

### ? Exercise

Test the limits of the user interface. What happens if you use a number of steps that is not an integer? What happens if you submit a start or stop value in units other than Volts? What happens if you leave one of the options empty?

### ? Exercise

Following the example above, add the same behavior for the delay, channel in and channel out values.

We are now at a stage where we can control our experiment from the user interface. We can change the parameters for the scan, start, and stop at will. At this stage, we should feel very proud of ourselves. It is also an excellent time to reflect because things change very quickly, new elements appear on the window, but we put minimal effort into making everything work together.

One of the values of separating in Model/View/Controller is that, usually, most of our work should go in the models. And models are the most pure-python modules in our program. Once we overcome the initial shock of working with more complex patterns, such as classes and threads, the rest is very straightforward. We loop through values; we save data. Most likely, it is the kind of things we were already doing when analyzing data.

Once the experiment model is ready, plugging the view on top of it does not require too much effort. How to design the experiment model in such a way that allows us such a degree of independence, is a skill that we can acquire over time, with critical thinking, and patience. For most programs, when developing the *View*, we would face the problem of something missing in the experiment model. Perhaps a threshold value, or a variable that lets us know something is running. We should always refrain from the temptation of complicating the view, and we should always favor putting all that effort into the models.

It's only the long run experience the one that congratulates us on a job well done in the past.

## 9.4 Validating User Input

We, as consumers of computer software, are used to a lot of interactions and visual cues when working with a user interface. It makes us expect the same kind of behavior in the programs we develop ourselves. One of the things we can notice in our program is that if we try to trigger a scan while the first one is running, the program prints a message to the terminal, but if we are not paying attention to it, we can miss it and wonder why the scan is not triggering.

One possible solution would be to gray out the *Start* button, to prevent the user from triggering a second scan. The question is, how can we achieve that behavior. If we look in the Designer, every widget that we add has a list of properties that we can change, including shape, color, text. In the case of the buttons, there is an option called `enabled`:

Property	Value
QObject	
objectName	start_button
QWidget	
enabled	<input checked="" type="checkbox"/>
geometry	[(1, 1), 690 x 43]
X	1
Y	1
Width	690
Height	43

If we remove the tick mark from the option, the button gets grayed out, and we won't be able to click it. We must learn how to change the enabled status of our program, not from the Designer. For this, we must read the documentation provided by Qt. If we search online for `QPushButton`, usually, the first result would be the one available at `doc.qt.io`, the official documentation. We only need to be sure we are looking at the Qt5 documentation and not at the Qt4. The page is long, and if we look for `enabled`, we won't find anything. It is not a very auspicious start.

If we scroll to the top of the page, we see that Qt informs us of the parent class of `QPushButton`: `QAbstractButton`. Remember that when working with objects, there is always a possibility that methods and attributes are defined in the parent class, not in the class we are using. We can follow the link, but there won't be any information on enabling or not, but if we go one level above, to the `QWidget` documentation, we find what we were looking for:

Header:	<code>#include &lt;QPushButton&gt;</code>
qmake:	<code>QT += widgets</code>
Inherits:	<code>QAbstractButton</code>
Inherited By:	<code>QCommandLinkButton</code>

To change the status of the button, we must use the `setEnabled()` method that is defined in the base `QWidget` class. On the one hand, we can notice that all widgets can be enabled or disabled. In some cases, the change is visible; in others, it won't be. We can also see that the Designer was already letting us know that the documentation was available as part of `QWidget` instead of `QPushButton`, just look at how it organizes the properties:

Property	Value
QObject	
QWidget	
QAbstractButton	
QPushButton	

Next time we want to learn how to do something, we know the best is to start by the Designer and see if the options are available. If we find it, we must pay attention to the base class that defines the behavior, and then we can go looking for the documentation. Navigating the Qt Docs takes a bit of getting used to, but once we understand how the information is organized, it becomes straightforward to follow it.

Back to the task at hand, we must disable the button when the scan starts running. One option would be to disable it right when we trigger the scan:

```
def start_scan(self):
    self.start_button.setEnabled(False)
    [...]
```

It seems like a very valid idea until we use it for the first time. Once the scan starts, the button is grayed out, but there is no place where the button can be enabled back. It is a typical pattern with user interfaces. There are different ways to tackle the problem, but the more straightforward one is using the timer we already have in place to update the plot.

Instead of just plotting data, we can use the same timer to update the different elements of the user interface. For example, we would like to let the user start the scan only if there is no scan running, and we want to let them stop the scan only if there is one scan running. Let's start by defining a new method in the main window:

```
def update_gui(self):
    if self.experiment.is_running:
        self.start_button.setEnabled(False)
        self.stop_button.setEnabled(True)
    else:
        self.start_button.setEnabled(True)
        self.stop_button.setEnabled(False)
```

The method `update_gui` is easy to follow. If the scan is running, we gray out one button but not the other and vice-versa. We only need to be sure this method runs periodically. Since we already have a timer for updating the plot, we can use it to update the GUI as well:

```
def __init__(self, experiment=None):
    self.timer.timeout.connect(self.update_gui)
```

With this simple approach, we check every few milliseconds whether the scan is running or not, and we set the buttons accordingly. There is also a risk with this approach. If we decide to show information on the Main Window that implies reading a value from the device, we end up polling it several times per second, regardless of whether the value changed or not. If we update values stored in an object, such as the Experiment model, there is nothing to lose, but if we stretch this pattern too far, we may encounter issues, and our program could become inefficient.

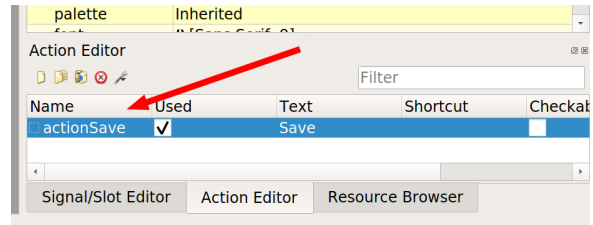
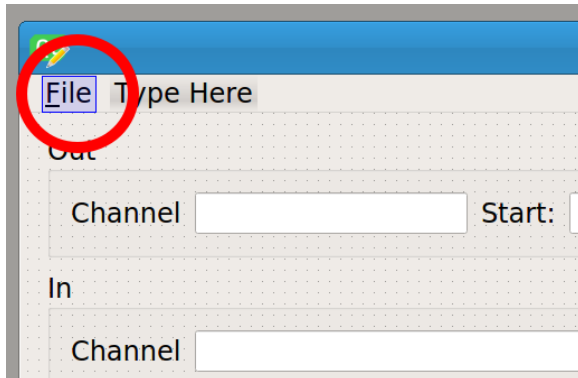
We have generated the most basic of the possibilities, disabling a button to prevent the user from triggering a second scan if there is one already running. There are many more possibilities. What happens if the user enters a value outside the range. For example, as a start without volts, or negative, or beyond 3.3 V. We can use the `update_gui` to monitor the values entered and run them through some checks. If they pass, then we leave them as they are. If they don't, then we change the line to red and disable the start button.

All these are possibilities that we start considering as soon as we start developing user interfaces. However, everything we do has ramifications, such as what we saw when we disabled the start button on our first attempt. Our general approach is not to complicate the user interface beyond what is needed. If we have proper drivers and proper models in place, at least our devices are safe. If we find ourselves or a user of our software, making the same mistakes over and over again, we can then optimize the flow to prevent it from happening again.

### 9.4.1 Saving Data with a Shortcut

The only missing important feature of our user interface is the ability to save data. We could implement a button for this, but we can also look into another important feature of Qt: **actions**. We saw that a convenient tool when developing with Qt are signals, such as the ones emitted by a button or a timer. Signals are associated with widgets of some sort. Another range is not associated with widgets, but the user can trigger them at any moment.

In the Designer, we can add a `File` menu, by double-clicking the top bar of the main window, where it says `Type Here`, and inside the File menu, we can create a `Save` option. As soon as we create the Save, the Designer also shows an *Action* defined. We can find the action listed at the bottom of the right sidebar, as we show on the image below. Note that the name assigned by default is `actionSave`.



If we pay attention to what appears on that square, we see that there is also the possibility of defining a shortcut. We can go ahead, double-clicking on it opens a small configuration window in which we can add a shortcut, such as `Ctrl+S`. Just clicking on the shortcut edit line and then pressing a combination of keys register them. After saving, we can rerun the program, and we have a File menu with a Save option, and the shortcut written next to it, so there are no possible confusions.

We need to connect the `actionSave` to the method in our experiment for saving data. Because of how we structured the code, this now becomes almost trivial, in the `__init__` of the `MainWindow` we just add the following:

```
self.actionSave.triggered.connect(self.experiment.save_data)
```

We can use signals to trigger methods in other objects, not only in the window itself. When the experiment gets more complicated, it can be constructive just to trigger the model to do something than first defining a method on the window to trigger the model then. If we rerun the program, we can save the data either by going to File/Save or by pressing `Ctrl+S`. Go ahead and check the folder you specified in the config file, and you see the data appearing there.



## Shortcuts

Adding shortcuts for actions can be a good idea, but it also has risks. Shortcuts are hard to document, and some shortcuts are so universal that it is better not to re-define them. We could have used `Ctrl+S` to Start the scan. But then, we need to stop it and choose `Ctrl+X`, because it is right below. To save, we need to choose another shortcut and choose `Ctrl+Q` because it is on top of the S. All these shortcuts are generally used for something else, like saving, cutting, and closing a window. We are free to use them as we wish, but over usage leads to a perilous path.

Of course, we may be tempted to open a pop-up dialog to ask for the folder where to save data. But again, this has a complex set of possibilities. If we open a pop-up every time we want to save data, it becomes annoying, especially if we save data often. If we show it only the first time, then we won't be able to change, later on, wondering why we showed it in the first place. It means we



need to establish two behaviors: *save* and *save as*. We won't show how to do it here, but we leave it as an exercise:

### Exercise

Qt bundles a `QFileDialog` widget, which allows the user to select a folder on the computer or create a new one. It can be used like this:

```
folder = QFileDialog.getExistingDirectory(self, 'Choose Folder').
```

Create a new action in the Designer to select the folder, and a new method in the main window to handle the selection of a directory to save the data.

## 9.5 Conclusions

This chapter is of great joy. It is the conclusion of a great journey, from developing a driver to creating models to abstract the behavior we expect both of the device and of the measurement we want to do, and finally to develop a user interface that works and looks beautiful.

This chapter focused on learning how to use the Qt Designer, a great tool to speed up the development of user interfaces. We covered the essential tools and patterns that one can expect to find in most user interfaces for scientific experiments. Qt, however, is a massive library that allows building almost anything, from the interface you see in some Mercedes cars to coffee vending machines, to programs running right now on your computer or mobile phone.

We have merely scratched the surface of what Qt offers. For scientific applications, however, we don't need to go much deeper. We are generally okay with the simple styling of the objects and a simple menu structure. We can change font-sizes, colors, the roundness of edges. We can make floating menus that open by right-clicking on an image, and many more things. But we always have to draw a line between how much effort and time we are dedicating to making a program look better and how much time we are dedicating to running an experiment.

## 9.6 Where to Next

If you reached this point, it means that you have kickstarted your career as a Scientific Python Developer, hurrah!. What we have covered in this book is only the beginning. Devices can be much more sophisticated than a simple DAQ, we may need to analyze the data before we display it, or the memory on our computer may not be enough to hold one complete measurement.

Our advice during the workshops is that you should work on what you have learned for at least 6 months—trying to develop solutions for the tools you already have at hand, such as an Oscilloscope, or your own data acquisition card. In this book, we have guided you; we have prevented you from falling in some errors. The reality is that you won't master any technique until you try by yourself, fail by yourself, and solve a problem by yourself.

To accompany you in this process, we have built a forum<sup>1</sup>. You can ask any question related to programming, work in the lab, or Python. If you have any comments, suggestions, compliments, critiques to send us, you can do it directly to me at [aquiles@pythonforthelab.com](mailto:aquiles@pythonforthelab.com). I always want to read how your path is going, what topics are making you struggle.

---

<sup>1</sup>available at: <https://forum.pythonforthelab.com>



If you are interested in a follow-up to this book, covering more advanced topics, please drop me a line. I am always seeking new ideas, topics, challenges, not only to produce content but to learn myself. I do a periodic braindump of what I learn on the website [pythonforthelab.com](http://pythonforthelab.com). If anything grabs your attention, you can always leave a comment to let us know.



# **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. You will notice that it is similar to what you would typically find in the manual of any instrument in your 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. It 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 – 3.3 V.

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 to mitigate this effect.

### A.2 Communication with a computer

The DAQ can 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 9600, and every command has to finish with the newline ASCII character. A newline character also terminates the messages generated by the device.

### 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. *Returns:* the value that the device received.

*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 – 1023

*Example:* IN:CH5

# 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 be able to develop software for controlling a complicated setup. However, not all programmers have the same background, and it is essential to establish a common ground from which to start.

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

### B.2 The Interpreter

You can start Python from the command line by typing Python and pressing Enter. It 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 gives 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. Power of numbers can be achieved using `**`, for example, `2**.5` means the square root of 2.

### B.3 Lists

From now on, the `<<<` is suppressed 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 for 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)]
```



It generates 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 essential to keep them in mind 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, like 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 obvious 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 matter which element is the base and which one is the height. However, for more complex applications, altering the order can have severe 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 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}
```

The syntax above, on the 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 the second element of a list or the element of a dictionary. 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 outputs 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.

```
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 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 to define not only 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, or string. 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 birthday, and have a function that returns the person's age.

In the beginning, it is not clear why objects are useful, but over time it becomes impossible not to think with objects in mind. Python takes the ideas of objects one step further and considers every variable an object. Even if you didn't realize, you might 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 custom behavior for most of the common operations.

### C.1 Defining a Class

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

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

When speaking, it is tough 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. It means that we use classes when referring to the type of variable, while we use object to the variable itself. It is going to become more apparent 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, you must start Python from the same folder where the file **person.py** is located. When you run the code above, you should see as output `John Snow`. We omitted an important detail: 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 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 had 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 potent 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. It can give severe 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 handy 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 is no error, it just returns an empty string. While initializing, you can also force the execution of other methods, for example:

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

It has the same final effect. It is 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 calling specific methods 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 stops working, because now you have to set the `name` explicitly. If there is any other code that does `Person()` fails. 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 defaults to `''`, i.e., an empty string.

### ? Exercise

Improve the `get_name` method 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 receives the new property. In the beginning, it is hard to understand why it is useful, but one day you need it, and it saves you plenty 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 generally 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 can 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 following:

```
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 see that no error appears, but also that nothing appears on the 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 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 many things. It is just a matter of thinking about how to connect different methods when it is useful to inherit. Without a doubt, it helps 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 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__` is used 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 important to point out is that this method is inherited. Therefore, if instead of printing a `Person`, you print a `Student`, you see the same output, which may or may not be the desired behavior.

## C.5.2 Defining complex properties

When you are developing multiple 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
```

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 allows you to use `me.year_of_birth` as an attribute, but it fails if you try to change its value. It 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 convenient. It takes a bit more time to develop than with simple methods, but it simplifies a lot the rest of the programs that build upon the class.