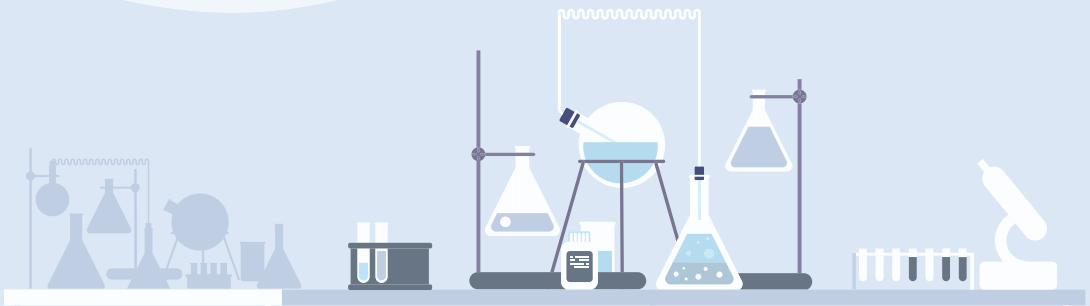
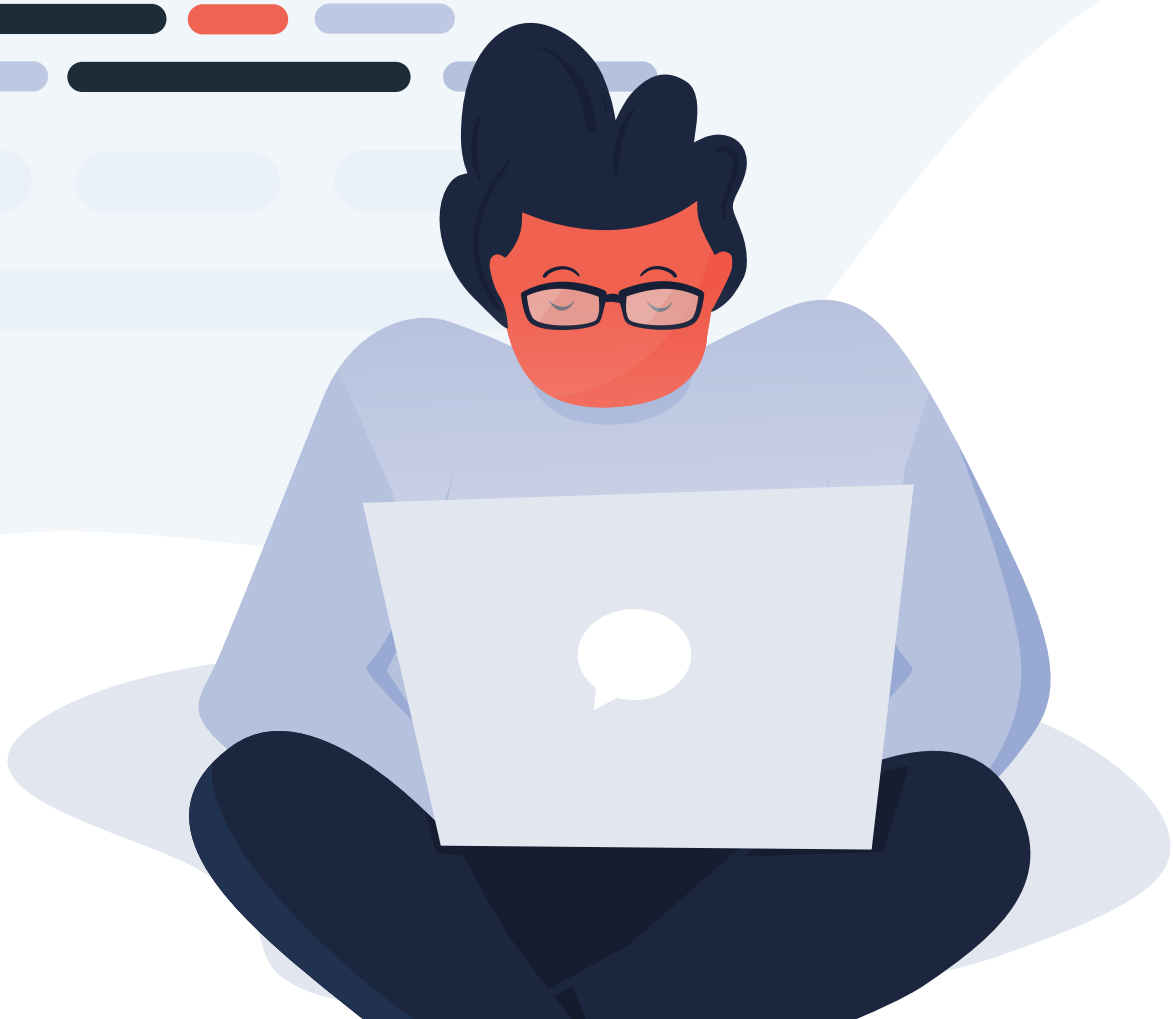
Dr. Aquiles Carattino



Python for the lab



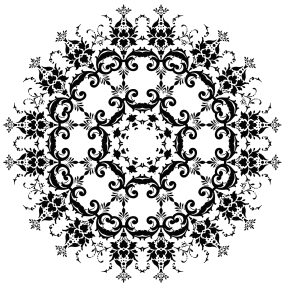
PYTHON FOR THE LAB

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 September 4, 2020

Contents

Chapter 1

Introduction

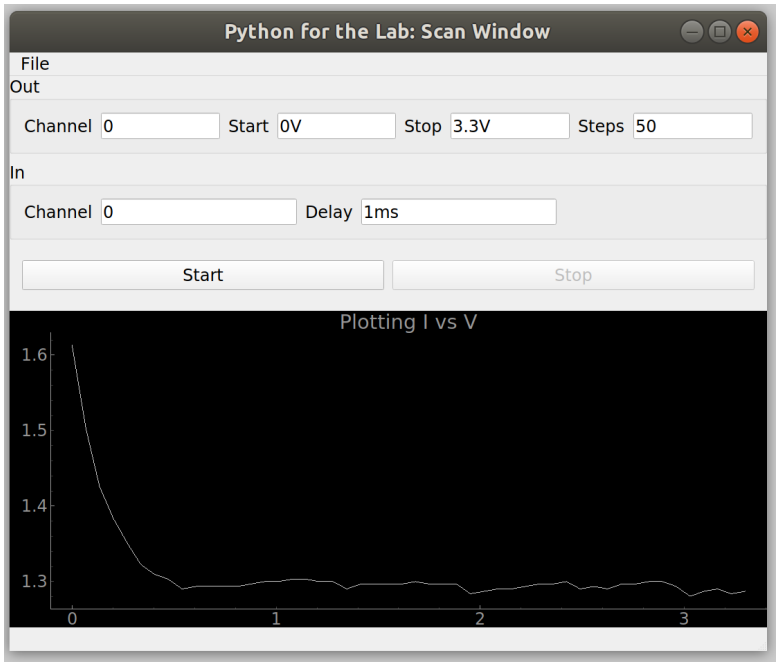
In most laboratories around the world, computers are in charge of controlling experiments. From complex systems such as particle accelerators to simpler UV-Vis spectrometers, there's always a computer that's responsible for asking the user for some input, performing a measurement, and displaying the results. Therefore, learning how to control devices with a computer is 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 focuses on how things can be done and not so much on theory or on how programming works in general. This approach can lead to some generalizations that may not be correct in all scenarios. We ask your forgiveness in those cases and your cooperation: if you find anything that can be improved or corrected, please contact us[1].

Together with the book, there's a website[2] where you can find extra information, anecdotes, and examples that we couldn't fit here. Remember that the website and its forum are the proper places to communicate with fellow Python For The Lab readers. If you're stuck on any of the exercises or have questions that aren't answered in the book, don't hesitate to post in the forum. Continuous feedback is the best way to improve this book.

[1] courses@pythonforthelab.com
[2] https://www.pythonforthelab.com

1.1 What You're Going To Learn



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

We crafted each chapter to introduce new Python topics alongside the specific tools needed to control a related 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 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. By following a task-oriented approach, students get the initial direction, tools, and vocabulary to ask for help if needed.

Regarding instrumentation itself, we've 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 other parts of 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 where it's not necessary.

1.2 Who This Book Is For

We don't assume any proficiency in Python for readers looking to follow the book. We ask only that you have a grasp of *if-statements*, *for-loops*, *while-loops*, and when to use them. You'll gain new knowledge when it's required through carefully planned examples and exercises. If you're already proficient in Python, we still believe there's value in the best practices we show, the structure of the code, and the way we decided to solve problems. There may be many paths, but the goal is the same: to 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. You don't need to understand the phenomenon; we use it as an example of an experiment in which one voltage is varied and another voltage is measured. This simple example is a building block of most experiments, like controlling temperatures, moving piezo-stages, or tuning a laser's frequency. By using an LED as the diode in the experiment, we can see the effect of applying a voltage.

1.3 How to Obtain the PFTL DAQ Device

We've developed a device called the PFTL DAQ that works as a data acquisition board. The instructor provides these boards during the workshops, but if you acquired this book online and would like to buy one of these devices, please contact us[3]. The devices are open source/open hardware and are based on the **Arduino DUE**. You can find the instructions for building one on our website. If you have access to any other acquisition card, you can adapt the course contents to your needs with a bit of tinkering.

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

1.3.1 What you'll need to build your acquisition device

If you want to build the acquisition device, the main component you should source is an **Arduino DUE**. The official web store[4] usually has them in stock. You can also check out other retailers such as eBay and AliExpress. If your budget allows for it, we strongly recommend getting an official board, since that's the best way to support the project.

In addition to the acquisition board, you'll also need a small protoboard, three jumper wires, one LED of any color you prefer, and a resistor of $100\,\Omega$ (although $220\,\Omega$ should also work fine). These components can easily be found in any electronics shop, both online or in store. If you work at a university or research institute, most labs will already have them in stock for you to borrow.

Once you have all the components, you can load the firmware onto the Arduino board. The code you'll need is freely available in our Github repository[5]. If you encounter problems at any point, please don't hesitate to contact us. Our website has more information on how to get started with Arduino. We're also willing to answer your questions either via e-mail or on the forum.

1.4 Why Build Your Own Software

Computers and their software 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 own programs, these limitations fall and creativity can sprout. With automation, researchers can increase the throughput of the setup and reduce human error. Even experiments that were once impossible can become reachable by introducing smart feedback loops.

[3] courses@pythonforthelab.com
[4] https://www.arduino.cc
[5] https://github.com/PFTL/pythonforthelab

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

1.5 Why Use Python

Python has become ubiquitous in many research labs for several 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 processes for generating data should be open to present and future colleagues. Python leverages the knowledge gathered in myriad areas to deliver a better product. Python can be found everywhere, from high-performance computing to machine learning, experiments, websites, and more.

Another factor to consider is that Python is free, so there's no overhead when it comes to implementing it. There's no limit to the number of machines on which you can install Python or the number of different simultaneous users you install it for. Moreover, there's a myriad of professionally-developed tools (such as NumPy, SciPy, and scikit-learn) that are also free for you to use. Anaconda, a popular data science platform, provides users with high-quality advice and troubleshooting, filling an often encountered gap in open-source software.

However, for experimentalists, there's a big downside when considering Python. Searching online for instructions on how to control an experiment yields few sources and even fewer if focusing on Python alone. Fortunately, this is changing thanks to an ever-growing 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 so it makes sense to bring its use to the source of the data, which is the experiment itself. Moreover, many more exciting things are possible with Python, like building websites, developing machine learning algorithms, and automatizing your daily tasks. Learning Python increases your odds of finding work both in and out of academia, and for both people who wish to continue working with experiments and those who want to focus on data analysis and beyond.

1.6 How to Use the Onion Principle

When we start developing software, it can be 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 a little deeper. Our software won't handle the new tasks, and we'll 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're talking about are not written in stone. You won't find them in any book (we haven't written them here, either). Instead, we're talking about a state of mind that empowers us to develop better, clearer, and more expandable code. Sitting down and reflecting is the best thing we can do, even moreso than sitting down and typing. When dealing with experiments, we have to

ask ourselves may questions. What do we know? What do we need to prove how to do it? Only then can we sit down to write a program that responds to our needs.

If we build something that we cannot expand, then it will become 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. This isn't something that occurs naturally, but we can develop our procedures to ensure that we're developing future-proof code. Once we get the hang of it, it will take much less time than being disorganized and not having the proper structure. We can avoid problems such as variables that aren't self-descriptive, lack of comments, and many more.

It's not all about being future-proof, though. When we start with a simple task, we want to solve it quickly. We don't want to spend hours developing useless lines of code just thinking *what if*. This approach is known as *premature optimization*. If we spend too much time trying to solve a problem that may appear, we might not see the issue that will arise. Therefore, it's 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 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, perhaps even starting with sample code.

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

1.7 Where to Get the Code

The code that you'll develop through the book is freely available on Github[6]. The code has been organized by chapters to make it more accessible while reading. There's 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, readers can think of the possible directions to take for their software after they've finished with the book.

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

1.8 How to Organize a Python for the Lab Workshop

Python for the Lab was born to bring unite researchers working in a lab with the Python programming language. With that goal in mind, we developed this book along with a workshop in which we can train scientists.

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

[6] https://github.com/PFTL/py4lab

Chapter 2

Setting Up the Development Environment

2.1 Introduction

To start developing software for the lab, you're going to need various different programs. The process of installing those programs will be different depending on your operating system. It's almost impossible to keep up-to-date, detailed sets of instructions for every possible version of each program, or for every possible hardware configuration. The steps below are generalized and should not present you with any issues. When in doubt, it's always best to either check the instructions that the package developers provide or ask in the forums.

2.2 Choosing Pure Python or Anaconda

If you're already familiar with Python, then you probably know that there are different **distributions** that are worth discussing. In and of itself, Python is a text document that specifies what to expect when it encounters certain commands. This gives you much freedom to develop different implementations of those specifications, each one with different advantages. The *official* distribution is available at python.org and is the distribution maintained by the Python Software Foundation. In the following sections, you'll see how to install it step by step. This distribution is also referred to as CPython because it's written in the programming language **C**. The official distribution follows Python's specifications to the letter. As a result, this is the one that comes bundled with Linux and Mac computers. Newer versions of Windows will start shipping the official Python distribution as well.

However, the base implementation of Python left room for improvement in certain areas. Some developers started to release optimized Python distributions that were tailored for specific tasks. For example, Intel released a specially-designed Python version to support multi-core architectures, one that leverages specific, low-level libraries that they developed 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's one that's very popular amongst scientists and those doing numeric computations called **Anaconda**, which we cover in this book.

To expand Python, we can use external packages that can be developed and made publicly available. In the past, the Python package manager (pip) was limited; it allowed you to install only more straightforward packages. There was a clear need to have a tool that allowed the installation of 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. Those libraries are not always easy to install on all operating systems, nor is it straightforward to keep track of their dependencies and versions. Anaconda was born to address these issues and is still thriving to this day.

Anaconda is a distribution of Python that comes with *batteries included* for scientists. It includes many Python libraries by default as well as several supporting programs. It also contains a potent package manager that allows you to install highly-optimized libraries for different environments, regardless of whether you're using Windows, Linux, or Mac.

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

2.3 Installing Anaconda

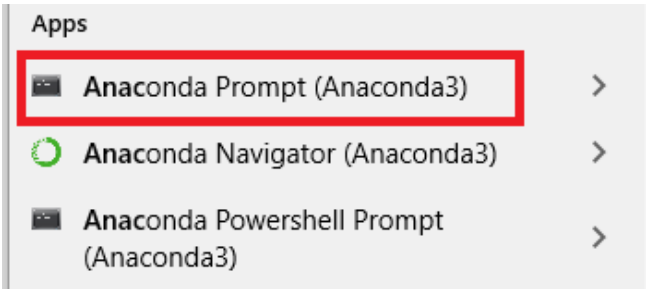
To install Anaconda, you just need to head to the official website: anaconda.com. Go to the download section and select the installer for the newest version of Python. The site usually auto-detects your operating system and offers you either a graphical installation (recommended) or a command-line one. If you're 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'll just need to be aware that other programs that rely on Python will use the Anaconda version and not the stock version.

Note

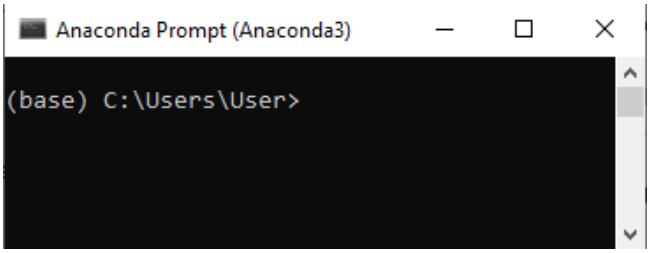
Similar to the different distributions of Python, Anaconda also comes in two primary flavors: Anaconda and Miniconda. The main difference is that the latter bundles fewer programs and is therefore lighter to download. Unless you're very low in space on your computer or have particular requirements, we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

2.3.1 Using Anaconda

Even though Anaconda comes with a graphical interface to install packages, it's highly recommended that you use the command line because it's easier to transmit ideas with words. If you're on Windows, you need to start a program called *Anaconda Prompt*, as shown in the image below:



If you're on Linux, you only need to open a terminal. On Ubuntu, you can do this by pressing Ctrl+Alt+T. What's important to note is that when you trigger Anaconda, you'll see that your command line has a `(base)` prepending it, as shown in the image below:



This is the best indication to know you're running the Anaconda installation.

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

```
conda list
```

The output you'll see will depend on what you have installed and whether or not you've already used Anaconda in the past. You'll see that at the beginning it tells you where your Anaconda installation is, followed by four columns: Name, Version, Build, and Channel. It should look 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
```

This image shows a few example packages, but your output should be much longer.

One of the good things about Anaconda is that it keeps track of each package, its version, and 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. You could also be using the same version of `numpy` but with a different version of Python (hence the `py37` that appears in the build numbers) which allows you to keep track of what you're doing at any given moment.

The last two lines show you a package called `pyyaml` that depends on a library called `yaml`, which you'll use later. With Anaconda, you can separately keep track of both the Python package and the lower-level library that this package uses. If you're coming from Linux, this won't be a great surprise, since this is precisely what the package manager on your computer does. If you're coming from Windows, however, this is something incredibly handy.

Say you want to install a package that you don't yet have on your machine. Let's see how you would install `PySerial`, which is a package that you'll use later in the book. Installing it becomes as easy as running the following command:

```
conda install pyserial
```

This command outputs some information, such as the version and the build, then asks if you want to install it. You can select 'yes' and it will proceed. If you list the installed packages again, you'll notice that PySerial is listed there.

But this is not all Anaconda allows you to do. You can also use separate environments based on your projects.

2.3.2 working with conda environments

A **conda environment** is, in practical matters, a folder where all the packages that you'll need to run your code are located, including any underlying libraries. the environments are **isolated** from each other; in other words, updating or deleting a package in one environment won't affect the state of that same package in any other environment. when you're working on different projects, there may be times where one needs a specific library version, and you don't want to ruin the other projects. to create a new environment, you need to run the following command (changing `myenv` to any name you want):

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

then you activate it:

```
conda activate myenv
```

now if you list the installed packages you'll see there's nothing there:

```
conda list
# packages in environment at /opt/anaconda3/envs/myenv:
#
# name                    version                   build  channel
```

from here, in your newly created environment, it's time to install the packages you want, starting with python itself:

```
conda install python=3.7
```

this will install the specified version of python in your new environment.

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

After installing Python, you can run it by typing:

```
python
```

The output will look 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'll need these packages:

- NumPy -> For working with numerical arrays
- pySerial -> For communicating with serial devices
- PyYAML -> For working with YAML files, a specially structured text file
- PyQt -> For building Graphical User Interfaces
- PyQtGraph -> For plotting results within the User Interfaces

You can install all of these by running:

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

Don't worry too much about these packages, since you'll see them one-by-one later on.

If you run `conda list`, you'll see that there are *many* more packages installed. Each package depends either on other packages or libraries, and Anaconda took care of installing all of them for you. With a `conda install` command, you can install packages that Anaconda itself maintains. These are official packages that come with a certification of quality. Many companies will allow their employees to only install packages that are officially supported by Anaconda, in order to avoid having malware installed within their network.

To follow the book, you'll need one more package called `Pint`. This package is *not* in the official conda repositories. To install packages that aren't yet in the official repository, you can use an unofficial repository called `conda forge`. Packages that aren't mature enough, or versions that are too new and not tested enough, are located in this repository. To install a package, you just need to run the following command:

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

The `-c conda-forge` specifies the `channel` you want to install the package from. With this, you've finished installing all the packages you'll need to follow the rest of the book.

If you want to leave the environment, you can run:

```
conda deactivate
```

This should return you to your normal shell prompt.

Creating environments more quickly

In the steps above, you created an empty environment and then installed the necessary packages. You can perform this operation slightly faster if you already know what you need. For example, you 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.

Removing 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` it means that you want to remove a specific package from that environment, while the `--all` option tells Anaconda to remove *all* the packages *and* the environment itself. Use with care, since you can't undo it!

2.4 Installing Pure Python

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

2.4.1 Installing Python on Windows

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

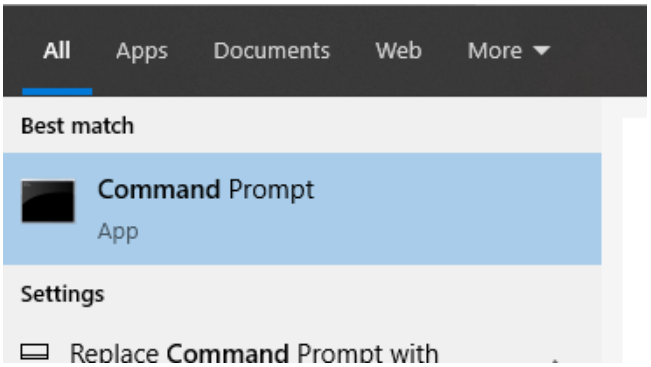
> **Note**
>
> We've tested all the contents of this book with Python 3.7, but newer versions shouldn't give you any problems. If you install a more recent version and run into any problems later on, come back to this step, uninstall Python, and then reinstall an older version.

Once the download is complete, you should launch the installer 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're good to go! Pay attention to the messages that appear, just in case anything goes wrong.

Testing your installation

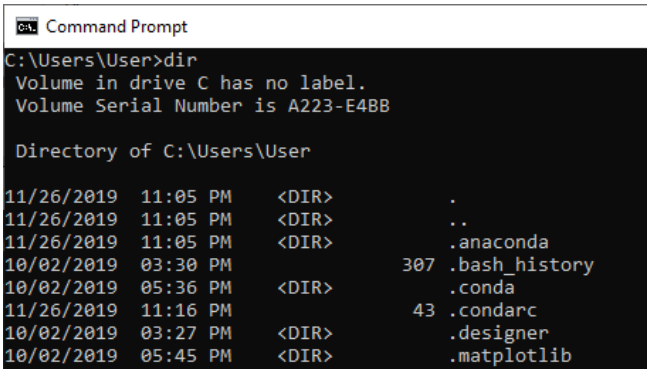
To test whether your installation of Python is working, you need to launch the Command Prompt, which is the Windows equivalent to a Terminal in most Unix operating systems. Throughout this book, we'll use the terms Command Prompt, Command Line, or Terminal interchangeably.

The Command Prompt is a program that allows you to interact with your computer by typing commands instead of using the mouse. To launch it, click the Start Button and search for *Command Prompt* (it may be located in the Windows System apps). It should look like the image you see below:



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`, which stands for *change directory*. If you want to go one level up, then you can type `cd ..`, where the two dots `..` represent the parent folder of the one you're currently located in. If you want to enter a folder, then you type `cd Folder` where *Folder* is the name of the folder you want to change to.

It's out of the scope of this book to cover all the different possibilities that the Command Prompt offers you, but you shouldn't have any problems finding help online. See the image below to get an idea of what using the Command Prompt looks like on Windows:



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.
```

The output shows you which Python version you're using as well as some extra information. You've just started what's called the Python Interpreter, which is an interactive way of using Python. If you come from a Matlab background, then you'll 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 you're 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, then 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, then you'll have to do this manually.

First, you need to find out where your Python is installed. If you paid attention during the installation process, then that shouldn't be a problem. Most likely you can find it in a directory like this one:

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

Once you find the file `python.exe`, copy the full path of that directory (that is, the location of the folder where you found `python.exe`). You then have to add that file location to the system variable called PATH. Here are the steps you'll need to take:

1. Open the System Control Panel. How to open it is slightly dependent on your Windows version, but it should be something like *Start/Settings/Control Panel/System*.

2. Open the *Advanced* tab.

3. Click the *Environment Variables* button.

4. Find the 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 semicolon (`;`).

5. Add the folder where you found the `python.exe` file at the end of the list. Don't forget the semicolon (`;`) to separate it from the previous entry.

6. Click OK.

You'll have to restart the Command Prompt for it to refresh the settings. Try to run `python.exe` again and it should work.

2.4.3 Installing Python on Linux

Most Linux distributions come with Python already installed. To check whether it's already on your system, open up a terminal (Ubuntu users can press Ctrl+Alt+T). You can then type `python3`, and press enter. If it works you should see something like this appear 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, then you need to install Python 3 on your system. Ubuntu users can do this by running the following:

```
sudo apt install python3
```

Each Linux distribution has a slightly different way of installing Python, but all of them more or less follow the same procedure. After the installation, check to see 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 add the *3* explicitly. In case there's an error, try running only `python` first and check whether your machine recognizes that you want to use Python 3.

2.4.4 Installing Python packages

One of the characteristics that makes Python such a versatile programming language is the variety of packages that you can use in addition to the standard library. Python has a repository of applications called PyPI, which stands for the Python Package Index. PyPI contains more than one hundred thousand packages. The easiest way to install and manage these 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 handles dependencies so you don't have to worry about them.

pip works both with Python 3 and Python 2. To avoid mistakes, you have to be sure that you're using the version of pip that corresponds to the Python version you want to use. If you're on Linux and have both Python 2 and Python 3 installed, there are probably two commands, `pip2` and `pip3`. You should use the latter to install packages for Python 3. On Windows, you likely have to use `pip.exe` instead of just `pip`. If this doesn't work for some reason, you'll need to follow the same procedure that you saw earlier to add `python.exe` to the PATH, but this time with the location of your `pip.exe` file.

Info

Since the moment Anaconda was born up until now, pip has gone through many trials and tribulations. Today, you can install complex packages such as NumPy or PyQt directly. However, there's still some discussion regarding how much we can expect from pip at the moment as far as compiling programs or performing complex tasks.

Now, installing a package becomes very simple. If you'd like to install a package such as NumPy, you should just type the following:

```
pip install numpy
```

Windows users should type this:

```
pip.exe install numpy
```

> **Before You Continue**
>
> Before installing the packages listed below, it's essential that you read the following section on **Virtual Environments**. These will help you maintain clean and separate environments for software development.

- NumPy -> To work with numerical arrays
- Pint -> To use units and not just numbers
- pySerial -> To communicate with serial devices
- PyYAML -> To work with YAML files, a specially structured text file
- PyQt5 -> To build Graphical User Interfaces
- PyQtGraph -> To plot results within the User Interfaces

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

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

2.4.5 Working with virtual environments

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

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

Unintentional library upgrades can set you back several days. Sometimes it might be 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. Other times you may want to check what would happen if you were to upgrade a library, or you might wish to reproduce the set of packages installed by a different user to troubleshoot any issues. There's no way of overestimating the benefits of isolating environments on your computer.

Fortunately, Python provides you with **virtual environments** that give you a lot of control and flexibility. A virtual environment is nothing more than a folder where you'll find copies of the Python executable and all the packages you installed. Once you activate the virtual environment, every time you trigger pip for installing a package, it will do so within that directory. The Python interpreter will be the one inside the virtual environment and not any other one. It may sound complicated, but in practice, it's incredibly simple.

You can create isolated working environments for developing software to run specific programs or for performing tests. If you need to update or downgrade a library, you'll do so within that specific virtual environment, and you won't alter the functioning of anything else on your computer. It may take some time for you to acknowledge the advantages of using virtual environments, but once you lose days or even weeks reinstalling packages because something went wrong and your experiment doesn't run anymore, you'll understand.

> **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 out Anaconda.

Working with virtual environments on Windows

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

Python comes bundled with a tool for creating virtual environments, so you can install the package with pip:

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

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

```
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 version of Python installed, then you can choose whether you wish to use, for instance, Python 2 or Python 3 for that specific project. The command also creates a folder called Testing, where you'll find all the required packages and programs. If everything went well, then you should see that your command prompt now displays a `(Testing)` message before the path. This means that you are indeed working inside the environment.

Once you've finished working on your project and you want to leave the virtual environment, you can type the following:

```
deactivate
```

This will return you 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's a new version available, then it will be installed. One of the most useful commands to run within a virtual environment is this:

```
pip freeze
```

This command gives you a list of all the packages installed within that working environment and their exact versions. This is so you'll know what you're using and 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 later.

You can install the packages listed before, such as NumPy and PyQt5, and see that they will only be installed within your `Testing` environment. If you activate or deactivate the virtual environment, the packages you installed within it will not be available, which you can see with `pip freeze`.

Working with virtual environments in Windows PowerShell

If you're using Windows PowerShell instead of the Command Prompt, there are some things that you have to change. First is the virtual environment wrapper package, which needs to be installed separately for PowerShell:

```
pip install virtualenvwrapper-powershell
```

Most likely, you'll need to change the execution policy of scripts on Windows. Open a PowerShell with administrative rights (to do so, right-click on the PowerShell icon and then select *Run as Administrator*). Then run the following command:

```
Set-ExecutionPolicy RemoteSigned
```

Follow the instructions that appear on the screen to allow the changes on your computer. This should allow the wrapper to work. You can repeat the same commands that you saw before to create a virtual environment.

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

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

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

```
.\activate
```

Now you're running within a virtual environment in PowerShell.

Working with virtual environments on Linux

Installing the virtual environment packages on Linux is more routine. Depending on where you installed Python, you may need root access to follow the installation. If you're 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're on Ubuntu, you can install the package through apt, although it's not recommended:

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

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

```
which python3
```

The output will tell you the location of the program that's triggered when you run `python3` in a terminal. Replace the location of Python in the following command:

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

This 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 your virtual environment and is the place where new modules will be installed. If everything went well, then you'll see the `(Testing)` string at the beginning of the line in your terminal. When you see it, you know that you're working within a virtual environment.

To close the virtual environment you have to type the following:

```
deactivate
```

To work in the virtual environment again, just do this:

```
workon Testing
```

If for some reason the wrapper isn't working, you can create a virtual environment manually by executing the following code:

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

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. This behavior can be handy if, for example, you want to share the virtual environment with somebody, or place it in a precise location on your computer.

Once you've activated the virtual environment, you can install the packages listed before, such as NumPy. You can compare what happens when you're in the environment to what happens outside, and check that you truly are isolated from the central Python installation. The packages that you install inside of `Testing` should not be available outside of it.

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

```
pip freeze
```

This command gives you a list of all the packages installed within that working environment and their exact versions. This is so you'll know what you're using and can revert the package version 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 the results at a later time.

2.5 Using Qt Designer

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

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

2.5.1 Installing Qt Designer on Windows

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

```
pip install pyqt5-tools
```

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

Info

The package `pyqt5-tools` is an independent package 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 this writing, it's known to work with Python 3.7, but not with Python 3.8.

2.5.2 Installing Qt Designer on Linux

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

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

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

2.6 Choosing a Text Editor

To complete the Python For The Lab book, you'll need a **text editor**. As with many decisions in this book, you are entirely free to choose whichever one you like. Still, there are some resources we'll point out that can be useful to you.

You won't need anything more sophisticated for editing code than a **plaintext editor**, such as Notepad++. This is available on Windows only, and it's very basic and straightforward. You can have several tabs open with different files and you can perform a search for a specific string in your opened documents, or even within an entire folder. Notepad++ is very good for small changes to code, perhaps made directly in the lab. The equivalent to Notepad++ on Linux are 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 simultaneously, being able to check that your code is correct before running it, and ideally being able to interface directly with virtual environments (whether you're using conda or virtualenv). For all this, there is a range of programs called IDEs or **Integrated Development Environments**. We strongly suggest you check out **PyCharm**, which offers a free and open-source Community Edition as well as a Professional Edition (which you can get for free if you're a student or teacher affiliated with a university). PyCharm integrates itself with virtual environments and allows you to install a package if it's missing, but you need it and many more things. It's 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 Code**, which is very similar to PyCharm in it's capabilities. 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 helping 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 do. Remember that the choice is always yours. Editors should be a tool and not an obstacle. If you've never used an IDE before, then we recommend you just go ahead and install PyCharm. That's 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, then you can reevaluate the situation.

Tabs or Spaces

Python is sensitive to the use of **tabs** and **spaces**. You shouldn't mix them! A good standard is to use four spaces to indent your code. If you decide to go for a text editor, then be sure to configure it to respect Python's stylistic choices. Notably, Notepad++ comes preset to

> use tabs instead of spaces, which is a problem if you ever copy and paste code from other sources.

Chapter 3

Writing the First Driver

3.1 Introduction

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

You 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-based messages**. The idea is that the user sends a specific command, usually in the form of a message, and then the device answers with specific information in the form of another message. Sometimes there won't be an answer as the message you sent simply tells the computer to perform an action, such as setting something automatically or switching itself off. Other times, the message you get back contains the information you requested.

For an idea of what these commands might look like, you can check the user manuals for devices such as oscilloscopes or function generators. Both Tektronics[1] and Agilent have complete sets of instructions. If you search through their websites, you'll find plenty of examples. A command that you can send to a device may look like this:

```
*IDN?
```

This is asking the device to identify itself. An answer to that request would look like `Oscilloscope ID############`. In this chapter, you're 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. Examples of these types of devices include oscilloscopes, lasers, function generators, lock-ins, and many more. The PFTL DAQ device that works with this book is also included in this category. If you obtained the book online and not as part of a workshop, then you can either build your own device, or you can contact us and we may be able to offer you one that's preprogrammed[2].

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

[2] courses@pythonforthelab.com

> **Device Drivers**
>
> There's an entire world of devices that do not communicate through messages, but that instead specify their own drivers. These are typically devices like 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're already confident in programming message-based devices and are ready to move on to non-message based ones, then you can check out the Advanced Python for the Lab materials.

Remember, **message-based** refers only to how the device exchanges information with the computer, not to the actual connection between them. It's possible to connect a message-based device via RS-232, USB, GPIB, or TCP/IP. However, beware that this 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 device manual and see how it's controlled. In this chapter, you're going to build a driver for a message-based device.

3.1.1 Understanding this chapter's scope

In the introduction, you saw 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, you'll learn everything you need to perform your first measurement. However, keep the onion principle in mind, which tells you that you should always be prepared to expand your code later on should the need arise.

3.2 Communicating with the Device

To communicate with the PFTL DAQ[3] device, you're going to use a package called `PySerial`, which you should have already installed if you followed Chapter 2. The first thing you can do is list all the devices connected to your computer to identify the one you're interested in. Plug your device into the USB port on your computer. You need to connect the PFTL DAQ device through the micro USB port closest to the power jack, also known as the *programming port*. Then, run the following command in a terminal, making sure that you're in your virtual environment containing the required packages):

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

> **Warning**
>
> From now on, you won't see explicit instruction telling you to open a terminal to run a command. If you see a command that starts with a dollar sign $, it means that you should run it in the terminal.

[3] PFTL is the shorthand notation for Python for the Lab

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

```
COM3
```

If you're on Linux you'll 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, and note which ones appear. Then, plug it back in and list the devices. The new one is the device you want.

Now it's time to start working with the device! Start Python by running the following command:

```
python
```

You can start working directly from the command line. First, you're going to import the package you need for communication:

```
>>> import serial
```

Interpreter Symbols

Earlier you learned that you must run everything prepended with a $ in a terminal. Lines prepended by `>>>` are lines that you run in the Python interpreter. Note that there's no need to type the `>>>`, as the Python interpreter prints it out automatically.

Then you can open a line of communication to the device. Bear in mind that you must change the port number to the one you retrieved earlier:

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

Now you're ready to get started exchanging messages with the device. Before you read the explanation of each line, try to run the code yourself first. Note that 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 though they're short, many things are going on in the code blocks above. First, you import the PySerial package. Note that you're actually importing `serial` and not `PySerial`. Then you open the specific serial port that identifies the device. Keep in mind that serial devices can maintain only one connection at a time. If you try to run the line twice, it will give you an error letting you

know that the device is busy. This could happen if, for instance, you try to run two programs at the same time, or if you were to start Python from two different terminals.

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

The other caveat 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 the letter *A* is. The serial communication can only send a stream of 1s and 0s. Therefore, you need to transform any information you're trying to send, such as `'IDN'`, into bytes before you can send it to the device. You'll see a lengthier discussion about encoding strings and what that means at the end of the chapter, in section 3.11.

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

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

In the manual, the first thing you 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 you know how to terminate commands, you can go ahead and see the list of options.

Many devices (but not all) follow a standard called SCPI[4]. The standard makes devices easy to exchange, because the commands for all oscilloscopes are the same, as well as for all function generators, and so forth. Moreover, the SCPI standard follows a structure that makes messages modular and easy to understand. 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 you know how to get started with serial communication, it's time to move to more complex programs. Typing everything on the Python interpreter is inefficient and takes too much time. It's time to start working with files.

3.2.1 Organizing files and folders

When you start a new project, it's always a good idea to decide how you're going to organize your work. In the previous chapter, you've set up an environment for developing a program. That's the first step to be organized. The second step is deciding where you're going to save the files you need to write your program. The general advice is to have a folder for all the programs, for example, `Programs`. Inside of that folder, each project you work on has its own sub-folder, such

[4] https://bit.ly/wiki_SCPI

as `PythonForTheLab`. Each person has their own way of organizing themselves, but from now on, every time we talk about creating a file, we're referring to that base folder for the project.

3.3 Writing a Basic Python Script

The code you've developed above can also be written as a Python script, which you can run from the command line without needing to re-write everything. Create an empty file called `communicate\_with\_device.py` and add the same code you saw 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 you can run the file:

```
python communicate_with_device.py
```

Changing Directories

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

What happens when you run the file

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

You want to know first when the program hangs, and then you can see how to fix it. Edit the file and add some print statements to check until which point the program 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's hanging during the `.readline()` execution. Can you understand what's going on?

There's something very different between writing in the Python interpreter and running a script, and that's the time it takes to go from one line to the next. While you type, everything happens slowly, but when you run a script, everything happens incredibly fast. Now, the `.readline()` is waiting to get some information from the device, but the devices aren't generating it. This means that the problem comes earlier when you send the `IDN` message. The command is not wrong in and of itself, but what's happening is that between opening the communication with the device and sending the first message, you're not giving it any time.

When you establish communication with most devices, there's a small delay until you can start using it. In this case, you must add a delay between starting the communication and sending the first message. You 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 (which you can also input as fractions) before continuing. You can try lowering the number until you get the minimum possible value. Still, in most typical cases, you start the communication only once, so waiting 1 second or .5 seconds won't have a significant impact on the overall execution time.

Reading an analog value

Before you continue, it would be great to also read a value from the device, not just the serial number. If you refer again to the manual, you see that the way of getting an analog value is using the `IN` command. You can modify the code on your previous program to read a value from 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're reading won't make much sense, especially if there's nothing connected to the input number 0; it's just noise. But it's an excellent first step! You can acquire a value from the real world using the device. You'll take care of all the things that you need to address in the following chapters. In the meantime, try your hand at the following exercises.

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're using the old RS-232 (also simply known as *serial*), the number refers to the physical number of the connection. On Windows, it's something like COM1, whereas on Linux and Mac, it should be something like /dev/ttyACM1. In modern computers, there are no RS-232 connections, and most likely, you have to use a USB hub for them. This means that there's no physical connection straight from the device into the motherboard. The numbering can change if you plug or 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 or unplug the device while it's being used, the port you get the second time will likely be different. The second time you run the program, you'll need to update the information.

? Exercise

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

3.4 Preparing the Experiment

Before you move forward with programming, you first need to set up the measurement you want to perform and determine what you need to achieve it. This book revolves around the idea of measuring the I-V curve of a diode. If you're not too familiar with electronics, don't worry! It's not essential for you to be able to follow the book. You can just copy the connections as shown below. If you're a bit more familiar with electronics, then it's worth explaining what you're 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's easy to have visual feedback on what is going on. On the other hand, you can't measure current directly. First, you need to transform it into a voltage. If you're familiar with Ohm's law, then you remember this relationship:

$$V = I \cdot R \tag{3.1}$$

Voltage is current times resistance. In other words, if you want to transform a current to a voltage, you just need to add resistance to the circuit.

To perform the experiment, you need to apply a given voltage and read another 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 you to apply a voltage and read a voltage are as follows:

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

? Exercise

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

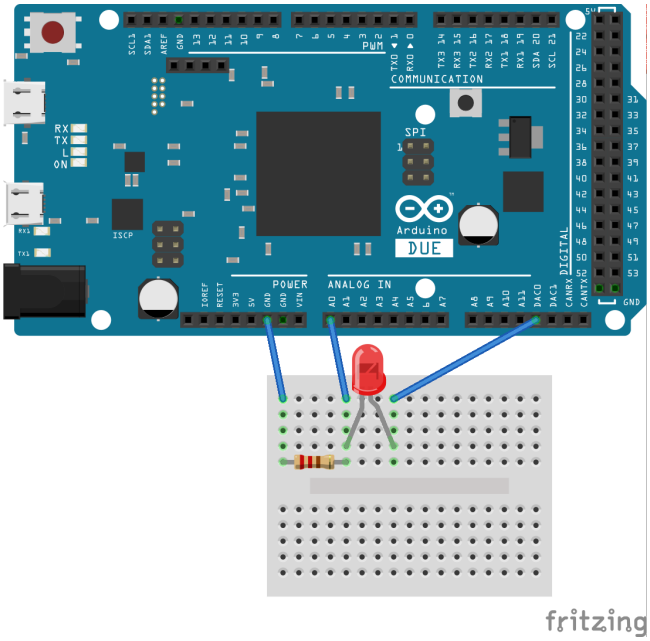


Figure 3.1: Schematic of the connections to perform the experiment

3.5 Making Improvements

You 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, this becomes very unwieldy. The PFTL DAQ device is an exception because it's minimal, but there are still a lot of possible improvements.

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

Since you know how to communicate with the device, you 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, and they're easy to document and to understand. Moreover, it's quite simple to expand on them later on. If you're not familiar with what classes are, check 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 to communicate 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 you rerun the code, you get the serial number of the device. Let's go line by line to understand what is going on. First, you create a class, and you define what you want to happen when you call `Device()`. In the `__init__` method, you specify that the class needs a port, and you 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 you sleep for one second to give time for the communication to be established.

The second method, `idn()`, just repeats what you have done earlier: you write a command, you read the line, and you return the output. If you look at the few lines at the bottom of the file, you now see that this way of working with this class is simpler. You 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 the 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 it's very useful for not overflowing your devices with useless requests for data.

Reading and setting values

You have just developed the most basic class, one that allows you to start communicating with the device and reading its identification number. You 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 you 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's 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. You 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()
```

However, it won't work, because even though you're adding the line ending, you're missing the `b` that you were using in the other examples. On the other hand, say you try to do something like this:

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

This will also fail, because `format` only works with strings, and as soon as the `b` is added in front of a string, it's encoded to bytes. This means that you 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 you form the message you want to send to the device, then you **encode** it, which is the same as adding the `b` in front of a string. Then you write it to the device. After writing, you return the line with the value. You 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`. You 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)
```

You 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 you 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 you read the analog input, you get the same value you passed to the analog output. It does not matter if it's 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, you get the same value that you set earlier.

This question is very tricky and requires that you 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 this method, you're just writing to the device and not reading from it. The message waits in the queue until the next time you read from it, and this happens when you try to read an analog input.

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

To solve the problem while setting the output, you 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()
```

You are not doing anything with the information you get. You just clear it from the device.

Transforming bytes to strings

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

```
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. You 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')`.

3.5.1 Abstracting repetitive patterns

There's a programming principle called **DRY**, which stands for *Don't Repeat Yourself*, which many developers use to help them build programs. Sometimes it's clear that code is repeating itself, for example, if you copy-pasted some lines. However, oftentimes the repetition is not about code itself but a pattern. DRY is not a matter of just typing fewer lines of code. It's a way of reducing errors and making the code more maintainable. Imagine that after an upgrade, the device requires a different line ending. You would need to go through all your code to find out where the line ending is used and change it. If you would specify the line ending in only one location, changing it would require that you just change one line.

Let's use the DRY principle to refactor your code. First, you can specify the default parameters for your device. They will be all the constants that you need in order to communicate with it, such as line endings. You 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's a lot of new information in the class. You 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). You also specify that you want to use ascii to encode the strings and that the baud rate is 9600. This value is the default of PySerial, but it's worth making it explicit in case newer devices need a different option. You also specify timeouts, which are allowed by PySerial and would prevent the program from freezing if writing or reading takes too long.

It's normally good practice to separate the instantiation of the class with the initialization of the communication. One task is to create an object in Python, and the other is to establish communication with a real device. Therefore, you 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 you change the number of arguments of a method. When you do this kind of change, it's called **refactoring**. It's a complex topic, but one of the best strategies you can adopt is to not change the number of arguments functions take and to keep the output the same.

In the class, the `__init__` definition looks the same, but its behavior is different. Now, it just stores the `port` as the attribute `self.port`. Therefore, to start the communication with the device, you need to do `dev.initialize()`. You can also see that you have used almost all the settings from the `DEFAULTS` dictionary to start the serial communication.

After you do these changes, you should also update the code you 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. You have to improve the rest of the class. You already know that for message-based devices there are two operations: **read** and **write**. However, you will only read after a write, since you should always ask something from the device first. It's possible to update the methods of the class to reflect this behavior. Since all the commands of the device return a value, you 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, you take the message, append the proper termination, and encode it as specified in the `DEFAULTS`. Then, you write the message to the device exactly as you did before. Finally, you read the line, decode it using the defaults, and strip the line ending. Now it's time to update the other methods of the class to use the `.query()` method you have just developed. Let's start with `.idn()`, which now looks like this:

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

You can do the same 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's something that happens very often in more extensive programs, and being able to identify those patterns can be the difference between a successful program and something only one person can understand. Note that even if you have changed the methods for identifying, reading, and setting analog values, there's no need to update the example code.

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

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

You now remember why this happened, but it's very likely that in the future, either you or someone else using your code runs across this error message that appears, and it's incredibly cryptic. Therefore, it's recommended that you do the following:

? Exercise

Improve the `.query()` method to check whether 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 you develop code, you must always keep an eye on two people: the future you and other users. It may seem obvious now that you must initialize communication before attempting anything with the device, but in a month, or a year from now, when you dig up the code and try to do something new, you're going to be different person. You won't have the same ideas in your mind as you do right now. Adding safeguards is a great way of preserving the integrity of your equipment. It also cuts down on the time it takes to find out what the error was.

There's only one last thing that you're missing. You have completely forgotten to add a proper way of closing the communication with the device. You can call that method `.finalize()`:

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

You first check that you have actually created the communication by verifying that the `rsc` is not `None`. Then, you can update your 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()
```

You may wonder why things worked out fine before, even though you 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 you will no longer use it. However, this is not always (for instance, 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, you must always check whether the port changed.

3.6 Interacting with the Real World

Up until now, everything you've completed may have looked like one big programming exercise, but now it's time to start interacting with the real world. As you know from reading the manual, the PFTL DAQ device can generate analog outputs, and the values you can use go from 0 to 4095. You can slightly expand 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's 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)
```

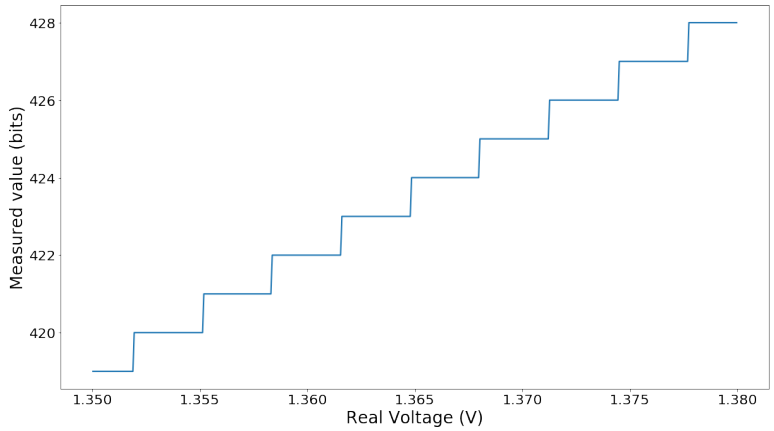
With this simple code, you can switch the LED on and off 10 times, and print to screen the values that you're reading when it's on or off. There are two things to note: first, you are switching it ON by using a value of 4000. You have selected this because it's high enough to switch the LED on, but it has no units, since it's not a voltage. The same goes for the value reported by `.get_anlog_input()`, which is just an integer, though you have no idea what it means yet.

Before you can proceed, you must understand how to transform analog signals to digital values and vice versa.

3.6.1 Digitizing Signals

Almost every device that you'll find in the lab transforms a continuous signal to a value that can be understood by the computer. The first step in doing this is to transform the quantity you're interested in into a voltage. Then, you 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 so you have discrete steps in your data.

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



The figure above shows what the digitization 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's a whole set of ramifications regarding visualization, data storage, and more.

In particular, the PFTL DAQ has different behavior for reading than for setting values. The output channels take on 4095 (or $2^{12} - 1$) different values. In other words, they work with 12 bits

instead of 10. Knowing the number of bits also allows you to calculate the minimum difference between two output values:

$$\frac{3.3\,\mathrm{V} - 0\,\mathrm{V}}{4096} \approx 0.0008\,\mathrm{V} = 0.8\,\mathrm{mV} \tag{3.2}$$

The equation above shows how the resolution of the experiment is affected by the digitization of the signals. You can't create voltages with a difference between them that's below $0.8\,\mathrm{mV}$, and you're not able to detect changes below $3\,\mathrm{mV}$. Later in the book, you'll come back to this discussion when you need to decide some parameters for visualizing your data.

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

Exercise

You have used a for-loop to switch on and off the LED, and you 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.

3.7 Performing an Experiment

At this stage, you can easily communicate with the device. You can set an output and measure a voltage. This means that you have developed everything that you need to measure the current that goes through the LED. You only need to combine setting an analog output and then reading an analog input. Since you're 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 `range()` function allows you to do this:

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

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

Exercise

Improve the method so that you can read analog values and store them once the measurement is complete. Returning the values can be a good idea so that you 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. You'll 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, your code is more straightforward to use. It would be simple to share it with a colleague who has the same device, and they'll be able to adapt and expand on it according to their needs.

You 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 you can just use that. 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. Interestingly enough, this is precisely the topic of the following chapter.

Remember the Onion: In the introduction, you learned that you should always remember the onion principle when developing software. If you look at the outcome of these last few exercises, you'll notice that you're failing to follow the principle. You 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 you 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 you'll see in the next chapter.

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

3.8 Using PyVISA

A few 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 you to communicate with devices independently from the communication channel selected and from the back end chosen. Different companies have developed different back ends, such as NI-VISA or TekVISA, but they *should* be interchangeable. The back ends are generally hard to install and do not work on every operating system, but they do allow you 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, you can use a library available for Python called `pyvisa`. There's also a pure Python implementation of the VISA back end called `pyvisa-py`, which is relatively stable, even though it's 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, you'll see how to get started with `pyvisa-py`, but it's not a requirement to follow the book. We decided to show it here to have it as a reference for other projects.

First, you need to install `pyvisa`, which is a wrapper around the VISA standard. You can do this with pip:

```
pip install pyvisa
```

Alternatively, you can also install it with conda:

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

If you don't have a VISA back end on our computer, you need to install one. The easiest one to use is the Python implementation. Again, you can install it with pip:

```
pip install pyvisa-py
```

Here's how to install it with conda:

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

There is also an interesting dependency missing: PySerial. Neither `pyvisa` or `pyvisa-py` depend on PySerial. If you're going to communicate with serial devices, you should install that package yourself (and the same is true for USB, GPIB, or any other communication standard.) The documentation for pyvisa-py[5] has handy information.

To quickly see how to work with PyVISA, you can start in a Python interpreter before moving on 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'
```

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

Pay attention to the `.query()` method that you use to get the serial number from the device. We didn't develop it. PyVISA already took care of defining it for you. Not only does PyVISA take care of the query method, but they also have plenty of options that you can use, such as transforming the output according to some rules or establishing the write termination. If you want to follow the PyVISA path, you could start by reading their documentation[6].

Why didn't you start with PyVISA?. There are several reasons. One is pedagogical. It's better to start with as few dependencies as possible, so you can understand what is going on. You had to understand not only what commands are available, but you also had to be aware of the encoding and line termination. We made explicit the fact that to read from a device, you 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.

[5] https://pyvisa-py.readthedocs.io
[6] https://pyvisa.readthedocs.io

3.9 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 few extra lines of code. However, there are many features that you may want but that someone needs to develop. For example, imagine that you want to limit the number of times the output voltage can change, or you don't always want to write the same value to the device, since the first time was enough.

You may want to establish some limits, for example, to the analog output values. Imagine that you have a device that can handle up to $2.5\,\mathrm{V}$. If you set the analog output to $3\,\mathrm{V}$, you would burn it. Fortunately, there are some packages written especially to address this kind of problem. We are only going to mention one because it's a project that you can collaborate on called Lantz[7]. You can install it by running:

```
pip install lantzdev
```

> **About Lantz**
>
> We introduce Lantz here for you to see that there's much room for improvement. However, through this book, you're 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 that's not what you want for an introductory course. If you're 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 you'll simply see 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 you'll see here, but with these basics, you can start off in the proper direction. You'll also notice that some of the decisions you made earlier were directly inspired by how Lantz works.

Let's first re-write your driver class to make it Lantz-compatible. Start by importing what you need and defining some of the constants of your device. You also add a simple method to get the identification of the device. Note that the first import is `MessageBasedDriver`, precisely what you saw at the beginning of the chapter.

```
from time import sleep

from lantz import MessageBasedDriver, Feat


class MyDevice(MessageBasedDriver):

    DEFAULTS = {'ASRL': {'write_termination': '\n',
```

[7] https://github.com/lantzproject

```
                        '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, note that you're importing a special module from Lantz, the `MessageBasedDriver`. Your class `MyDevice` inherits from `MessageBasedDriver`. There is no `__init__` method in the snippet above. The reason for this is that the instantiation of the class is different, as you'll see later. The first thing you do in the class is to define the `DEFAULTS`. At first sight, they look the same as the ones you have defined for your driver. The `ASRL` option is for serial devices. In principle, you can specify different defaults for the same device, depending on the connection type. If you were using a USB connection, you would have used `USB` or `GPIB` instead of, or in addition to, `ASRL`.

The only method included in the example is `.idn()` because, though simple, it already shows some of the most interesting capabilities of Lantz. First, you can see that you have used `.query()` instead of `.write()` and `.read()`. Indeed, Lantz depends on PyVISA, so what's happening here is that under the hood, you're using the same command that you 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. This is called a **decorator**, and it's one of the most useful ways of systematically altering the behavior of functions without rewriting them. Without going too much into detail, a decorator is a function that takes another function as an argument. In Lantz, when you use this decorator, it checks the arguments that you're 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 you 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, you need to define new methods. If you're stuck with this 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 you have done may end up being a bit confusing for people working with Lantz and with instrumentation for the first time. When you use `Features` in Lantz, you 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. You have to trick Lantz because your device doesn't have a way of knowing the value of an output. When you initialize the class, you create an attribute called `output0` with a value of `None`. Every time you update the value of the output on channel 0, you're going to store the latest value in this variable.

The first method reads the value, pretty much in the same way as the `.idn()` method. The main difference here is that you're 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 you to define the first method, also called a `getter`. It's the reason why you have to trick Lantz, and you couldn't simply define the `setter`. On the other hand, if the `setter` is not defined, it means that you have a read-only feature, such as with `.idn()`. The second method determines how to set the output and has no return value. The command is very similar to how the driver you developed earlier works. Once you instantiate the class, you can use the two commands like so:

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

Even if the programming of the driver is slightly more involved, you 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're 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. Try and see what happens if you try to set a value outside of the limits of the device. In other words, try something like `dev.set_output0 = 5000`.

The method you developed works only with the analog output 0. It means that if you want to change the value of another channel, you'll have to write a new method. This is both unwieldy and it starts to violate the DRY principle. If you have a device with 64 different outputs, it becomes incredibly complicated to achieve a simple task. Fortunately, Lantz allows you 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 only has two outputs, you initialize a variable `output` with two elements. The main difference here is that you 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 you can send to the device. The last `1` is there just to make it explicit that you take values in steps of 1. You can use the code in this way:

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

Now it makes much more sense, and it's even cleaner than before. You can also check what happens if you set a value outside of what you have established as limits. The examples above only scratch the surface of what Lantz can do. Sadly, at the time of writing, the documentation for the latest version of Lantz is missing. The best starting point is the code repository: https://github.com/lantzproject.

With the examples above, there's one small step for you to take in order to understand how to solve the following:

> **? Exercise**
>
> Write a `@DictFeat` that reads a value of any given analog input channel.

3.10 Conclusions

You have covered many details regarding communication with devices. You have seen how to start writing and reading from a device at a low level, straight from Python packages such as *PySerial*. You have also seen that it's handy for you to develop classes and not only plain functions or scripts. You 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's impossible to cover all the possible scenarios that you're going to observe over time in the lab in just one book. You may have devices that communicate in different ways, or even 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's going to be much easier for you 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're using. Sometimes manufacturers will 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. This is not only to prevent damage to your 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 for the packages you are using. PySerial, PyVISA, PyUSB,

and Lantz 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, since they'll very often be able to give you a hand with your problems.

3.11 Addendum 1: Unicode Encoding

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

A standard that appeared several years ago is called **ASCII**. The ASCII table allows you to transform 128 different characters into binary. Characters also include punctuation marks such as periods (`.`), exclamation marks (`!`), or semicolons (`:`), as well as numbers. 128 is not a random number, but it's 2^7. For the English language, 128 characters are enough. But languages such as Spanish, which have characters such as `ñ`, or French with its different accents, or even the myriad 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 you write a text in French, using a particular encoding, and then you share it with someone else. That other person does not know which encoding you used and decides to decode it using a Spanish standard. What will the output be? It's very hard to know, and it will probably very hard to read for that person. This isn't even considering what could happen if someone writes in Thai and shares it with a Japanese speaker, for example!

This still happens with some websites that handle special characters very poorly. Depending on how people configure their 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 **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 you want to send a command to a device such as the PFTL DAQ, you need to determine how to encode it. Most devices work with ASCII values, but since they overlap with the Unicode standard, there's 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 you want to choose how to encode a string, you can do the following:

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

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

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

```
var_ñ = 1
```

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

Chapter 4

Model-View-Controller for Science

4.1 Introduction

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

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

- What you develop should be readable by both current and future colleagues
- It should be easy to add solutions that have been developed by others
- The program should allow you to exchange 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)

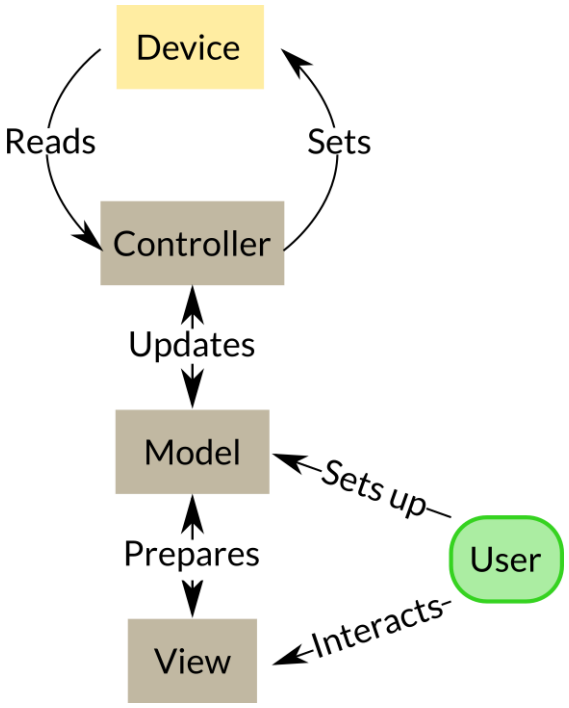
Let's take a look at each of these in turn. The first mantra isn't too challenging to understand in and of itself. The second mantra talks about solutions developed by others, meaning that often another developer may have already written a driver for a device, or perhaps a measurement script. Therefore, it should be easy to get code from other developers and use their software in your projects.

The third mantra discusses exchanging hardware, which is something that's often not valued until it happens. In most labs, there's always a legacy device that sooner or later breaks down, and you'll need to replace it. In another case, you might move to a different lab and need to continue your experiments with different hardware. There are patterns that you can follow to allow a simple exchange of devices. The last mantra speaks about context. Experiments can be very different, but the logic behind them is often quite similar. You measure the I-V curve of a diode, but this 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 not rules. They're just points on which you should reflect to try and recognize when you're departing from the path you wanted to follow when you started your project. In the

following sections, you're going to explore a design pattern for software that has many benefits when developing scientific software for controlling experiments. It's called **The Model-View-Controller for Science.**

4.2 Understanding the MVCs Design Pattern



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

A **Controller** for the purposes of this book can also be called a *driver*, which is responsible for communicating with devices. This can be a Python class you developed yourself, such as the one you built in the previous chapter, or it can be a Python package that was developed by someone else. For instance, many manufacturers provide 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 more and nothing less. For example, if a device can acquire just one data point at a time, then the driver shouldn't include a function for acquiring an array of data using a loop. You saw this briefly 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. This is where you determine how you're going to use a device for your 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 you would like to transform that information into voltages, then you could do this in the model.

Moreover, in the experiment itself, you measure voltage, but you can convert it into a current with Ohm's law. This behavior is particular to your experiment, and thus the option shouldn't be hard coded in the driver. The place to include this information is the Model. The main advantage

of splitting up *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's a second type of model called the **Experiment Model** in which you link different devices to perform a measurement. You could also use a single device, but you add the features that an experiment needs (for example, saving data, plotting, analyzing, or transforming units). With straightforward cases, the boundary between the device model and the experiment model can be blurry. Still, when you're dealing with several devices or more complex workflows, it becomes much clearer. At the end of the book, we give a reference to some of the projects we've worked on, which can be a good source of inspiration.

The *View* is the place where you can place everything related to how you show data to the user and how the user can change the parameters of the experiment. In practice, it's a 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 on to the model to acquire the data, save it, and plot it. Note, however, that the user interacts through the View with the Model, but never directly interacts with the Controller. It's also essential for you to keep in mind that all the logic is implemented in the Model. For example, if you save the data to files, the procedure to create new file names should be specified in the Model and not in the View.

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

Why Separate the Code?

For those who are new to developing code for the lab, it may be hard to understand why and how to split up the *Controller* and the *Model*. When you have only one device that you use for only one goal, one that is as simple as the device you're 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 own code, it's crucial that you separate the capabilities of your device from the logic of your experiment. If you don't do so, then all your code will only work when performing 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 those in the lab are. Therefore, the MVC pattern definition can change from one field to another. The details are not that important; once you establish a structure and follow it, then everyone else will be able to understand quickly what the code is doing and where. Once you understand what each component is, you will very quickly understand where you need to change the code to solve a bug or add new functionality.

4.3 Structuring the Program

In the program you're developing in this book, you follow the MVC design pattern quite literally. This means that you have to create three folders called *Model*, *View* and *Controller*. In the previous

chapter, you've 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 *Model* folder. Inside the file, define a class called `AnalogDaq`. What methods do you think should belong to the device Model?

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

> **? Exercise**
>
> Create a file in the *Model* 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, so don't worry about making it work, but focus on 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 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 output channel number 1, and other times to channel number 0. You don't want to change the code every time you change where you've connected the LED. The same happens, for instance, with the channel you want to monitor, or the time delay between steps. You'll include all this behavior in the View that you'll develop in the last chapters of the book.

Now that you've started to split the code into different folders and files, it's essential for you to learn how to make programs that can use the code that's located in those separate files. This process is called **importing**, and it's the focus of the next section.

4.4 Importing Modules in Python

In the previous chapter, you saw some lines that look like this:

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

The first line imports the NumPy package, but changes its name to `np`. Changing the name makes it easier for you to work with libraries, since you can type fewer letters (`np` instead of `numpy`). The second line is importing one specific function from a package called `time`. It's

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

```
sleep(1)
```

With *Numpy*, you will need to specify which module you want:

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

If you know you only want to use `random` from NumPy, then you can also import and use it like this:

```
from numpy.random import random

random(1)
```

You may wonder why you would import all of NumPy if you're just using one of its functions. Well, the name *random* is not defined solely by NumPy. Python also provides its own `random` module. You can import it like this:

```
import random
```

If you needed both *random* functions (the one from NumPy and the one from Python's standard library) in the same program, you would have a clash. How could you be sure that you're using NumPy's function and not Python's? What's more, you might even define your own random function, and you would like to be able to choose which one to use. More importantly, you want to avoid generating unexpected behavior because of redefining functions without realizing it.

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

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

Now you have the controller available for use. You can then do the following:

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

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

To avoid having the code run as soon as an import happens, you must change the file **pftl_daq.py**. You can add one line of code at the end of the file, as well as 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, you first must close Python by running the `exit()` command and then open it again. Python won't import the same modules twice, so it won't realize that the file with your Controller changed. After you restart Python, you can try to import the Controller again. Now, things are going to look fine, and you can use the `Device` as you intended. You can leave the space at the bottom of the files to hold examples of how to use the code above. If you ever find the Device around, and you don't remember what you were supposed to do with it, you can always go to the bottom of the file and see how it works.

Every time you encounter a behavior that's not easy to understand, the best idea is for you to transform it into simpler components and explore them one by one. In this case, the import process may be a bit confusing. To understand it a bit more, especially with respect to how the line `if __name__` works, you 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, you can just type `python Controller.dummy_controller.py`. The output should be like what you see below:

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

What you see is that the entire code was executed.

Exercise

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

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

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

Python imports might be easier to work with than they are to understand, especially when you're trying to pay attention to all the different definitions. The Official Python Documentation[1] has an excellent chapter covering a lot of the ideas discussed. Many of the properties and behaviors can be learned through trial and error, though this can be very time consuming, and it may lead to unexpected errors.

There's one final remark that's worth mentioning about packages. Right now, you have three folders in your project: *Model*, *View*, and *Controller*. But what would happen if you added a new folder called, for example, *serial*? Next time you try to `import serial`, how would Python know whether it's supposed to import the PySerial library or your package? This can also happen the other way around. Perhaps you don't know that there's something already called *Controller* in Python, and you don't care about this module; you want to import your own modules.

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

[1] https://docs.python.org/3.6/tutorial/modules.html

Exercise

Create a folder called *random* and inside of that create a new file called **test.py**. Define a function inside the file. It's not important what it does. Then, save it. Start Python and see what happens if you run `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.

The init files of packages allow you 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 Changing the PATH Variable

There's still a huge difference in the way you perform imports when you compare your own code to a library such as NumPy. Packages like NumPy can be imported regardless of where you've started the Python interpreter. You can go to any folder in the Terminal, start Python, and `import numpy`. However, you can only import your package if you're in its directory. If you're sitting in a different folder and try to import the Controller, you 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 searches, but Python is not aware of your package yet. One way to let Python know where your package is located is to add the folder to a system variable called `PYTHONPATH`.

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

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

You have to replace `/path/to/PFTL` with the full path to the folder where you're keeping the code. After running that, you can type `from Controller import *` wherever you are in your computer, and Python will find the appropriate folder. On Linux, this change is not permanent, so you'll need to rerun the command the next time you start the Terminal. You can modify environment variables permanently, but that goes too much into the details of the operating systems. We leave this as an exercise for the reader.

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

4.6 Finalizing the Layout

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

However, if you create extra folders next to the three main ones, then the structure of the program will start to become polluted. It won't be clear what's part of the program and what's a user-specific setting. Therefore, it's a common practice to make a folder to hold the program itself. Then, next to it, you create an extra folder to contain any non-essential elements. In this case, the folder structure could 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*. This last folder holds the *Model*, *View* and *Controller* folders with all the code that you've developed up until now. The folder *PythonForTheLab* also requires an **__init__.py** file, because it's your main package. This folder structure would allow you to do the following:

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

This code is cleaner than importing the Controller directly because it also allows you to, in principle, have different programs at the same time and import only the elements you need from each one. For example, you 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 those devices.

4.7 Conclusions

Following a pattern when you're programming makes your code much easier to understand, maintain, and explain to others. Patterns, however, are not set in stone. Sometimes there's no clear divide between what you should develop and where you should put it. The important thing is for you to be consistent. For example, if at some point it's decided that controllers should handle real-world units, then all controllers should do so. If, on the other hand, you decide that models should handle units, then all models should do so, 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 a person were to change labs or experiments. As time passes, you start accumulating a collection of tools, and being able to

transfer them from one experiment to another makes it much faster for you 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, you manage to quickly communicate with the device. You saw that you can acquire data and switch the LED on and off. If the measurement was a one-off task, then you would be finished. However, if you'd like to start changing parameters, or if you want to use the code for another experiment, then you need to keep going. Expanding the code is what you're going to do next. In these sections, you have explored a sustainable way of following the *Onion Principle*. Factoring also tackles concerns about the speed at which you 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 for someone else. We can only lay out the foundations for what we consider a robust solution. If you find a different path, you're always free to follow it, but never forget the bigger picture.

Chapter 5

Writing a Model for the Device

5.1 Introduction

Diligent thought is the key successfully developing a model for a device. You need to know what you expect from the device how you are planning to use it. The first time you start to think about models, it can be quite overwhelming, because you'll need to anticipate your future needs. For example, the PFTL DAQ device doesn't handle units. It would be great if the model would allow you to specify the output voltage instead of converting it to an integer, because that's what the driver uses. This requirement seems trivial and is probably the first one that comes to mind. Later, once you start using the program, you'll notice that some other useful options were missing, and you could have saved time had you thought about them.

There's no magical recipe to teach precisely how to develop models for devices. Each device and each experiment is unique; the best you can do is to focus on the task at hand. You 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, you may not see immediate gains from having models separated from controllers. Still, as soon as the complexity grows, this value becomes apparent.

> **Device Manuals**
>
> It's impossible to overestimate the importance of reading the manuals for your devices. Hardware in the lab is not the same as consumer hardware. Things can break, signals may not make sense, and you could run into a host of other issues. Always be sure that you understand the limits under which your components operate.

5.2 Developing the Device Model

The first thing you should think about is how you want to interact with a particular device. Of course, you would like to initialize it, set a voltage, read a voltage, and then finalize the device. However, you don't simply want to repeat what the Controller can do. Rather, when you initialize or finalize the device, you want to be sure that the output voltages are at $0\,\mathrm{V}$. In this way, you can ensure that no current flows through the LED unless you explicitly want it to do so. You also want to be able to input values in volts when setting an output, and get values in volts when reading a voltage.

With these things in mind, you can develop a skeleton for the Model. You can use empty methods to get an idea of what you'll need to develop, as well as the arguments and outputs of each method. Start by creating a file **analog_daq.py** in the *Model* folder and 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 you can add code to the file line by line. You'll start very similar to how you started the Controller. The `AnalogDaq` class takes `port` as an argument for initializing. The main difference is that you don't use PySerial directly, but instead you use the Controller. You can start improving your 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)
```

You initialize the class by storing the `port` and creating a `self.driver` attribute. Remember that the `Device` has a separate method for initializing. The `.initialize()` method now not only initializes the driver itself, but also sets the output voltages to 0. You haven't developed a way of setting voltages yet, but you can 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()
```

You first set the voltages to 0, and then you finalize the Controller. This is a clear example of how you might impose your own logic onto the device. In some cases, you won't want to set the voltage to 0 when stopping communication. Perhaps you're just switching on a laser, and you want it to stay on even if you switch off the computer. Or perhaps you're using piezo stages where it's not recommended to suddenly shake them by setting a different voltage; it's better to just leave a voltage applied to them. Scenarios like these show why adding these features to the Controller

would imply violating the separation of models and controllers. What you do with the voltages is part of the logic, not the device itself.

Now that you have this code, you can also add an example to the end of the file, to show you 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 you do in these steps was discussed in Section 3.6.1. To set a voltage, you first need to transform a number in the range $0 - 3.3$ to an integer in the range $0 - 4095$. Then, you apply it:

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

You 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
```

That is all that's needed. You can now run the code and see that you can read and set voltages. Because of how the experiment works, the values you get at the analog input are going to be very small (in the order of a few tens of millivolts), but it's enough to be detected by the PFTL DAQ.

5.3 Developing the Base Model

In this book, you're working with only one device, so you're using only one model. However, if you want to make your program compatible with more devices, you'd need to start developing models for each new device. Since you already have one working model, it would probably be more reasonable to copy it and adapt the methods based on what the new drivers allow you to do. Another option is to create a base class that all other models inherit from. In this way, you know that all the methods are defined. Perhaps they don't do anything, but at least they're there!

What you'll do in this section is not a requirement for you to keep going, but it's important for you to see this pattern. That's because sooner or later, when the program grows, it will become a handy approach. Create a file called **base_daq.py** inside the *Controller* folder and 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 **schematic** of what a model should contain. You added `pass` after every function definition to take care of the places where nothing happens. You can see that you also specify the arguments that each method takes: `.initialize()` takes a `port`, `.set_voltage()` takes `channel` and `value`, and so forth. In programming, this is also called an **Application Programming Interface** (or API for short). The base class defines the interface that all the DAQ models use. There's an initialize method, a get and set analog, and a finalize. Just by looking at this simple example, you already know how things are going to work and what you need to do to make them run.

As an example, let's create a dummy DAQ that can generate random values when requested. Since it's not connected to any real device, it doesn't do anything else. You 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()
```

If you copy the example code from your 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 you set a voltage, initialize, or finalize the Model, nothing happens; but when you ask for a value, you get one. You see that the way of using this class is the same as the real Model, and it took you only 3 lines of code to develop! Perhaps in the future you might move to a more complex DAQ, such as an oscilloscope. If you maintain the same names for the methods of the Model, then everything will keep working in the same way.

? Exercise

Update the real Model to inherit from the base class.

5.4 Adding Real Units to the Code

Your Model for the PFTL DAQ device is working great, but it does have one problem. It allows you to set the output in volts and read a value in volts, but if you ever were to make the mistake of supplying the value in millivolts, then the program wouldn't work as expected. In real cases, it's tough to remember the units that every method should take. Sometimes you have very different outputs and inputs, with each one taking on different units. Imagine that you want to make a periodic signal, where perhaps the device asks for the frequency, perhaps for the period.

In Python, you can overcome the limitations of working with plain numbers by using a package called *Pint*, which allows you 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, you import the package and start the `.UnitRegistry()`. In principle, Pint allows you to work with custom-made units, but the fundamental ones are already included in the `.UnitRegistry()`. Then, because of convenience, you define the variable `meter` as actually the unit meter. Finally, you assign the value of 5 meters to `b`. In this case, `b` is of type `Quantity`. Therefore, it's not just a number, but a number with a unit attached to it. *Pint* allows you to convert between units, and this is how you create the variable `c`, which is 5 meters converted to inches. With this, things can get very interesting:

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

Even if the numeric values of `b` and `c` are different, they're still equal to each other, exactly as you would have imagined. You 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, you have always been transforming between units of the same type (for example, a length in meters to a length in inches). But Pint can also handle combined units like you would see 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.

Pint also has one more useful feature: it can parse strings to separate the units from the numbers. For example, you can do the following:

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

Having the ability to parse strings so easily will make your life much easier when dealing with user input.

Now you've seen how to handle *real* units in your code. However, the device still requires that you set the output using plain numbers. In the context of Pint, the number on its own without units is called the **magnitude**. To get the magnitude of a quantity, you 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
```

You start with a quantity called `current` of 5 A. If you just append the `.m` to the variable, you get the magnitude in whatever unit it's already expressed. If you want to be sure to get the magnitude in a specific unit, you use the command `.m_as()`. In this case, you will need to transform the user input to an integer. You won't need to assume it's in volts, since you 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)
```

You transform the value to volts and get only the magnitude. Then, you transform that value to bits, using `round()` to get an integer after the operation. You use that rounded value to set the output on the device. This 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 output 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.

You 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 been 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, you 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. You 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're hesitant about the impact that different unit systems can have, there's a great example involving a multi-million dollar satellite[1] that you can use as a reference. 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.

[1] You can check the article on LA times: https://bit.ly/la-times-mo

5.5 Testing the DAQ Model

At this point, you have a very functional program! You can handle units, and you've separated the logic of the units from the driver, meaning that you can easily share your code with colleagues (or even the rest of the world).

It's time to test your program. One of the reasons you created the *Examples* folder was to be able to add extra Python files that don't belong to your core program. In this folder, create a file called **test_daq.py**, and try to start using the Model to do some measurements:

```
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(current)
```

You can run the code above, but you'll notice that you're printing currents with units of volts. This can be very confusing for someone looking at your results, or even for your future self. You have to remember that you can transform volts to amperes by dividing them with the resistance you're using. If you have a 100 Ohm resistance, you 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 you run the code with the changes above, you will get the following error:

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

The error is descriptive, but it's still hard to understand if you don't know the underlying principles of Pint. The unit registry is a collection of rules that allows you to transform one quantity into another. Still, these rules belong to a unit registry. In principle, two distinct unit registries hold rules for different sets of units. This means that you can't convert units across unit registries. You need to use only *one* registry throughout the program. Right now, you're creating the registry in two different places: the device model and the example.

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

```
import pint

ur = pint.UnitRegistry()
```

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

```
from PythonForTheLab import ur
```

> **? Exercise**
>
> Improve the DAQ model to use the central unit registry and not the one defined locally.

> **? Exercise**
>
> Modify the example that you developed for testing the DAQ model so that it uses the central unit registry, and see that it works as expected.

After completing the exercises above, you should be able to run the example and get the values you wanted. You're getting currents in amperes, and you're setting voltages in volts. You're 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, you saw how to add the root folder of the project to the computer's PYTHONPATH, which allows Python to find your program and allows you to import 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, you may be overwriting something important if there are any name clashes between the program you develop and some other library on the computer.

Therefore, you can go a different route and add the folder to the path directly from within Python. This change is not permanent, and it's in place only while the program runs, but no longer. First, you need to learn how to identify the folder you want to add to the path. For this, Python offers a module called `os`. The code below looks cumbersome, but you'll see an explanation in just a moment. You 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__)))
```

Let's start from the innermost part of the function. The `__file__` variable is a way of letting Python know that you're interested in the current file, which in this case is `test_daq.py`. First, you get the absolute path to this file, including all the folders you'd need to traverse to get there. Next, you grab the directory that holds the file, which would be the *Examples* folder. Then, you

grab the directory that contains the examples, which in this case is the root folder or `base_dir` it's called here.

Finally, you need to add the `base_dir` to the path. For this, you use another package called `sys`, which is a wrapper for the operating system. It means that this package adapts accordingly to the operating system you're using to run the program. To add the folder, you only need to do the following:

```
import sys

sys.path.append(base_dir)
```

That's it! It doesn't matter if you modify the PYTHONPATH variable anymore, as you can always run the `test_daq.py` file.

> **The PATH**
>
> Since appending to path only works while the program runs, if you try to run the package files independently, then Python won't know where to find the modules. The test files you created in *Examples* are what are called **entry points**, and the program should be run directly from them.

5.7 Brainstorming a Real World Example

It might still be difficult for you to truly grasp the usefulness of models at this point. You may be tempted to define units and transformations in the Controller itself. After all, you're the only one who's going to be using the program, and only for one experiment. While this may be fine when you start out, at some point, it's highly like that your code is going grow, especially as the experiment progresses and gets more complex. As an example, at Python for the Lab, we've developed software for controlling a microscope using a camera. However, there were several cameras available, some which were more expensive and more 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, other times a Basler, and still other times 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 had an entire package called PyPylon to control their cameras. The controller layer, therefore, was not developed by us, but was provided for us.

At the model level, however, we made sure that all cameras would work in the same way. They all had 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 the Model we were importing. The rest of the code stayed the same. If you want to see the real code, you can head to the repository for the UUTrack project[2].

[2] https://github.com/uetke/UUTrack

5.8 Conclusions

In this chapter, you learned more about the *Model* component of the MVC pattern. You focused on adding features with regard to how the device works, such as switching off the outputs when you finish using the device. You've included real-world units by using **Pint** and learned some of its quirks regarding the unit registry.

You've also covered how to append folders to the PATH through Python. This allows you to run all the import statements you could want without having to alter the environment variables of the operating system manually. On the one hand, this is useful because your code runs unaltered on Linux, Windows, and Mac. On the other hand, you don't make any permanent changes to the configuration of the computer. For example, it's a possibility that at some point you could have two projects with the same name, projects that contain code you wrote for two different but very similar experiments, but you still want to be sure you're importing the correct one.

Chapter 6

Writing The Experiment Model

6.1 Introduction

In the **MVCs** design pattern, the *Model* is where you should place the logic and the decisions you make to perform a measurement. In the previous chapter, you started specifying the logic of how you're going to use the DAQ device. You 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.

You're still missing, for example, the possibility to perform a scan in a given range of voltages, as there's no way of saving data or the parameters used to generate it. You can develop these tools in a script, which is fine for getting started. At some point, however, you'll want to be systematic, and you won't want to keep editing a script every time you must perform a measurement. You'll want to save data consistently, or you'll want to enable other people to run their experiments based on your routines.

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

Splitting into Modules

With a simple device and experiment such as the one you use in this book, sometimes it's hard to put a limit on what should go in the device model and what should go into the experiment model. There are no strict rules. You should always reflect on what you believe is useful in the long run. Once you're 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 for a long enough tim. On the one hand, the models are going to lead to reproducible experiments. If you keep track of the parameters, you can go back to precisely the same conditions in which you've performed a specific measurement. On the other hand, having

a well-structured model is going to make it very straightforward for you to build a Graphical User Interface (GUI) on top of it, though you won't delve further into that subject here.

6.2 Writing the Skeleton of an Experiment Model

Every time you want to start developing a model, it's handy to start from an empty skeleton. It helps you to know what parts of the code you need to develop, what things you don't know how to do yet, and what things you still need to learn. It also allows you to have a quick glimpse of how you expect your program to be used. You did this for the device model in the previous chapter, and you can do the same for the experiment model in this chapter. First, create a file called **experiment.py** inside the *Model* folder, and then 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, you include a `.load_config()` method, which will rely on the `config_file` you specify at the `.__init__()`. Having a separate config file is useful not only to change parameters between measurements, but it helps you with your code. Especially when the number of things you need to remember grows, it's always handy to have a file where you can see how you named things.

You've also included a `.load_daq()` method. So far, you only have one DAQ to use, but you can assume that at some point, there may be more than one, so you should have a way of loading one or the other. This can also be a case of premature optimization, in which you anticipate a future that never comes. Nevertheless, it's pedagogically useful to define it here.

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

6.3 Writing the Configuration File

6.3.1 Working with YAML files

Before you keep developing code, you need to stop for a second to think about what you want your program to do. You need to think about what inputs you need for your program. For example, you need to know the port to which the device is connected. You also know that you 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 you can easily store all these parameters into a text file.

In the *Examples* folder, create a file called **experiment.yml**. You can create the file with any text editor, including the one you're using for editing Python files. The only difference is the extension *yml*. You're going to use this file to hold all the parameters of the experiment. The format of 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 for both a person and a computer. It has just a few rules. The most important one is that the indentation is **2** spaces. In the example above, there's a main element called **Experiment**. Everything that is indented relative to that element belongs to it. To read the file, you're going to use a package called PyYAML, which you installed in Chapter 2. You 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*'-')
```

You begin by opening the file using `open()`. You use the `with` command because it's convenient for working with files and other resources that have the same pattern of "open, do, close." YAML is then responsible for interpreting the information contained in the file. You store this information in a variable `e`, which turns out to be a dictionary.

To get the elements stored in dictionaries, you use keys. In the example above, there's a main key called `'Experiment'` with sub-keys `'name'`, `'range'`, and `'list'`. If you want to use one of those elements, you can type `e['Experiment']['name']`, for instance. 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. You can see that `'name'` is a string, but `'range'` and `'list'` are not. YAML automatically detects what kind of information you're storing, making your job easier if you already know what you 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, you can use `yaml.dump()` to save it. For example, you can define a dictionary with the following information:

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

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

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

You're using the `'w'` option to open the file, which means that every time you run the code, it overwrites the file, and you lose the previous contents. After running the code, you can open the file **data.yml** with any text editor and see that the contents are very similar to the one you created 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.yml** and check that they are the same as the file you saved.

Exercise

Modify some values in **data.yml** 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. What 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 you know how to work with YAML files, it's time to start thinking about the experiment again. You need to stop and think about what you need to know to perform an experiment.

Exercise

Create a new experiment.yml file in the *Examples* folder. Use what you've 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

You're using YAML files because of their simplicity, and because it's easy to map them to dictionaries in Python. The main advantage of having a separated config file is not only that it allows you to run different measurements while changing parameters as you desire, but also because it helps you keep your code well organized. You always know how to get the information you need, just by looking at the config file.

First, you need to decide what you're going to include in the configuration file. The information is not static, and it may happen that after working for a while, you realize there's some important parameter missing, or you realize something that you thought was important is not necessary anymore. Every time you find yourself deciding a value, that information should go into the config file. For example, if you need to decide which port the DAQ is connected to, that information should go into the config file. You're going to use the **experiment.yml** file in the *Examples* folder to store all this information. Below, you can see what the config file might look like. There are extra options included that you haven't seen yet, such as the name of 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.
```

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

You have to update the `Experiment` class to load the file. You're storing the filename within the class, so all your methods can be triggered without arguments, but this is just a stylistic choice:

```
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
```

You've used the same code you saw in the previous Section, but this time as part of a method. When you load the configuration file, it's stored as an attribute of the Experiment class, called `self.config`. Now that the experiment is starting to have shape, you can create a file to show how to use it and check whether you're doing things properly. In the *Examples* folder, you 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)
```

You use the strategy explained in Section 4.5 to let Python know where to find your program. The rest is easy to understand: you start an experiment, load the config file, and print the parameters that you've loaded. So far, you're not doing much, but it's a good start!

Now you have a consistent interface for loading the configuration file into your programs. This gives you a great deal of flexibility and makes the code nicely reusable and extensible.

> **Exercise**
>
> When `.load_config()` loads the configuration from a file, verify that some essential parameters are defined. For example, verify that there's 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 your experiment, you can start with the simplest approach. You import the model for the DAQ, and then you initialize it using the `.load_daq()` method. You can update the experiment model like so (noting that some parts of the code have been omitted 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 locate the information for the DAQ port. The main key is `DAQ`, and the sub-key is `port`. In this way, there are much fewer things you need to keep in your head. You can always go back to the file and read what you need to know. You've also decided to initialize the device right after loading it. You could have developed a separate method, but at this stage and for this experiment, there's no real need.

Since you've taken care of loading and initializing the DAQ, you can also develop the finalize method, which takes just one line:

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

You can update the **run_experiment.py** file to reflect the additions you've just made to the Experiment:

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

When you run the code above, you see that the output of `print(experiment.daq)` is not pretty, and it's hard to understand. You can update the device model to make the string that appears on screen nicer. You 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 you 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. This happens, for example, when you use the `print()` function. If you rerun the experiment, you see that the output is clearer than before.

> **String Representation**
>
> Adding a string representation to the classes you build is an excellent addition to the code, though it's not mandatory. You should be careful not to get sidetracked on details that don't bring you close to the goal of performing a measurement.

6.4.1 Including the dummy DAQ

In the previous chapter, you took a small detour to create a base class for the device models. In Section 5.3, you also created a dummy device, which is a fake model that outputs random numbers when requested. This means that you have two different DAQ devices that you can use, one real and one fake. Therefore, you can improve the Experiment model to accommodate the possibility of using one or the other. You're going to follow a non-standard approach here, one which does not comply with the general principles of Python, but that's convenient nevertheless. In the config file, you included the name of the DAQ, and it's 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 you're doing something that may seem strange at first sight. You're not importing Python modules at the top of the file, but deep inside a method. This is not standard, and in some contexts it's discouraged, but it's crucial to understand why you're following this approach. When working with real devices, the models may depend on drivers that are not installed on the computer. If you import the device model at the top of the experiment file, you may get an error because of a device you do not intend to use.

On the other hand, this behavior is discouraged because if there's a problem in one of the modules, you won't notice it until you're running the program. That can be a waste of your time and, more importantly, your data. For example, `AnalogDaq` depends on having a proper controller available. If you make the import at the top of the class, and the controller is not in place, you get an error even before you instantiate the experiment class. If you plan to use only the dummy model, then you 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.

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

> **? Exercise**
>
> In Section 5.4 you included units for the `AnalaogDaq` class, but you didn't add units for the `DummyDaq`. This 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 Performing a Scan

The core of the experiment model is being able to perform a scan, change the voltages on the output, and read 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 you're calling this `scan`, which is a fairly generic name.

> **? Exercise**
>
> Think of 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, you use the parameters that you've defined in the `Scan` Section of the config file. Since you already performed this type of measurement either from the command line or from example files, it's easy to adapt what you 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, you need to stop and think. After you introduced units to your model, you didn't try to perform a scan, and so you need to see how Pint works. You know that `start`, `stop`, and `step` all have units of volts or related. However, if you try to use the start, stop, and

step values as they are, you will fail. Neither the Python `range` function nor NumPy's `arange` function know how to deal with quantities. However, Pint allows you to transform a quantity to a plain number in given units by using the `.m_as()` method. Therefore, if you want to have a range of values over which to perform the scan, you can do this:

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

If you explore the `scan_range` variable, you 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; it's just how `arange` works. The documentation clearly states that `arange` generates values in the semi-open interval $[\text{start}, \text{stop})$. If you want to include the last point or not, it's 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'))
```

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

You could use numpy's `linspace`, which allows you to generate equally-spaced values if you provide a start, stop, and the number of points you want. This seems like a viable solution. You 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)
```

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

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

This is not spaced by $0.5\,\text{V}$, but it respects the start and stop values. The last option to which you can resort is to change the config file. You may have thought that defining the step was a good idea, but perhaps it's better to use the number of steps you want in your scan instead of the step size. Therefore, you 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 you proceed with the experiment class, you see that the array that NumPy generates has no units, and this makes sense because you 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 quite complete now. You use the number of steps specified in the config file. You're also using the delay, and so you import `sleep` at the top of the file. Then, you go through all the voltages. You also defined a variable called `scan_data` to hold the values as you measure them. It's a big achievement, and you need to reflect this in the file you were using for testing the Experiment class. You can update **run_experiment.py**:

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

When you run the experiment, you encounter a problem:

```
...

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

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

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

Now you can go ahead and run the experiment without errors. After you do the scan you can finalize the experiment, but there's no way for you to actually look at the data that you acquired. To make the data available to the outside world, or to other methods in the same class, you 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'))
```

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

```
[...]

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

Finally, you can see the data you acquired after the scan runs. This would allow you to do plenty of things like saving, plotting, and analyzing. However, you should not get too far ahead of yourself at this stage. You had laid out a plan for the Experiment class, and you should follow it as much as you can. The strategy of adding `self.` before a variable is beneficial, and this is the real power of objects. The idea is that you should use objects when you need to maintain *state*. It's not just running a function and returning a value, but it's updating the *state* of the experiment, or the device.

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

6.6 Saving Data to a File

After you perform a scan, you must save both the data and the *metadata*. The metadata is the information on the parameters used to perform an experiment that would allow you or anybody else to repeat the measurement. Let's start at the beginning, with simply saving data to a file. You're going to use the built-in functions of NumPy for this, because you have to save two arrays: `scan_data` and `scan_range`. Since you already know that Pint and NumPy interact in special ways, you can 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. You can go ahead and run the experiment, and it generates a file with two columns (which is the reason for the `vstack` and the `.T`) as well as a header specifying that the data is in units of Volts. However, if you run the experiment for a second time, the data gets overwritten. Moreover, you're saving the data to the same folder where you run the experiment, and this is a terrible idea. It would be better to save the data in a dedicated place. Therefore, you need to update the config file first, and then the saving method. The folder you're using for saving data here is only an example, and you should change it according to your needs:

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

One best practice to consider when saving data is to organize it by dates. You can create a folder with the date of the measurement inside the *Data* folder. First, you'll see snippets to understand what you need, and then you can add the modifications to the method itself. To get the date of today as a string, you can combine the `datetime` package with 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, and you know how to get the folder name based on today's date. Now you 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 `os` module allows you to take care of common problems that can appear when dealing with directories. For example, Linux uses the forward slash to separate the folder structure, while Windows uses the backslash. Also, you may define the folder with a trailing / or not. `os` takes care of all of this complexity for you. In the code block above, you join the data folder and today's folder and check if it exists; if it doesn't, you create it. The `makedirs` also creates the missing parent directories. For example, if the *Data* folder does not exist, the program creates it.

Next, you have to be sure you will not overwrite the files when you save new scans. Ideally, you would like the files to have a naming structure like `data_001.dat`, `data_002.dat`, and so on. Using what you specified in the config file, you 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
```

This seems like too much work just to format the name, but once you've done it, you can use it in all your experiments. You only need to know what the first available name is and then save the data there. You're now ready to save the data, as well as 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's also powerful. You start with a base folder and create a folder based on the current date, with the format Year-Month-Day. You format the data to make it easy to store as two columns of a text file, and you also add a header to explain what you're storing. You extract the base part of the filename and the extension as two separate variables, and you 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 zeros in front. You use a while loop to increase the counter until the first file is available, and then use that value to store the data. You also create another file with the same number and a similar name to store the metadata. In this 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 you can use it as the config file for the next experiment. If you want to repeat a measurement from the past, you just need to point the experiment to the config you saved from that date.

You 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()
```

Save for the part where you append the folder to the system path, you can run a relatively complex experiment in less than 10 lines of code, including saving data and metadata. This ability to reduce a complex problem into a relatively simple flow is thanks to the effort you put into defining the proper methods in the experiment and device models.

6.7 Conclusions

With this chapter, you've finished writing 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 are in place, parts like performing a scan or saving data to a file, you just need one line of code to run your script. 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.

There are still a few more things that you should do for your experiment. For example, you're still acquiring voltages instead of currents. You're not going to explicitly solve this problem in this book, because it's an excellent exercise for you to start thinking for yourself.

> **? Exercise**
>
> Update the config file to include information on the resistance that you're 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 you transform the arrays to being unit-less before saving them.

Some other topics that are still relevant include how to change the parameters of the scan directly from within the running script, which you'll cover in the next chapter. This would allow you to, for instance, scan the voltages with varying delays to understand if there's some form of hysteresis, or scans with an increased resolution around specific features. You've also neglected what would happen if someone were to forget to load the config file or the DAQ before starting a scan, for example.

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

Chapter 7

Running an Experiment

7.1 Introduction

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

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

7.2 Running an Experiment

In the previous chapter, you've seen how to run an experiment from a script. Each of the steps needed is triggered one after the other. You called that script **run_experiment.py** and it contained the following code (which skips 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's relatively straightforward, but this is not the only thing that you can do with the experiment. You can access the attributes of the experiment class while you're performing a measurement. For example, you could show the data once the scan finishes:

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

However, you could also change the parameters of the scan after loading the config file. Sharing information with classes in Python always occurs in one of two ways: you can read from them, but you can also write to them. Let's see what happens when you change the `config` attribute of the experiment before triggering the scan. You 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()
```

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

BPython

Running Python from the interpreter usually is cumbersome. There's another version of the interpreter called **bpython** which has auto-complete functions and allows you to edit code before running it. It's not recommended for anyone to run experiments from the interpreter, but sometimes it's the handiest solution to change values when desired.

In the code above, you've changed the value of the number of steps before triggering a scan. First, you've set it to 5 steps, and then to 10 steps. You can check the metadata and see that this information was saved. If you open the data file, you would also see a different number of data points. This flow suggests some exciting possibilities. For example, if you would like to see if there's any form of hysteresis on the measurements, you 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.

When you start developing complex logic in the running script, it's fair to wonder if it's better to put all that logic into the experiment class. As always, when you perform a task only once, knowing it will be a one-off task, then it's quicker to run it from a script. If, however, that one-off task becomes an essential part of your experiments, or you believe that others could benefit from performing the same measurement, then you 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 you work with units, you leave the values as a string, and only transform them into real quantities when you need them. If you explore the `.do_scan()` method, you'll see that you start by running the parameters through the unit registry. This means that if you want to change the values of, for instance, the start, stop, or delay, you only need to change a string:

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

7.3 Plotting Scan Data

You've learned how to do several things with the device, but you're not doing anything with the data after you save it. It would be a great addition to your workflow to be able to see what you're doing through a visual plot. For this step, you're going to use a library called PyQtGraph, which was developed mainly to generate fast data visualizations. This is exactly what you would need when you're dealing with experiments. Usually, those a bit more familiar with Python would become acquainted with matplotlib, which is an excellent tool for paper-ready plots, though a bit slow for real-time visualization. In any case, at this stage, you don't care about speed, so if you're more confident with another tool, then feel free to adapt the code to use it. In any case, you'll see PyQtGraph again later, when you build the user interface.

Making a simple plot with PyQtGraph is very easy. You can add the following after you 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 you must run the program slightly differently:

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

You must append the `-i` or the plot will close right after it appears on screen. It's simple, but it works. PyQtGraph also gives you some mouse interactions out of the box. You can drag the image, zoom in and out with the mouse wheel, and even transform the scale of the axes by right-clicking. You can make the plot more aesthetically pleasing 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, you can only plot data after the scan finishes. The `.do_scan()` method takes a relatively long time to complete. When a method or function works in this way, they're called **blocking** functions. The program can't continue until they finish. If you would like to monitor the progress of the experiment while it's running, then you need to find a way of running the scan in a non-blocking manner.

7.4 Running the Scan in a Non-Blocking Manner

Methods or functions like `.do_scan()` take a long time to run. Imagine that you would like to acquire a movie for one hour. In that case, the function would take one hour to complete. If you

were to run a very complex simulation or data analysis process, your functions could also take a very long time 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. Other times, they take longer to execute because the process is slow. When you 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 your computer can perform other tasks at the same time if you know how to make it do so.

Python has different options for achieving this. Still, the one that gives the best results, not only in terms of flexibility but also in terms of simplicity of implementation, is the *multithreading* module[1].

7.4.1 Threading 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 you to do is to start other threads at any point in time. The more threads you have, the more entangled the program will be. Python, however, does not run the threads at the same time. There's a tool called the **Global Interpreter Lock (or GIL)** that prevents two things from happening 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 at exactly the same time, but if one thread is waiting on a `sleep`, for instance, Python can go ahead and let another thread run. Whenever you start Python, you're starting a thread, also called the *main thread*, which lives through your program. From that thread you can start a second one, as you 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 you run it, you get the following output:

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

It's essential for you to understand what's happening in the code above, so let's dissect it step by step. If you run the function `func` on its own, you see that the numbers appear one by one.

[1]Another library which is gaining popularity is AsyncIO. It is, however, harder to efficiently implement it for the purposes of this book.

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

When you used `threading.Thread`, you were creating a new child thread. The `target` is whatever function you want to run on that thread. You can also pass arguments, as you did in the code block above, using `args=(3, )`. Once you create the thread, you have to start it by calling `t.start()`. Note that the `target` is `func` and not `func()`. If you would use `func()`, you 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's very important for you to be able to distinguish between the proper function and its output. To run a function on a separate thread, you use `target=func`. If you call the function instead, you would be sending its *output* to a thread, which is not useful. If things don't work, check whether there's an extra pair of parentheses `()` after `func`.

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

You 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. You've added one extra detail to your code,using `join()` to wait for the threads to complete. Now you can see that the program prints `'There'` after the function finishes. Using `join()` is usually a good idea in order to not finish the program and leave some orphaned threads.

Threads don't run in parallel

The fact that things can happen more or less at the same time does not mean the code is running in parallel. Threads have been around for a very, very long time. Even in single-core computers, users were able to switch from one program to another. You 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. Parallelization means running computations at the same time. In multi-core processors like the ones available in most computers and cell phones today, you can split tasks and run them at the same time in different cores. For truly running on different cores, Python offers the *multiprocessing* package. However, multiprocessing is challenging. The Advanced Python for the Lab book covers in detail how to leverage it for more extensive and more data-consuming experiments.

7.5 Threading for the Experiment Model

You've seen how to run a straightforward function on a different thread, but you can also see how more complex functions do not present a challenge. Going back to the experiment model, you can already test what you've 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, you can do other things in the main thread, such as plotting. You need to refresh the plot while the acquisition is happening, and so you need to update the plot within a loop. Combining what you've seen so far, you 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 you trigger the scan inside its thread. After starting it, you create a plot exactly in the same way as you did before. The important part is the `while` loop. First, you 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, you update the plot with the data available in the experiment class. It's 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 it's not important, and since you're not going to follow this approach in this book, you won't see it discussed further.

Exchanging information between threads is a topic that needs to be handled with care. Right now, there are two threads: the main thread, in which you define the experiment and the plot, and the 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's 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. However, data is not the only thing that is shared. The experiment communicates with the PFTL DAQ device. This communication is open to both the main and the child threads. Nothing prevents you 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 you do something like this, you 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 a voltage is read. 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. This 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.

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

All the code that hadn't changed was removed to truncate the output. It's very important to define the `self.is_running` attribute in the `__init__` because if you don't, the program crashes the first time you try to run a scan. Then, you check if the scan is already running. If it is, you 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 you 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. You can go ahead and re-run the script to see that this time you get a nice message warning you that a scan is already taking place.

Thread safety

Those experienced with threads may dislike the solution above, and they are right to do so. There's a chance that both threads will check if the scan is running precisely one after the other, and then both see that it's not. Then, both threads set the safety variable to `True`, and you face the same issues as before. This situation can happen if you trigger two threads to do the scan one right after the other. In practical terms, however, you trigger the scan by clicking on a button, or by typing in 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[a]

[a] We have written an extensive tutorial on threading on our website. Feel free to check it out to learn more.

There's only one extra feature that your `do_scan` method is missing: the ability to stop whenever you want. You've seen that the main thread can read data stored in the experiment class even if a child thread generates this data; but remember also that the child thread can see the changes to the attributes of the experiment class. Therefore, you can use an attribute, similar to the `is_running`, that signals that you want to stop the scan. You 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 you start the scan, `keep_running` is set to `True` because you want to keep running the scan. Then, in every iteration, you check whether this variable changed or not. If it's `False`, then the loop would stop. You 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. You start the scan, wait for two seconds, and then you stop it. It's not a particularly useful situation, but it's crucial for cases when you want to be in control and not lose data or 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 what you did for plotting. Then, you can use a try/except block, using the KeyboardInterrupt exception.

7.6 Improving the Experiment Class

Throughout the book, you've always come back to the *Onion Principle*. Every time you find something that you believe can be useful in the future, you shouldn't leave it as an example in a Python script, but you should instead try to implement it in a robust way. This is exactly what you did with threads. You've seen that you can run the scan without blocking the program, but to achieve it, you 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, you 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 to not have to remember which attributes do what. Let's create the methods for starting and stopping the scan, directly in 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
```

Now you 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 you the freedom to decide whether you want to run the scan on its own thread or not. There's only one more detail, and that is that if you finalize the experiment while the scan is running, then you may face some issues trying to read from a closed device. It's better to update the finalize method:

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

You stop the scan, and then you wait until you're sure it has finished. It's important because the delay between data points may be significant, or because acquiring data takes a long time, such

as what happens with long exposure times for cameras. Once you know the scan stopped, then you continue to close the communication.

7.6.1 Threading and Jupyter notebooks

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

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

7.7 Conclusions

This chapter aims at polishing some of the details that you 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. You quickly saw how to plot while the scan runs, and took a look at some of the problems that can appear when you run multiple threads. You explored some new strategies to prevent a second scan from starting, as well as for stopping the scan without the risk of losing data.

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

Chapter 8

Getting Started with Graphical User Interfaces

8.1 Introduction

Building a **Graphical User Interfaces** (or GUI) may seem more complicated than it actually is. Once you learn the fundamentals, you'll see that it's possible for you to put something together very quickly, especially if you already have some code with which to work. In the previous chapter, you saw that you could control an experiment from the command line. It's possible to ask, then, why go through the trouble of creating a user interface? The truth is, Graphical User Interfaces can be very useful for performing such tasks as controlling the parameters of an experiment, monitoring the output in a specially designed window, or even for viewing process in real-time in order to make decisions while the experiment runs.

There are several options for building GUIs with Python, and the community has no clear consensus on which path is the best one to follow. For scientific applications, however, the only library that's powerful enough to achieve what you want to achieve is called **Qt**. Qt was developed as an application framework that allows developers to build native applications (programs that look like they "belong" to the operating system on which they're run), without having to change the code. Qt itself is a Finnish company with a long history. It was part of Nokia for a while, though now they're publicly traded in the Helsinki exchange. This means that Qt is going to be around for a long time!

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

Qt, PyQt, and PySide Licensing

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

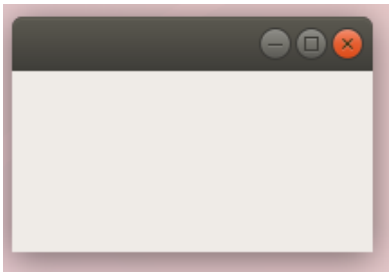
8.2 Creating a Simple Window and Buttons

Now it's finally time to start using the empty folder from the MVC design pattern: the **View**. You'll start by learning how to create simple windows directly with Qt, and then proceed to a fully-featured user interface for your experiment. You'll start with scripts and slowly grow your program as it increases in complexity.

The best way to get started with Qt is with a quick example. You can create a new file called **simple_window.py** in the *Examples* folder with this code:

```
from PyQt5.QtWidgets import QApplication, QMainWindow

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

You will come back to the code above over and over again. In the beginning, it may seem hard to remember, but once you write it often enough, it will stick. After the import, you create a `QApplication` and a `QMainWindow`. Then, you show the window and run the app. This code should produce a very simple window that looks like the image below:



The style will match the operating system of the machine your code is running on. It's a simple, empty window. However, you can already start understanding how Qt works. A user interface is a program that keeps running in a loop. When you click and drag to resize a window, for example, there's always a program responsible for knowing how to do this. In Qt, this never-ending loop is the `QApplication`. Whatever window you want to create needs to belong to an application, and that is why the first thing you did was define `app`.

In the following line, you 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, you can open dialogs and other windows, but the main window is central to the program. After creating it, you show the window. The last line is where the application loop starts. The `app.exec()` command is **blocking**. This means that nothing that comes after will be executed until you're finished with the user interface.

? Exercise

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

window before the application.

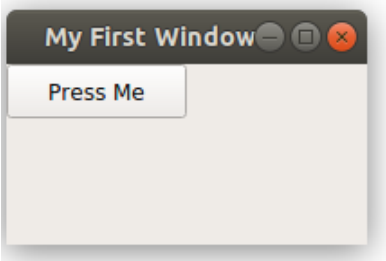
Having an empty window is not particularly useful, so you can start adding elements to it. First, add a title to the window, like this:

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

Very slowly, the program starts taking shape and looking more professional. You can also add an interactive element, such as a button. You 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()
```

This will produce a small window:



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

Exercise

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

Exercise

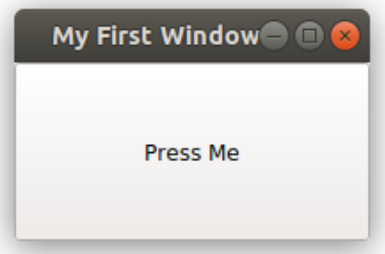
Alter the order and make the button the parent of the main window. Does this work?

A big part of working with Qt is finding out how to relate different widgets to each other, or how to position them. `QMainWindows` are special because they must hold widgets within them. That's

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). You can explicitly declare this:

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

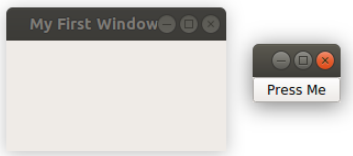
Here, you removed the `win` from the declaration of the button, but if it's there it won't change the behavior. The window now looks somewhat different:



You can try resizing the window, and you'll see that the button scales. By declaring the button as the central Widget of the window, you made the relationship even stronger. The last possibility worth mentioning before moving on is that you 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, as shown in the image below. For the program to finish, you must close both the button and the window.



Qt offers a great deal of flexibility, but you needn't take advantage of all of it. Having buttons floating around the screen does not sound like a good idea, but it's a possibility in case you ever do need it.

Now that you have a button on a window, it's time to do something with it.

8.3 Using 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 a *slot*, which does something with the information provided. When you move the mouse over a button (also known as hovering), its background changes color. This is a clear example of the signal/slot paradigm.

Every Widget that you can place on the screen has a myriad of signals, from interactions with the mouse to changes in shape or size triggered by reshaping the window, and even to time-based

functionality. This does not mean you need to know them all, or that you'll even use them all. However, once you understand the pattern, you'll know where to go to find what you're after.

This button has one signal called, very eloquently, `clicked`. To use it, you must define a function that can be called every time the signal fires. This function is the *slot*. You can expand your example code like this:

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

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

You can see that every time you click the button, the program prints a message to the screen. It's very important to note that you used `button_clicked` and not `button_clicked()`. It's the same as what was discussed in Section 7.4.1 on multithreading. You must use the function itself as a slot, and not the outcome of the function.

8.3.1 Starting a scan

With what you've done so far, triggering a scan from the user interface becomes almost trivial. For the time being, you can keep working in the *Examples* folder, but this time let's create a new file called **start_gui.py**. The only functionality you'll need is that when you press the button, a scan starts. You need to mix what you already have in **run_experiment.py** with what you've 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 you did in the previous chapter and this one. You define the experiment as always, and then you add in the code for the window and button, as you've just learned. However, the line that does all the magic is this one:

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

You can try to run the program. When you click the button, a scan starts. However, there's something else happening. The window freezes, and you're not able to reshape it or even close it. In some cases, especially on Windows, the program may even crash, and you get a message asking whether or not you want to report the issue.

> **? Exercise**
>
> Can you guess why the window freezes?

When you saw the description of the flow of a Qt program, you learned about a loop taking care of the interactions within the program. However, if you trigger a scan using `do_scan`, you're going to block that loop. Both Qt and Python are single-threaded applications by default, and when one blocks, the other blocks as well. Speaking of which, all this talk about single-threaded applications should give you a hint as to how this problem can be solved.

At the end of the last chapter, you 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. You can change just one line of code and achieve a very different behavior:

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

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

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

8.4 Extending the Main Window

You've seen how to get started by creating the main window and adding a button to it. However, notice that if you try to add more elements, the code is going to become more and more convoluted. It would be a nice addition if the window you design here could be used for different purposes as well. As you've already seen many times, a good idea when you 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 *View* folder you can create a file called **main_window.py** and add the following code:

```
from PyQt5.QtWidgets import QMainWindow


class MainWindow(QMainWindow):
    def __init__(self):
```

```
        super().__init__()
        self.setWindowTitle('My First Window')
```

Before you see a discussion of what you've done here, you 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()
```

You should also remember to change the imports at the top of the file:

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

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

Naming Conventions

It's 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 you overlook the small difference in both names. Depending on your tastes, an alternative is to call the windows by what they are supposed to do, such as `ScanWindow`. This really depends on the reader's preferences.

You 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')
```

You can test this code again and see that you've recovered what you had before. Every time you press the button, a message appears on the screen. You 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 you would like to be able to trigger a scan if you press the button. Therefore, the experiment has to come from outside of the class and be stored within.

You've already done something like this. When you developed the driver in Section 3.5, you could send the port number to the class through the `__init__` method. You did the same for the device model in Section 5.2, and for the experiment model in Section 6.2. In all those cases, you were using simple strings, but you're not limited to these. Arguments of methods, or any function for that matter, can be complex objects as well.

You can adapt the `MainWindow` to accept an experiment as an argument, store it as an attribute, and make it available for use when you need it. 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 are minimal but significant. You've included `experiment=None` in the init. You've provided a default value for the experiment because this allows you to run the program even if there's no experiment defined. This functionality is useful if you want to quickly test how the window looks. However, as soon as you press the button, the program crashes. You 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()
```

You pass `experiment` directly to the window and it takes care of the rest. You can now safely trigger a scan from the user interface. What's important to note here is that once you reach this step, all the rest happens directly on the view. The script that you use to open the window stays unaltered. Even in much more complex programs, you'll use the same pattern[1].

8.5 Adding Layouts

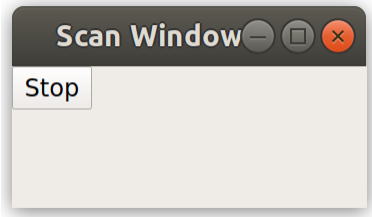
So far, the window holds only one button, and you've set that button to be the central Widget of the window. This makes it virtually impossible for you to add any other button or object. Therefore,

[1] See, for example, how PyNTA starts its user interface: https://bit.ly/WindowExperiment

it's time to start making the user interface more sophisticated[2]. The basic building blocks in Qt are `QWidgets`, and Qt allows you to place widgets inside of widgets at will. Recall that Main Windows require a central widget, so you can create a widget that holds two buttons, start and stop, where that Widget is the central Widget of the window. Use this opportunity to clean up the names you've used, and to make them more descriptive as well. It's important that you pay attention to all the changes made:

```
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 you run the program again, you will see a Window like the one below:



The stop button is visible, but the start button is not. This happens because Qt has no way of knowing where you 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 you define the buttons and see that one or the other goes on top. If the button that is below has much longer text, you'll see it beneath the top one.

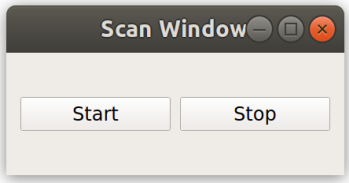
You could specify explicit coordinates for the positions of the buttons, but there's a much simpler approach, and that's by using layouts. In Qt, there are 4 basic layout types: Horizontal, Vertical,

[2]In this chapter, we decided to go for a lower level, programming every feature; but in the next chapter you will see how to do this with the QtDesigner software, which will speed up the process

Grid, and Form. With the first two, each time you add a widget, it's added either below or to the right of the previous one. With the grid layout, you can control the position, width, and height based on a grid you define. The form layout defines two columns, ideally to hold some labels and inputs. You'll see more about layouts in the following chapter. For the time being, if you want to add two buttons, you can choose a horizontal layout. The code of the `MainWindow` needs 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 you apply it to the `buttons_widget`. Then, you add the start and stop to the layout, instead of directly to the widget. This window will look much better:



If you resize it, you see that the buttons take up the entire width, and that they are always centered. It's already a good improvement compared to the simple window you started with. Before you finish this section, these two exercises are a good way of practicing the skills you've acquired so far:

> **? Exercise**
>
> Change `QHBoxLayout` to `QVBoxLayout` to see the buttons stacked vertically.

> **? Exercise**
>
> Connect the stop button to a method that stops the scan.

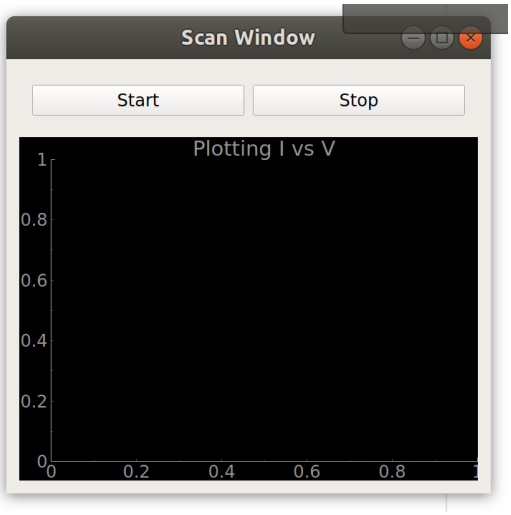
8.6 Plotting Data

You'll finish out this chapter by plotting the data in real-time. You already did something similar in Section 7.3. Parts of the code are very similar. Your window slowly starts getting more complex,

as you're adding more and more elements. So far, you have two buttons stacked horizontally, but you 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, you can make another one that contains both 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)
```

Here are some remarks about the code before you run it. You're 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, you define three widgets now: `central_widget`, `button_widgets`, and `plot_widget`. The plot widget is very similar to what you used in the previous chapter. The only difference is that you store the Widget itself and the plot separately, and you'll see an explanation as to why later. You didn't touch the buttons, but instead of adding them as the central Widget, you add them to a higher-order widget. By stacking the buttons horizontally with respect to each other, but vertically with respect to the plot, you get a window that looks like this:



You're halfway to what you wanted to accomplish! If you resize the window, the plot changes, taking up all the space available. Even if you were to expand the window vertically, there would be no gray area around the buttons. The manner in which the surrounding elements control the shape of a widget is one of the properties that you can specify.

Qt options

We make several remarks about possibilities with Qt that you won't explore further in this book. The authors simply want to point out that every single thing that happens on a user interface was decided, and can therefore be changed. Not only can features such as the aspect and colors be changed, but also the way in which different elements relate to one other and change shape when the container window changes.

To plot the data you acquire, you just need to update the plot periodically. Qt offers a special object called `QTimer` that also specifies signals. With timers, you can trigger periodic actions without interrupting the rest of the program. You 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's an object that triggers a signal, `timeout`, periodically. You connect that signal to the method `update_plot`. When you start the timer, you need to specify the time interval in milliseconds, so 50 ms means a refresh rate of 20 Hz. The `update_plot` method here is different from what you saw in the previous chapter. Instead of using

`plot`, you're using `setData`. There are two reasons for this. First, if you were to use `plot()`, then you would be creating a new plot on top of the existing one. You wouldn't be refreshing the data, but instead drawing on top of it. After a while, especially if the parameters or results change, you 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, and ticks. By using `setData`, PyQtGraph automatically reuses the elements available.

However, if you try to run the code, you'll get an error:

```
AttributeError: 'Experiment' object has no attribute 'scan_range'
```

? Exercise

Find out why this error is occurring, especially since it wasn't raised when you ran the more straightforward script in the previous chapter.

When the window starts, the timer also automatically starts, and it in turn tries to update the plot. However, the experiment class does not have a `scan_range` or any `scan_data` until the experiment starts running. A way to bypass the problem would be to start the timer after you've started the scan, but this is very unreliable. Best practices in Python indicate that you should always define attributes in classes in the `__init__` method. This means that as soon as you create the object, the attributes exist, even if that's with placeholder values.

When you developed the *Experiment class*, you completely neglected this best practice. You added `self.` whenever you needed to have data available through the class and also from outside of it. We left the definition of most of the attributes up to the reader, but here we show you how to solve the problem with the scan. Going back to the experiment model, you 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')
```

You define both attributes as NumPy arrays holding only one value: $0\,\mathrm{V}$. If you try to plot these results, you get a single point at the origin. This may raise other questions, such as whether it's better to have a 0 or a `None` value, because $0\,\mathrm{V}$ could be a valid measured value. It's left to the sensitivity of the reader to judge what is best in their specific case. For the purposes of this book, 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 check to see whether there are attributes used that have not been defined at instantiation.

8.6.1 Increasing the refresh rate and number of data points

When you follow the strategy of using a timer for refreshing the plot, you can be tempted to increase the refresh rate to make the animations more appealing, but you must be careful with this. On the one hand, if you're generating data at, say, 1 Hz, it doesn't matter how fast you refresh the plot, since it won't change faster than once per second.

But now, let's assume you're acquiring data at a much faster rate than once per second, perhaps at hundreds or even thousands of new points per second. You have to consider how fast the screen of the computer can redraw the elements on it. Most screens work at 30 Hz, and some may even go to 60 Hz. Therefore, if you try to update the plot faster than that, you'll just waste computer power on something that the screen never can show.

There's one additional limitation to take into consideration, and that is the human eye. Humans can't process images at a rate that's faster than 30fps. Already at 50 Hz, you can't see the lights in your room blinking. If you're not interested in video quality for the update of your plots, you can safely go down to 20 Hz, and the images will still look fluid.

> **? Exercise**
>
> Instead of plotting data from the device, you can update the plot with points that oscillate in time to see up to which point the refresh rate affects the quality of what you're showing.

There's one more thing to consider beyond the refresh rate, and that is the number of points you're plotting. Most screens have a few thousand pixels in each direction. A very common resolution is 1920×1440 pixels. If you acquire 10000 data points and try to show them on the screen, they'd have to be reduced almost 5 times to fit the number of pixels available on the screen. In this reduction process, one can lose many details. If you use downsampling, for example, and you're looking for a narrow peak, the chances of it appearing on the image are very slim.

You must be aware of the number of data pixels that you're trying to show, not only on user interfaces, but also when you're 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, you've started building a user interface for the experiment. You explored how to get started with Qt and PyQt, as well as how to use buttons to trigger actions using signals and slots. You also saw how to connect a basic user interface to the experiment model. This showed you the advantages of having an experiment that already runs measurements in its threads.

You also saw how to extend the basic building blocks of Qt, such as QMainWindow, by subclassing them and adding the elements you needed. Then, you worked on building widgets with more widgets inside, and laying them out in more complex patterns. Finally, you added simple plotting capabilities to the window, refreshing whatever data the experiment is acquiring in real-time.

This chapter typically generates much satisfaction for those who are developing user interfaces for the first time. On the other hand, you've done much work, and you have a window that does

not look nearly as nice as the windows with which you're familiar from other programs. On the one hand, this helps you understand just how much effort is behind every window you see! On the other, it pushes you to go one step further.

In the next chapter, you'll start to see how you can improve the design of your User Interfaces by using a program called Qt Designer.

Chapter 9

User Input and Designing

9.1 Introduction

You started building a GUI by programming every aspect of the interface. You built a `MainWindow` class and added some buttons as well as a plot. One logical next step would be to add a way to change the parameters that build up the scan. You would like to be able to change the `start`, `stop`, and `num_points` values directly from the user interface instead of from a config file that gets read only at the beginning.

You could continue adding elements to the program as you did in the previous chapter, but that would be time-consuming. In this chapter, you're going to introduce a program called **Qt Designer** that is targeted precisely at speeding up the design of user interfaces. You'll see not only how to make the program look better, but you'll also see what happens when you let users enter data, what problems may arise, and how you can keep improving based on what you've achieved by the end of the chapter.

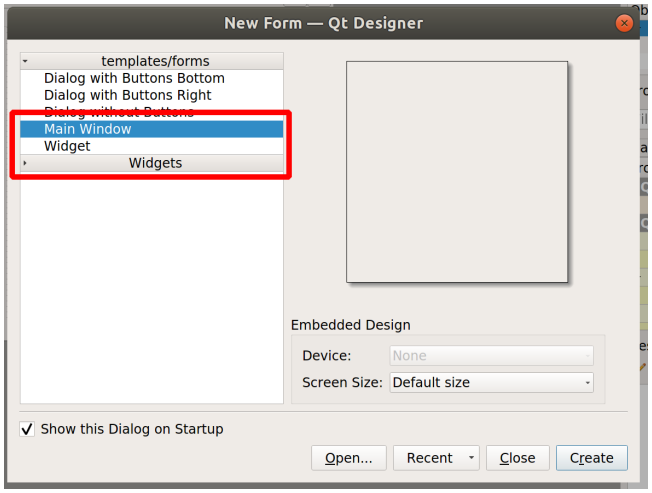
9.2 Getting Started with Qt Designer

You learned 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.

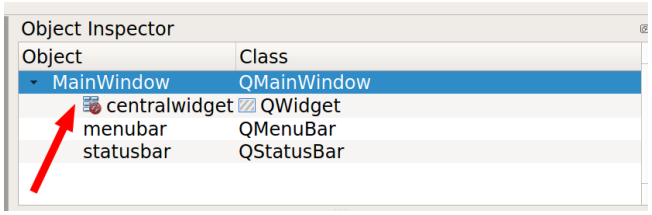
If you're using **Anaconda on Windows**, you only have to open the start menu and type *Designer*, then press enter once it's found. The process for opening **Anaconda on Linux** is similar: open a terminal, either from the base environment or the environment used for this book, then type `designer`. Press enter, and a window should open.

If you're using **plain Python on Windows** and you installed `pyqt5-tools`, you only need to start the *Command Prompt*, then activate the environment where you work, type `designer.exe`, and press enter. If you're using **plain Python on Linux**, you should have 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.

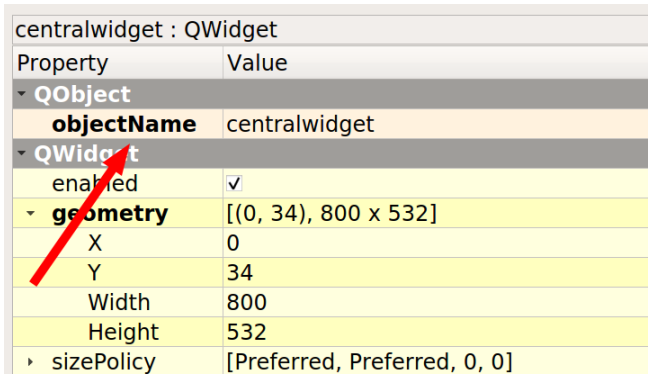
The Designer welcomes you with a screen like the one below. In between the *template/forms* options, you can already see two familiar options: Widget and Main Window.



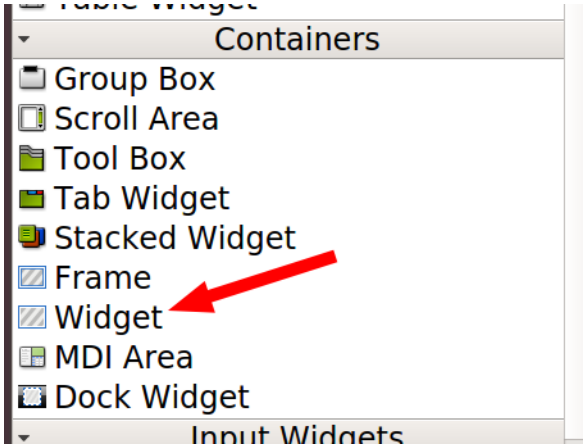
Let's recreate the window you developed in the previous chapter. You start by selecting **Main Window** and clicking *Create*. The Designer opens in the working area with an empty main window. That space is your canvas on which you'll start to add elements. In the previous chapter, you created a central widget explicitly. Here, the Designer already does this for you. You can see it on the object inspector in the right sidebar:

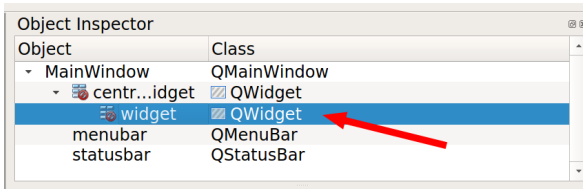


Note that the Designer named the central widget `centralWidget` instead of `central_widget`. How to name classes, variables, methods, and functions is completely up to you in Python, but some conventions make the code easier to understand at first glance. One convention is to name classes with the Camel Case convention, such as `MainWindow`, with attributes in lower-case and words separated by underscores. You can change the name of the central widget by clicking on it and changing the `objectName` name property:

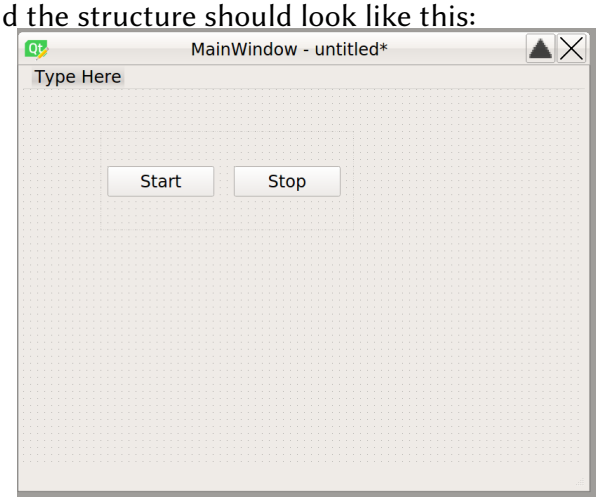


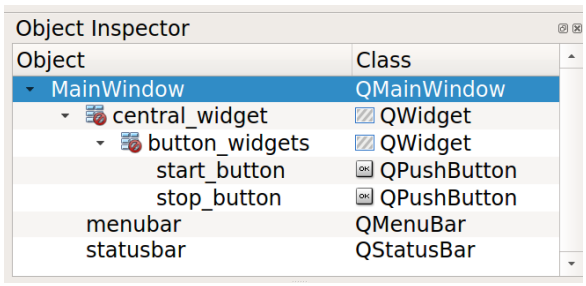
Now you have the central widget with the same name that you used in your Python class. You can add the buttons widget by dragging and dropping an empty widget to the window, which can be found in the containers group of elements. After you add the widget, you can also see it appear in the object inspector in the right sidebar. You can change its name to `button_widgets`.





To add the start and stop buttons, you can just drag and drop two `QPushButton` elements to the window. You need to be sure that you drop them inside the buttons widget you created earlier. To change the text that appears on the button, you can double click on it and edit the text. You 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. You must change the name of the buttons, and to repeat the same structure of the previous chapter, you're going to call them `start_button` and `stop_button`. The window and the structure should look like this:





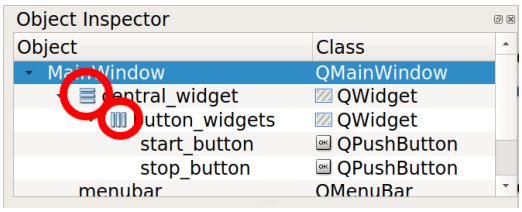
Screenshots

Reading an explanation of how to use a designing program through screen captures requires much patience from the reader. We try to highlight the checkpoints that allow you to compare your own output to what you see here. Practice and critical viewing is the best thing that can the reader can do at this stage.

You now have a window with the buttons, but you're missing the layout to make it look as good, as you did in the previous chapter. Luckily, the Designer has a tool bar 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, you can first click on the widget inside the Object Inspector and then click on the desired layout. You can apply a vertical layout to the `central_widget` and

a horizontal layout to the `button_widgets`. When you apply a layout to widgets, it shows as a different icon on the Object Inspector.



You're ready to save the window. Let's create a new folder called *GUI* inside the *View* folder, and you can call the file `main_window.ui`. Once you save the designer file, you're ready to go back to Python to use this window. Go back to **main_window.py** and start editing the `MainWindow` class. Since some of the elements are now defined directly in the designer file, you can 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 you load the file using `uic.loadUi`. When you try to import files in Python, you always have to be careful with determining the path from which you're importing. You already encountered the syntax of `dirname` when you were adding the root folder to the path in Section 5.6. The idea is the same here, but now you're interested in the folder where the current Python file is located.

The command `uic.loadUi` takes two arguments: the first is the file you want to load, and the second is the object you want to apply it to. You use `self` to indicate that you want to apply the layout to the entire `MainWindow`. If you develop a more modular program, such as defining individual widgets, you could apply the Designer file just to the widget that you're interested in.

The rest of the window stays the same. You only remove the definition of `central_widget` and the buttons. Since the layout of the central widget is specified in the Designer, you need to access it to be able to append the plot widget. That's what this line is doing:

```
layout = self.central_widget.layout()
```

You can go ahead and run the program, and you'll see that the window appears as it did before, without changes, even though you're now pulling the elements from the Designer file.

9.2.1 Deciding whether or not to compile UI files

Most online tutorials add one step after creating the Designer file. They normally suggest transforming the `.ui` files into a `.py` file. There's a program that can do this for you, which can be triggered from the command line:

```
pyuic5 main_window.ui -o compiled_window.py
```

You can explore both files and see the differences. If you open the *ui* file with a text editor, you'll find that it's in plain text and formatted following a standard called *XML*, or eXtended Markup Language. This format is very similar to how websites are built with *HTML*, which stands for HyperText Markup Language. Going through the file, you'll find the place where you defined the buttons, surrounded by some more information, such as the layout and the text of the button.

If you open the generated Python file, you'll find the same information but formatted in the manner you did previously. Go to the declaration of the start button, and you find a line almost identical to the one you used in the previous chapter:

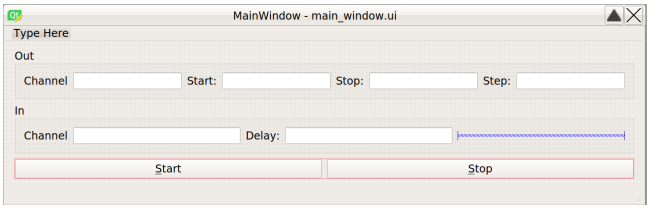
```
self.start_button = QtWidgets.QPushButton(self.button_widget)
```

The question is whether you *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's essential to have each variable defined with a specific type before compiling the code. Qt offers a program called `uic` to transform *ui* files into *C++* compatible files. Most Python tutorials have built on that experience and kept the recommendation; but for Python, this step is not necessary at all.

There's one caveat, however. If you don't transform the file to a Python file, then text 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's a `start_button` or not defined. This means that, to be sure, you'll need to keep opening the designer program to see how you named the buttons and elements. Still, this is the only drawback for choosing not to compile.

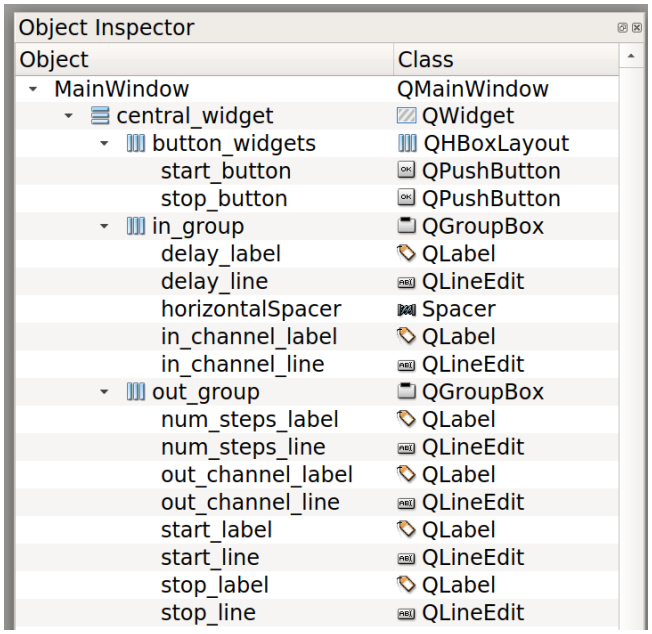
In our experience, modifying the designer files and seeing the changes as soon as you restart the program outweighs the problems of not being able to auto-complete. If you're systematic with the naming conventions, you won't face many issues. If you do encounter issues with attributes not defined, you'll know that the root of the problem is that you used one name in the Designer and a different one in Python.

9.3 Adding User Input



It's time to take your program to the next level by adding the possibility of entering the scan values before triggering it. The goal is to have a window that looks like the image above. The window is structured in three rows: one for selecting the range of the scan and the channel, one for the input and delay between points, and one for the buttons. Below this, you'll append the plot, as you've done before.

First, you'll start by designing the window. You won't see covered every single detail as you did in the previous section, because it would become too cumbersome for you to follow. Therefore, it's left up to the discretion of the reader whether or not to try this by themselves or to get the designer file from the online repository. The easiest way to reproduce the window is by looking at the structure you'd want to obtain after putting all the elements in place:



The central widget and the buttons are there, as always, but there are two more widgets added now. These are of a particular type called `QGroupBox`, meant for grouping elements together. They also have a title that appears at the top left, which becomes quite handy for understanding what you need to edit before starting a scan. To handle input from the user, you use `QLineEdit` widgets. These widgets allow the user to input any type of text, which is precisely what you need.

The rest of the elements are to give consistency to the user experience. Before each line edit, you include a label to show what information is supposed to go in each box. In the bottom group, you also used a `Spacer` to push the elements to the left. If you don't do this, the boxes for editing will take up the entire space and become visually unappealing. It is, however, a purely aesthetic decision.

The most important things for you to pay attention to are the names of each input line, because they're going to be fundamental for the Python code. What you'd normally use is a distinctive name followed by the type of element you're dealing with. That is why you have the `start_button`, but also the `stop_line`, `delay_label`, and so on. In this way, you 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 you were to swap the in and out channels, the error might go unnoticed until you used two different values. By that point, it would be tough to understand if the problem exists within the program or within the experiment.

After improving the window design, you can run the program again and you will see that the window has a different aspect, with inputs, labels, and groups. However, they can't do much yet. If you 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 that are started when you load the config file. You 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 you to change what's 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. You need to force it to do so by adding the `str` function.

Exercise

You've seen how to add the values of start, stop, and the number of steps, but you're still missing the input and output channels, as well as the delay between points. Add them by following the example above.

You know how to set the values to the lines, and you can even change them, but you're still using the ones defined in the config file. You already saw in Section 7.2 that, thanks to your strategy of having a robust experiment class, you can change the values of the parameters after the experiment is defined. You 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 in which you used `setText` to set the text on the `QLineEdit` objects, you can use `text()` to retrieve what is written. For the start and stop there are no problems, but for the number of steps you need to have an integer, not a string. Once you have the values, you update the `experiment.config`, specifically the group of properties that belong to `Scan`. Using `update` on a dictionary is a shorter way of doing this:

```
self.experiment.config['Scan']['start'] = start
self.experiment.config['Scan']['stop'] = stop
self.experiment.config['Scan']['num_steps'] = num_steps
```

It's important for you to remember that in the experiment, when you trigger the `start_scan`, the model takes care of creating the scan range directly from the values stored in the config dictionary. If you supply values without units, the experiment complains.

> **? Exercise**
>
> Test the limits of the user interface. What happens if you use a value for the 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.

You're now at a stage where you can control your experiment from the user interface. You can change the parameters for the scan, as well as start and stop it at will. At this stage, you should feel very proud of yourself! It's also an excellent time for you to reflect, because things can change very quickly. New elements may appear on the window, but you've put minimal effort into making everything work together.

One of the benefits of separating the Model, View, and Controller is that, usually, most of your work should go in the Model. The good news is that Models are the most pure-Python modules in your program. Once you overcome the initial shock of working with more complex patterns, such as classes and threads, the rest is very straightforward. You loop through values; you save data. Most likely, it's the kind of thing you 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. Figuring out how to design the experiment model in such a way that it allows you such a degree of independence, on the other hand, is a skill that you can acquire over time, with critical thinking and patience. For most programs, when developing the *View*, you would face the problem of something being missing in the experiment model (perhaps a threshold value, or a variable that lets you know something is running). You should always refrain from the temptation of complicating the View, and you should always favor putting all that effort into the Model.

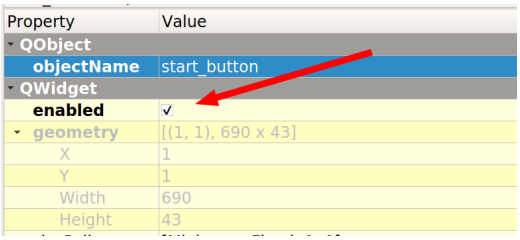
In the long run, it's your future experience that will congratulate you for a job well done in the past.

9.4 Validating User Input

Consumers of computer software are used to a lot of interactions and visual cues when working with a user interface. It makes users expect the same kind of behavior in the programs you develop yourself. One of the things you can notice in your program is that, if you try to trigger a scan while the first one is running, the program prints a message to the terminal; but if you're not paying attention to it, you can miss it and wonder why the scan isn't 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 you achieve that behavior? If you look in the Designer, every widget that you add has a list of properties that you can change, including shape, color, text, and so on. In the case of the buttons, there's an option called `enabled`:

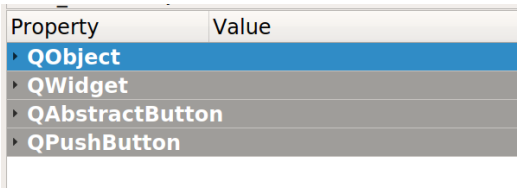


If you remove the tick mark from the option, the button gets grayed out, and you won't be able to click on it. You must learn how to change the enabled status of your program, but not from the Designer. For this, you must read the documentation provided by Qt. If you search online for `QPushButton`, usually the first result would be the one available at `doc.qt.io`, the official documentation. You only need to be sure you're looking at the Qt5 documentation and not the one for Qt4. The page is long, and if you look for `enabled`, you won't find anything. It's not a very auspicious start.

If you scroll to the top of the page, however, you'll see that Qt informs you of the parent class of QPushButton: QAbstractButton. Remember that when you're working with objects, there's always a possibility that methods and attributes are defined in the parent class, and not in the class you're using. You can follow the link, but there won't be any information on how to enable or disable buttons. However, if you go one level above, to the `QWidget` documentation, you find what you were looking for:

Header:	#include <QPushButton>
qmake:	QT += widgets
Inherits:	QAbstractButton
Inherited By:	QCommandLinkButton

To change the status of the button, you must use the `setEnabled()` method that's defined in the base `QWidget` class. On the one hand, notice that all widgets can be enabled or disabled. In some cases, the change is visible; in others, it won't be. You can also see that the Designer was already letting you know that the documentation was available as part of QWidget instead of QPushButton. Just look at how it organizes the properties:



Next time you want to learn how to do something, you'll know your best bet is to start with the Designer and see if the options are available. If you find one, you must pay attention to the base class that defines the behavior, and then you can go looking for the documentation. Navigating the Qt Docs takes a bit of getting used to, but once you understand how the information is organized, it becomes straightforward to follow it.

Back to the task at hand! You must disable the button when the scan starts running. One option would be to disable it right when you trigger the scan:

```
def start_scan(self):
    self.start_button.setEnabled(False)
    [...]
```

It seems like a very valid idea until you use it for the first time. Once the scan starts, the button is grayed out, but there's no place where the button can be enabled again. This is a typical pattern with user interfaces. There are different ways to tackle the problem, but the more straightforward one is to use the timer you already have in place to update the plot.

Instead of just plotting data, you can use the same timer to update the different elements of the user interface. For example, you would like to let the user start the scan only if there's no scan running, and you want to let them stop the scan only if a scan is 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, you gray out one button, but not the other, and vice-versa. You only need to be sure this method runs periodically. Since you already have a timer for updating the plot, you 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, you check every few milliseconds whether the scan is running or not, and you set the buttons accordingly. There's also a risk with this approach, however. If you decide to show information on the Main Window that implies reading a value from the device, you end up polling it several times per second, regardless of whether the value changed or not. If you update values stored in an object, such as the Experiment model, there's nothing to lose, but if you stretch this pattern too far, you may encounter issues, and your program could become inefficient.

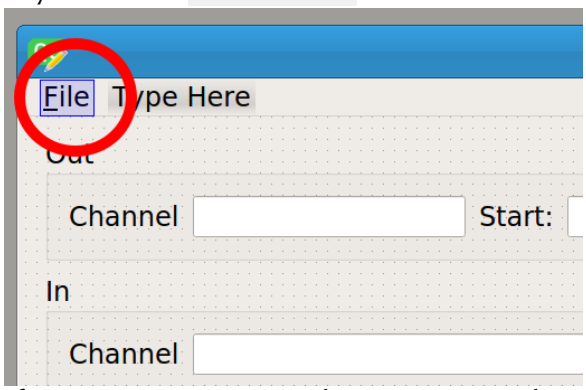
You've generated the most basic of the possibilities, disabling a button to prevent the user from triggering a second scan if there's one already running. There are many more possibilities. What happens if the user enters a value outside the range? For example, what if they enter a start value without volts, or a negative value, or even a value beyond $3.3\,\mathrm{V}$? You can use the `update_gui` to monitor the values entered and run them through some checks. If they pass, then you leave them as they are. If they don't, then you change the line to red and disable the start button.

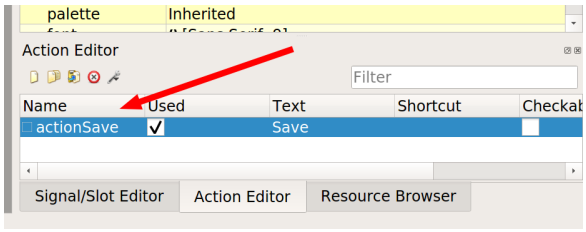
All these are possibilities that you'll need to consider as soon as you start developing user interfaces. However, everything you do has ramifications, such as what you saw when you disabled the start button on your first attempt. One general approach is to not complicate the user interface beyond what's needed. If you have proper drivers and proper models in place, at least your devices are safe. If you find yourself or a user of your software making the same mistakes over and over again, then you can optimize the flow to prevent it from happening again.

9.4.1 Saving data with a shortcut

The only important missing feature of your user interface is the ability to save data. You could implement a button for this, but you can also look into another component of Qt: **actions**. You saw that one of the conveniences of 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, you can add a `File` menu by double-clicking the top bar of the main window, where it says `Type Here`. Inside the File menu, you can then create a `Save` option. As soon as you create the Save option, the Designer will also show an *Action* defined. You can find the action listed at the bottom of the right sidebar, as shown in the image below. Note that the name assigned to it by default is `actionSave`.





If you pay attention to what appears on that square, you'll see that there's also the possibility of defining a shortcut. Double-clicking on it opens a small configuration window in which you can add a shortcut, such as Ctrl+S. Just clicking on the shortcut edit line and then pressing a combination of keys will register them. After saving, you can rerun the program, and you'll have a File menu with a Save option, as well as the shortcut written next to it, so there are no possible confusions.

You need to connect the `actionSave` to the method in your experiment for saving data. Because of how you structured the code, this now becomes almost trivial. In the `__init__` of the `MainWindow` you just add the following:

```
self.actionSave.triggered.connect(self.experiment.save_data)
```

You 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, instead of first defining a method on the window to then trigger the model. If you rerun the program, you 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'll see the data appearing there.

Shortcuts

Adding shortcuts for actions can be a good idea, but it also has its risks. Shortcuts are hard to document, and some shortcuts are so universal that it's better not to re-define them. You could have used Ctrl+S to Start the scan, but then, you'd need to stop it and choose Ctrl+X, because it's right below. To save, you need to choose another shortcut like Ctrl+Q because it's on top of the S. All these shortcuts are generally used for something else in many other programs, like saving, cutting, and closing a window. You're free to use them as you wish,

but over-use can be a perilous path.

Of course, you may be tempted to open a pop-up dialog to ask for the folder where the data should be saved to; but again, this has a complex set of possibilities. If you open a pop-up every time you want to save data, it becomes annoying, especially if you save data often. If you show it only the first time, then you won't be able to change the location later on, prompting one to wonder why you showed it in the first place. This means that you need to establish two behaviors: *save* and *save as*. You won't see how to do it here, but it's left to you 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 in which to save the data.

9.5 Conclusions

This chapter is one of great joy! It's the conclusion of a long journey, from developing a driver and creating models, to abstracting the behavior you expect of both the device and of the measurement you want to do, and finally to developing a user interface that works and looks beautiful.

This chapter focused on showing you how to use the Qt Designer, a useful tool for speeding up the development of user interfaces. You 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 you to build 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.

You've merely scratched the surface of what Qt offers. For scientific applications, however, you won't need to go much deeper. You'll generally be okay with just the simple styling of objects and an uncomplicated menu structure. You could change font sizes, colors, and even the roundness of edges. You could make floating menus that open by right-clicking on an image, and many other things as well. However, one always has to draw a line between how much effort and time you're dedicating to making a program look better, and how much time you're dedicating to running an experiment.

9.6 Where to Next

If you've reached this point, it means that you've kick-started your career as a Scientific Python Developer. Hurray! What you've covered in this book is only the beginning. Devices can be much more sophisticated than a simple DAQ. You may need to analyze the data before you display it, or the memory on your computer may not be enough to hold one complete measurement.

Our advice that we give during the workshops is that you should work on what you've 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, you've been guided and prevented from falling in some ways. 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've built a forum[1] where you can ask any question related to programming, working in the lab, or writing Python. If you have any comments, suggestions, compliments, or critiques to send us, you can do so directly to the author at aquiles@pythonforthelab.com. I always want to read how your path is going and what topics are making you struggle.

If you're interested in a follow-up to this book, one covering more advanced topics, please drop me a line! I am always seeking new ideas, topics, and challenges, not only to produce content for reader, but also to learn myself. I do a periodic brain dump of what I learn on the website pythonforthelab.com. If anything grabs your attention, you can always leave a comment to let us know.

[1] available at: https://forum.pythonforthelab.com

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 derives 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 to change 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 $0 - 3.3\,\mathrm{V}$. Digital-to-Analog conversions are done with a resolution of 12 bits in the range $0 - 3.3\,\mathrm{V}$.

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 on-board 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 Available Commands

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 for the Lab** is to bring a developer from knowing the basics of programming to being 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 get started with 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. This should start the Python interpreter, and you should see something like this:

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

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

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

The first line is what you type (therefore, it has the prompt symbol `<<<` 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, and more. Powers of numbers can be achieved using `**`; for example, `2**.5` means the square root of 2.

B.3 Lists

From now on, the prompt symbol `<<<` 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 then you print them to screen. You should see 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. You 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 here, 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 at 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'>
```

In other words, you can have lists with lists in them, and many other combinations as well.

> **? 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 concise syntax:

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

This 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)]
```

You can even make it more complex. For example, if you want to get only the even numbers you can type this:

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

> **? Exercise**
>
> Given a list `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 for you to keep them in mind to be able to follow the syntax of some programs without getting lost.

B.4 Dictionaries

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

```
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 in a dictionary, 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 it 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, then you can do the following:

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

Here you can 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 as with lists, it is possible to access every element with a for loop:

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

Now the key has a value that can be printed and used. You 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 also that 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 one 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 you have only 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}
```

On the one hand, the syntax above 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 an 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 needed 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 list `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 double.

? 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 paradigm that allows developers to define not only the type of data that a variable stores, but also the operations that can act on that data. For example, a variable can be of type integer, float, or string. You can multiply an integer to another integer, or divide one float by another, but you cannot add an integer to a string. Objects allow programmers to define operations between different objects as well as between an object and itself. For example, you can define an object `person`, add a birthday, and have a function that returns the person's age.

In the beginning, it may not be clear why objects are useful, but over time it becomes impossible not to think about code with objects in mind. Python takes the idea of objects one step further and considers *every* variable an object. Even if you didn't realize it, you might have already encountered some of these ideas when working with NumPy arrays, for example. In this chapter, you are going to cover the very basics of object-oriented design, before proceeding to slightly more advanced topics in which you 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 Python. Defining a class is as simple as this:

```
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 you use classes when referring to the type of variable, and object to refer to the variable itself. This is going to become more apparent later on.

In the example above, you've defined a class called `Person` that doesn't do anything (that is why it says `pass`.) You 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, you have to create a variable of type `person`. Back in the Python Interactive Console, you can, for example, do this:

```
>>> 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`. One important detail has been omitted: 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 about it, but bear in mind that when you define a new method, you should always include the `self`; 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, like this for example:

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

This can be used like so:

```
>>> 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 in the fact that they have access to *all* the information stored within the object itself. The `self` argument that you are passing as the first argument of the function is exactly that. For example, you can add the following two methods to your `Person` class:

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

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

Then, you 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 in 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 you see in the last line is that you can access the property directly without needing to call the `get_name` method. In the same way, you don't need to use the `store_name` method:

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

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

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

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

C.2 Initializing Classes

Instantiating a class is the moment in which you 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 you 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, as 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 a common (and smart) practice to declare all the variables of your class at the beginning, inside your `__init__` method. In this way, you won'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, so 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()`, it 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 like so:

```
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 `''`, that is, 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 so is as simple as declaring them before the `__init__` method. For example, you could do the following:

```
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, you 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 it; 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 aside, once you change the class property, like you did in the example above the birthday, next time you 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'll need it, and it will save you plenty of time.

C.4 Understanding 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 thing would be to see it in action. So far, you have a class called `Person`, which is generally not too useful. Let's assume you want to define a new class, called `Teacher`, that has the same properties as a `Person` (that is, a name and a birthday), in addition to having the ability to teach a class. You can add the following code to the file **person.py**:

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

    def get_course(self):
        return self.course

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

Note that in the definition of the new `Teacher` class, you have already added `Person`. In Python jargon, this means that the class `Teacher` is a child of the class `Person` (or vice versa, that `Person` is the parent of `Teacher`). This is called **inheritance** and is not only very common in Python programs, it is also 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 (that is, the base class) by adding the following:

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

When you use `super()`, you are going to have direct access 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 to run the method `me.get_name()` again, you'll 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 Looking into the 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 inheritance would be useful. Without a doubt, it helps you understand the code developed by others. There are, however, a few finer 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 note that the method `__str__` is used when you want to transform an object into a string, for example:

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

This 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'd 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`, you 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
```

This can be used like so:

```
>>> 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 the assignment operator `=` does 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, you 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 this can be convenient. It takes a bit more time to develop than with simple methods, but it simplifies the rest of the programs that build upon the class quite a lot.