Python in a Physics Lab

Gergely Imreh
Institute of Atomic and Molecular Sciences
Academia Sinica, Taiwan
gergely@imreh.net

Abstract

Physics laboratories rely on computing in all aspects of their research, and usually mix and match many different software tools. Python can serve well in all stages of the scientific work, and provide benefits for professors and students in the long term.

Keywords*: scientific computing, academia.*

1. Introduction

The software environment in physics research laboratories is usually best described as chaotic. This can be attributed to the unique environment of academia, which on the negative side combines lack of time, resources, training, and interest with the need of high quality results.

The emphasis is on quickly moving research where, often, the slogan is that “an experiment should fall apart right after the last data point was recorded; if it doesn't then it was over-engineered.” Software developed in such an environment rarely moves out of an alpha-release stage, when “it works, barely.”

Most researchers are not programmers but self-educated in those aspects of programming which are directly relevant to their work. Software development best practices such as using version control software, and documenting code, are often neglected, or made harder by the choice of systems to use.

Practising scientists are still a small community, and while they share experiences and advice, the pool of knowledge is limited. This leads to the fact that many people decide to simply use the same software used by the supervisor of their masters, Ph.D., or post-doctoral positions, whether it was old or new, free or expensive, good or bad. These decisions are then passed on to their own students as well.

I propose that Python is a great tool to improve on all these issues, enabling efficient and knowledgeable laboratories.

2. Experiment X

I demonstrate this by walking through an example of Experiment X, using Python at all the steps along the way, highlighting the different tools that can be used. Most of the tools I described have been used successfully in our physics laboratory.

Experiments, in general, have the following 4 aspects and stages:

- Planning and theoretical predictions
- Instrument control
- User interface for control
- Data analysis and archiving

2.1. Planning and theory

In a grossly simplified form, to plan a physics experiment, you develop a theoretical background; estimate the signals recorded, and adjust the equipment requirements if the numbers are unsatisfactory; then, formulate an idea on how to extract the results from the recorded signals.

For theoretical physics calculations, the libraries with the highest profile are NumPy (http://www.numpy.org/), SciPy (http://www.scipy.org/), and SymPy (http://sympy.org/). NumPy enables a wide range of numerical computations with matrices, and provides interface to C and Fortran code, a lot of which exists in the scientific community. SciPy offers many user friendly and tested functions for scientific computing, from numerical integration, through optimization, to statistical functions. SymPy provides tools for symbolic mathematics - very useful to work with physics equations.

All three libraries above have abundant documentation and examples to get started on any particular calculation. These libraries are sufficient for the majority of mathematical problems that physicists encounter, while there are more specialized libraries for different areas of physics. The rationale for many specialization is to provide functions for complex calculations that would be error-prone if everyone re-implemented it themselves, or to reduce the amount of boilerplate code that one sometimes need to resort to when bending a general tool such as NumPy to specific needs. An example of such a library in my own area of atomic physics is QuTip2 (https://code.google.com/p/qutip/), The Quantum Toolbox for Python. The Python wiki lists more numerical and scientific libraries as well (https://wiki.python.org/moin/NumericAndScientific).

In theoretical physics calculations, a common situation is to run the same mathematical functions a number of times, with slightly altered parameters, e.g. optimizing the physical dimensions of an apparatus, or modelling particle trajectories. To use the available computing power such as multiple CPU cores more efficiently, these calculations should be parallelized whenever possible. Python's own multiprocessing library makes it much simpler to write parallel code, compared to most other programming languages.

Preparation is not just calculation, however. Engineering, and modelling physical arrangement of the experimental equipment is often helped by Computer Aided Design (CAD) software: FreeCAD (http://www.freecadweb.org/) is a parametric design program

written in Python, that can be scripted in Python as well, enabling easier and more interesting designs. The Python Power Electronics library (http://sourceforge.net/projects/pythonpowerelec/) enables modelling electronics circuits, another area where a lot of work is done in a physics laboratory.

Python scripting makes possible connecting tools from different steps of the research, e.g. visualising simulations and experimental results within the 3D modelled experimental apparatus, which can improve understanding. I hope more software will be available in the future offering Python scripting and interoperability.

2.2. Instrument control

The next step is to make the instruments do the required experimental sequence, such as setting device parameters, output signals, record results, all with appropriate timing. This is generally made difficult by the variety of instruments and their methods of communications. Fortunately, Python provides tools to talk to almost any instrument that can communicate.

RS-232, or often called "serial" connection, is one the most common communication standard that I have found around the laboratories I've worked in. PySerial (http://pyserial.sourceforge.net/) enables simple communication over the serial port, in a pattern similar to this:

```
import serial
instrument = serial.Serial("/dev/ttyUSB0", baudrate=19200, timeout=1)
instrument.write('command')
```

Due to the longevity of the serial standard, there are a lot of variations between devices, fortunately PySerial can adjust almost any parameter of the protocol, and newer devices are increasingly using just a small subset of configurations.

One of the most accessible hardware that can be communicated with over Serial port and enables a lot of laboratory automation is Arduino (http://www.arduino.cc/). Its ubiquity, and codes that are all custom made, necessitates good knowledge of the workings of RS-232.

The second most common way to talk to instruments is via the General Purpose Interface Bus (GPIB, IEEE-488). It was developed in the 1960s, and it's still very popular with high end instruments, especially due to the fact that they can be daisy-chained. Many companies producing GPIB enabled instruments rely on National Instruments' Virtual Instrument Software Architecture (VISA) for control. In Python, the PyVISA library makes GPIB communication possible (as well as providing RS-232 and USB access). It requires the National Instruments libraries to work, which in my experience in practice limits this library to the Windows platform, though it was designed to work on both Windows and Linux.

GPIB instruments have their GPIB address. They respond to specific messages, some of them are standardized in practice, some of them are instrument or company specific. The following code segment for example asks for the instrument ID string, which is usually the model number, from the device on GPIB address 12, and prints the answer:

```
import visa
device = visa.instrument("GPIB::12")
device.write("*IDN?")
print(device.read())
```

For high speed communication some devices are using FireWire IEEE-1394. It is especially common among cameras. Pydc1394 (https://launchpad.net/pydc1394) wraps the libdc1394 library under Mac and Linux to communicate with these cameras. This arrangement enables for example the common task of real-time image processing, leveraging NumPy's numerical calculation power. An example of taking an image with pydc1394 is the following:

```
import pydc1394, numpy
lib = pydc1394.DC1394Library()
cams = l.enumerate_cameras()
cam0 = fw.Camera(l, cams[0]['guid'], isospeed=800)
image = numpy.array(cam0.current_image, dtype='f')
```

Pydc1394 is one of the more scarcely documented libraries because it's less used than the ones mentioned so far, thus requiring more background knowledge and experimentation.

The USB Test and Measurement Class (USB-TMC) can be found in some of the newer instruments, especially oscilloscopes. Linux has a built-in driver for USB-TMC, thus communication with a device is as easy as writing to and reading from a simple file handle.

When devices require special drivers for communication, vendors often resort to providing only LabView enabled drivers, or pre-compiled DLLs. In this case, Python's foreign function library, ctypes comes to the rescue. Using ctypes to directly call the functions provided by the DLLs usually means more boilerplate and somewhat un-Pythonic code. It needs to accommodate pre-defined types for function arguments and return values, but the implementation is relatively straightforward and readable. An example of such interface code is:

```
import ctypes
my_dll = ctypes.windll.dll_name
receive_data = my_dll.ReceiveData
receive_data.restype = ctypes.c_long
print receive_data()
```

To reduce repetition and common errors, I recommend creating libraries for the specific instruments so that their interface code can be reused from experiment to experiment. It is especially important as a single experiment's control software should be immutable: archived after finishing the work together with the recorded data, and forking it for the next experiment. One can also automate common tasks by creating scripts in Python, such as saving data from an often used instrument over GPIB, serial, or other available interface, instead of using floppy disks or USB pen drives to trasfer reading values.

There are a number of devices that take interfacing with hardware to another level by enabling control directly by Python: the Python Controlled Microcontroller (PyMCU)(http://www.pymcu.com), and Micro Python (http://micropython.org/). Using such

devices abstracts away the hardware requirements for communication (which is on over serial connection), and makes controlling the input-output pins straightforward in Python. For example, with a PyMCU "blinking" one of the output pins for 500ms becomes:

```
import pymcu
board = pymcu.mcuModule()
board.pinHigh(1)
board.pausems(500)
board.pinLow(1)
```

More examples of controlling hardware can be found in my Github repository containing laboratory instrumentation code (https://github.com/imrehg/labhardware).

2.3. User interface

Once we know what should the experiment do, and how we will talk to the required instruments, a communication method has to be developed for the intended users as well, i.e., a user interface should be created.

For small scripts, a console interface with text input is quick to develop, and this seems to be adequate in practice. This was usually my first step with any newly developed controller program.

Better usability can be achieved using a graphical interface. I had good results with an arrangement borrowed from web development. The base of the code provides an API for the required tasks, and control is done via a web interface. The Bottle web framework (http://bottlepy.org/) is a very quick way to get such a setup working. This has the benefit of making the control software accessible over the local network or the Internet as well, which can benefit control and monitoring projects.

A more involved approach is using PyGTK (http://www.pygtk.org/) or wxPython (http://www.wxpython.org/) to create a native interface for the control software. One advantage is that the user interface can be decoupled from the control logic using graphical user interface design tools like Glade (https://glade.gnome.org/) for GTK, or wxGlade (http://wxglade.sourceforge.net/) for wxPython. Such combination can form a powerful basis to create stand-alone instrument control application, especially when a control software is developed for the long term.

PyGame (http://www.pygame.org/) is another alternative, as many control programs are graphics-heavy and interactive: real-time display of experimental results, showing graphs of instrument control sequences, and displaying the status of different devices.

Making a good interface requires quite a bit of knowledge in different libraries, beyond what is immediately required for the experiment, and better development practices, making it more of an advanced area of Python in laboratories.

2.4. Analysis and archiving

When the experimental results are collected, it is time to analyse them and draw conclusions. Many of the tools used in the preparation and theory section become useful here, such as NumPy and SciPy. They provide good numerical, statistical, and fitting functions.

Besides NumPy's ability to interface with data stored in "comma-separated values" (CSV) files, PyTables (http://www.pytables.org/) makes importing and exporting data much easier, not just from simple tables, but other large data formats such as HDF5 which is used for long-term data storage.

A large part of the analysis is visualizing the results, and the most well-known tool for that is Matplotlib (also called Pylab) (http://matplotlib.org/), which can create almost arbitrary graphs, plots and graphics, exporting them to many different image and document formats. It has a daunting number of options, and it is very well worth checking out the examples given in their gallery, together with their source code. However, in its most simple form, plotting results is quite straightforward:

```
import pylab, numpy
readings = numpy.loadtxt('data.csv')
pylab.plot(readings[:, 0], readings[:, 1])
pylab.show()
```

CyNote is an electronic laboratory notebook software, developed for Biologists and Bioinformatics, but can be used for physics just as well (Ng and Ling, 2010).

MoinMoin (http://moinmo.in/) is a wiki written in Python that makes knowledge sharing and collaboration in a team very easy, storing information from lists of equipment in the lab to operating procedures, from theoretical ideas to standard readings of some instruments.

3. Competition

Python has a lot of competition in the physics laboratory, especially large and expensive closed-source software.

Matlab (http://www.mathworks.com/products/matlab/) is widely used by scientists in many disciplines, and it mostly competes with Numpy, SciPy, Matplotlib. Matlab has a large number of add-ons for different tasks, but it is still less cohesive than Python which is a full-fledged programming language.

Mathematica (http://www.wolfram.com/mathematica/) is a similar problem space of numerical and symbolic computing. For the latter, it currently provides more functionality than SymPy. It stores programs in binary files and requires license to use, thus making calculation results less accessible in practice.

LabView (http://www.ni.com/labview) is a dominant player in instrument control. It is using a graphic programming environment which makes many software development tasks, for example, source control or debugging, difficult or impossible. Developing simple interfaces is very easy, but large scale programming gets extremely complicated and hard to debug

quickly. The LabView runtime is very resource-hungry, while using Python, even a Raspberry Pi can be sufficient to control the same instruments.

IGOR Pro (http://www.wavemetrics.com/products/igorpro/igorpro.htm) is a data analysis tool often used in atomic physics laboratories. Most of the functionality can be replaced with NumPy and SciPy, though it has to be hand coded, while IGOR provides better user interface for common tasks.

A different kind of competitor is Node.js (http://nodejs.org/), which enables developing control software similar to Python+Bottle, using Javascript. Much fewer libraries are available, but some of the existing ones are promising (such as event-driven serial adapter https://npmjs.org/package/serialport/). Seeing the development of Node.js will hopefully contribute to the improvement of Python libraries, and enable cross-pollination between those projects.

In numerical computing, there are still a lot of legacy C/C++ and Fortran code since they are well tested and often have a higher performance than Python. Moving on from legacy code will likely only happen with the new generation of scientists using different languages, including Python, for their work.

An up and coming addition to numerical computation languages is Julia (http://julialang.org/), which combines high-level, high-performance computation and dynamic programming, and is often compared to Python (Bezanson and Karpinski, 2013).

4. Discussion

As a physicist in an academic environment, I value education highly, and I believe that the expensive commercial software that is used for many tasks around the laboratory is a non-starter, because it is practically a barrier to learning and scientific development. Students graduating from a laboratory can no longer afford to buy the software they used for their studies. Using Python on the other hand gives them a transferable skill that they can continue to use and improve. Theoretical calculations, experimental control programs, and data analysis of a laboratory can all be verified by independent parties as they have free and open access to the tools required, which is a fundamental requirement for responsible and high quality science.

Another advantage that makes Python programming a great transferable skill is that Python is a full-fledged programming language. Unlike competitors such as Matlab or LabView, Python can be used for any programming task, for example even if someone starts out with numerical programming, a module can quickly be added to the code to receive the results of a long computation over email. The skills learned in the laboratory enables people to apply programming to any problem they want to, not just scientific work.

One downside of Python is that the libraries are often segmented between different Python versions. I have come across libraries restricted to any one of the still common 2.6, 2.7, or 3.x versions of Python, make interoperability and maintaining the code base somewhat difficult.

Python has a large and active community (as the huge number of cities hosting PyCONs demonstrate), and solutions to most problems can be found much easier than for most commercial software, and there's a much smaller barrier to fixing bugs in one's own code, in libraries, and in Python itself.

As the previous sections demonstrate, there are many libraries that cover almost every aspect of the laboratory work. There are new libraries created weekly, and the quality of the existing libraries are increasing as well.

Finally, Python itself is a great language, as it promotes best practices (see for example PEP 20, The Zen of Python http://www.python.org/dev/peps/pep-0020/), quicker to develop, and has easier to read codes, which creates better software developers as well as better scientists.

I would like to thank Spike Curtis and Kai Carver for their helpful comments during the revision of this manuscript.

References

Ng, YY and Ling, MHT. 2010. *Electronic Laboratory Notebook on Web2Py Framework.* In: Peer-Reviewed Articles from PyCon Asia-Pacific 2010. The Python Papers 5(3): 7.

Bezanson, Jeff, and Karpinski, Stefan. 2013. J*ulia and Python: a dynamic duo for scientific computing.* SciPy 2013, conference talk.