

Measurement Automation using Python

Dr. Richard G. Ranson, Radio System Design Ltd.

Abstract

In this paper and the accompanying presentation, python will be demonstrated as a rapid development programming environment for measurement automation using the GPIB interface common on many instruments. The free python language, a low cost USB to GPIB interface and a basic driver written in python allows live instrument control from a command shell. To illustrate the power and utility of this approach, a demonstration will fetch array data from a network analyser, then use standard libraries to manipulate and plot the results. Finally to show the extendibility of this idea, the paper presents a python based graphical user interface (GUI) to an HP8753 network analyser for plot capture and storage. The concepts demonstrated are very cost effective, highly productive and ideally suited to small and medium sized enterprises.

Introduction

Python is a modern, cross platform programming language, originally aimed at scripting, but has now matured into a full featured language. There are modern graphical user interface development tools and an extensive set of mature libraries to enable quick and comprehensive solutions to a wide range of applications. Python and the associated extensive set of tools demonstrated in this paper are also free under the GNU GPLv3 license.

The core Python interpreter and fundamental libraries are available from [1]. However this paper recommends and uses a distribution called Pythonxy [8] because it includes a number of stable and important extensions. In particular it includes the IPython command interface, specifically targeted at interactive use and the 'big three' scientific extensions libraries (scipy, numpy and matplotlib) which provide a wealth of established scientific data structures and computational methods. This combination provides a Matlab like environment for scientific calculation and data visualisation. The distribution also includes Spyder a program development environment and Qt Designer which is used to create GUI applications.

This paper forms the background for the conference presentation that will include interactive use of the python tools, with the commands and results shown as figures in the text.

Python, the Very Basics

Python is an interpreted language, reminiscent of HP Basic, but object oriented with a sophisticated and extendible class hierarchy. It is unusual in that it is case sensitive and has mandatory tab indenting for code blocks such as those used in loops and conditional statements. Once that is mastered, much will be familiar those that have experience with the basic, pascal, c etc

family of computer languages. One bonus of advantage to electrical engineering is a built in class for complex numbers with a library for the associated functions and operators. See appendix for a little more.

In operation, the interpreter produces a *byte code*, intermediary file which is compiled only once and enhances the run-time performance. Even so, for applications like instrument control, where speed is often limited by the nature of the measurement or operator interaction, the run-time speed of the python application is rarely significant.

The command shell, IPython, is included in the Pythonxy distribution [8] and is ideal for demonstrating capabilities as well as running code snippets to test segments prior to adoption into an application under development. The shell can be used to demonstrate basic features, but note that it also loads other libraries to provide large string, numerical arrays and matrix functionality in an interactive environment similar to Matlab. Some of this will be demonstrated in the interactive control section.

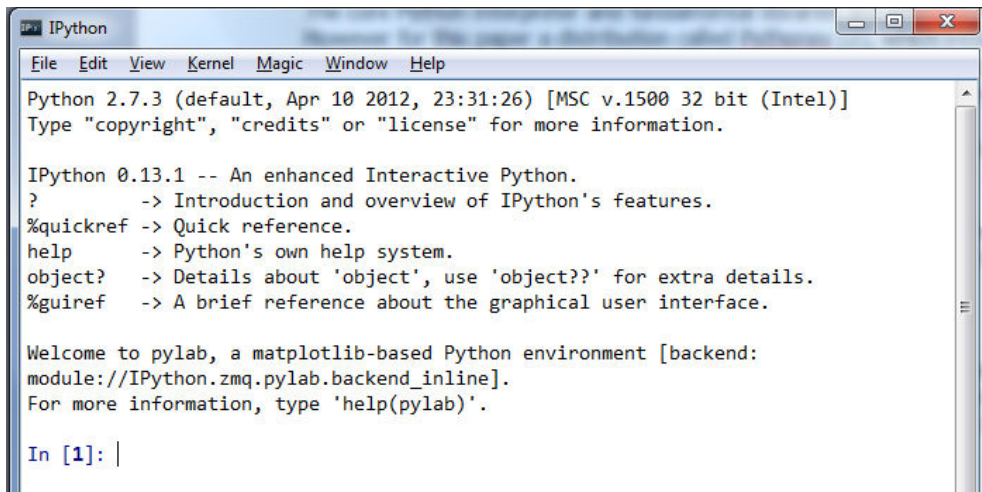


Figure 1, startup of the IPython shell.

The shell prompt is *live* allowing interaction with variables and data as well as showing *introspection*, which allows not just the data, but the class and underlying program functionality to be seen while classes, structures and operators are being typed.

Instrument Control

There are several options for PC control of GPIB instruments from vendors such as Agilent and National instruments. In this work, a low cost alternative, shown in Figure 2, from Prologix has been used. It operates rather like an old serial modem, with a number of device specific commands prefixed with '+' and all other communications being essentially pass through from the PC to the instrument and back again. This is a



Figure 2, USB to GPIB Adaptor.

very simple device to understand and work with. It does contains some specific internal registers to control data flow, end of message notification and time outs, but all those details can be embedded into a driver, written in python. Finally, since the gpib standard can interface with several instruments on the same bus, but only communicate with one at a time, the most common register to use and understand sets of the address of the instrument to be controlled.

The basic setup for the instrument and controller is shown in Figure 3. Several instruments can be controlled from the GPIB, but in this case only one is shown. The python modules and classed for the various interfaces are shown on the bottom. By convention, module names are lower case and class names use upper case for the beginning of each word (camel case).

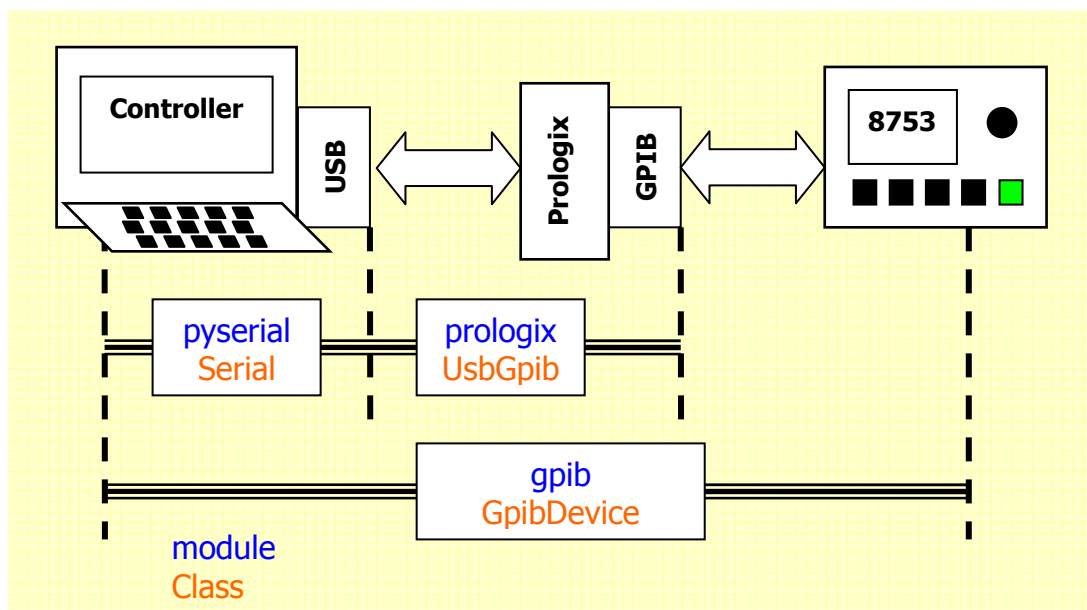


Figure 3, demonstration test set up

Pyserial is a standard python module that provides a programming interface to an RS232 serial port, which in this case is physically a USB port. The module prologix encapsulates the details of the Prologix adaptor, providing a simple interface to perform gpib commands from the computer via the serial port. The module gpib and class GpibDevice demonstrate the essentials of sending and receiving commands via the gpib interface, hiding the details of the serial communications, the gpib conversion and interacting with the instrument status register.

Interactive Session

Configuring and using the Prologix interface is simply a matter of importing the necessary module and opening the gpib communications channel using the machine specific 'comm' number for the USB port being used. (7 in this case)

```
import prologix
ib = prologix.UsbGpib(7)
```

At this level, once the instrument address has been set, basic commands can be sent backwards and forwards from the controller to instrument. There are read(), write() and query() commands to accomplish that.

```
ib.addr = 16
ib.query('star?')
' 7.500000000000000E+08'
```

The command sequence shown sets the address to 16, which is the default for the hp8753, and then shows commands to send data to

and get information back from the instrument. The default action of the interpreter is to display the result of an operation, and the reply from the query command is shown in red. The interactive nature is valuable here as it shows that the returned data is a float formatted string in Hz.

Much can be done with this rudimentary control, but recognising that several instruments can be controlled via the same interface, it is useful to expand the programming ideas one more level and this is illustrated in the base class GpibDevice.

```
import gpib
vna = gpib.GpibDevice(ib, 'hp8753', 16)
vna.ask('star?')
' 7.500000000000000E+08'
```

More sophisticated classes can be derived from this one, but an instance of it represents one instrument on the bus. Each instance holds all the control

parameters that are specific to the prologix adaptor (including the address for the instrument) so that the instance can be referred to directly, hiding all the complications of the serial to usb to gpib interfaces.

Reverting back to the basic control using the UsbGpib instance, the command

```
ib.write('form2')
rd21 = ib.query('outpdata')
```

sequence shows how to specify the data format and then get the trace data from the network analyser.

This data represents the fully error corrected measurement from the active channel. It is s21 in this case, and in the instrument format specified by 'FORM2'. Consulting the documentation shows that the returned result consists a header, a count of data bytes, then 4 bytes each for real and imaginary pairs of data repeated for each measurement point.

FORM2 IEEE 32-bit floating-point format, 8 bytes-per-data point. The data is preceded by the same header as in FORM1. Each number consists of a 1-bit sign, an 8-bit biased exponent, and a 23-bit mantissa. FORM 2 is the format of choice if your computer supports single-precision floating-point numbers.

Figure 4, hp8753 array data formatting information.

Knowing that there are 801 points in this measurement, you can easily check that the data has been transferred from the network analyser.

```
len(rd21)
6412
```

The standard command len() returns the length of an object which in this case is the number of bytes, and that corresponds to the expected number from the hp8753 documentation.

$$(\text{header} + \text{count} + n * (\text{real} + \text{image})) = 2 + 2 + 801 * 4 * 2 = 6,412.$$

Decoding such data could be challenging, but the python module library makes it straight forward. The module *struct*, provides an interface to/from arbitrary binary or string data, mapping it into a specific data structure using a compact descriptor string. In this case only the `'unpack_from()'` function is needed and used to interpret the binary array of bytes sent from the hp8753.

```
from struct import unpack_from
count = unpack_from('>h', rd21, 2)[0]
count
6408
```

This function takes 2 parameters with a third optional start value (default=0). The first parameter is a format string to specify how to interpret the data provided via the

second parameter. The byte count value is a 2 byte integer after the header and so starts at position 2. Extracting an integer required the code `'h'` and the only other complication is the byte order of integer. In this case it is big-endian and the *struct* module handles this easily using the prefix `'>'`. (6408 = 801*4*2) Working interactively is a clear advantage here, because if you don't use the `'>'`, then the PC assumes little-endian and you can see the wrong value extracted. (2073 in this case)

The last point is that `unpack_from()` always returns a tuple, even if there is only one value, so the slice `[0]` is used to return the first (and only) value.

Extracting the real and imaginary data pairs is also straight forward using the same principles. There are 801 pairs of floating point values in 4 byte sequences, which is `count/4` and a string formatting instruction can be used

```
'>{:d}f'.format(count/4)
'>1602f'
fmt = _
d21 = unpack_from(fmt, rd21, 4)
```

to create the required format code (`'>1602f'`). In the command sequence, left I have also shown another useful tip; you can see the result of an operation interactively and then keep it because the last

result is always held in the variable `'_'` (underscore). So assigning that to `fmt` saves it for future use. After the second `unpack_from()` operation, the variable `d21` is a list of 1602 floating point values with the real and imaginary parts interleaved in the list.

Once the measured data has been read and interpreted, it is then convenient to re-arrange it into a complex number vector and this is the speciality of the *numpy* module. After importing *numpy*, an empty array of the correct size is

```
import numpy as np
s21 = np.empty(len(d21)//2, 'complex')
s21.real, s21.imag = d21[::2], d21[1::2]
```

created and the elegance of the slice operator shows how the `d21` list can be re-arranged into a complex numbered vector. The slice operation

is `[start:stop:step]` with blanks indicating the extremes, so the real parts are extracted from the start to the end stepping is 2s. Similarly the imaginary parts are from 1 to the end stepping in 2s.

The last thing to illustrate is that the IPython environment also gives access to data plotting and visualisation. It is actually via the *matplotlib* module, [8] but it is imported automatically into the IPython namespace, so the commands are available directly.

```
start = float(ib.query('star?'))/1e6
stop = float(ib.query(' stop?'))/1e6

freq = np.linspace(start, stop, len(s21))
ms21 = 20*np.log10(abs(s21))
plot(freq, ms21)
```

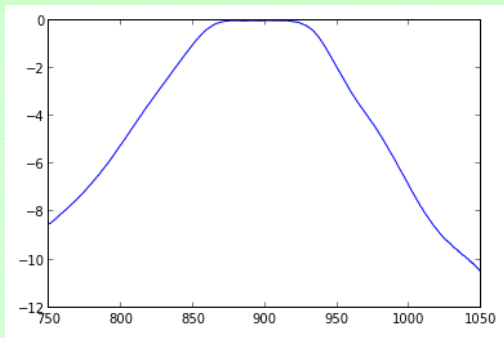


Figure 5, inline plotting from the IPython shell.

First, get the start and stop frequencies from the network analyser, remembering that the result is a float formatted string in Hz. Then, using *numpy*, create a frequency vector to correspond to the measure data. Next, form the magnitude squared values for s21 from the real and imaginary parts. Finally plot the values to see the results.

There are other plot options, including creating a separate figure that can then be annotated with axis labels, titles etc. See the *matplotlib* samples page to illustrate the numerous options. [8]

Much of this should be familiar to Matlab users, which is no coincidence, but the important point is that the instrument control is via a low cost adaptor, and simple driver written in python, with the python language, libraries and all other the tools free. Also, never mind debugging and single step execution of programs, the command shell allows you to work with the variables and data on the fly, interacting with them and developing the program steps as you go. This is a simple, yet powerful and very productive concept.

Python GUI Applications

Finally, in order to illustrate that this concept is not limited to simple command line ideas, Figure 6 shows a plot capture dialog that is part of an automated test sequence for products made at Radio System Design Ltd.

The application is developed using the Qt tool set that is an established open source, cross platform design framework. [9] There is a library of GUI (graphical user interface) objects, called widgets that can be used to display information and interact with the user in the familiar GUI manner. The interface specifies 'signals' which are essentially screen, keyboard and mouse events that can then be linked to code via 'slots' to achieve the desired interactive effect. The plot capture example is part of a final test sequence, making a particular device measurement and showing the result before adding suitable information labels and saving the results in serialised product folders for record keeping.

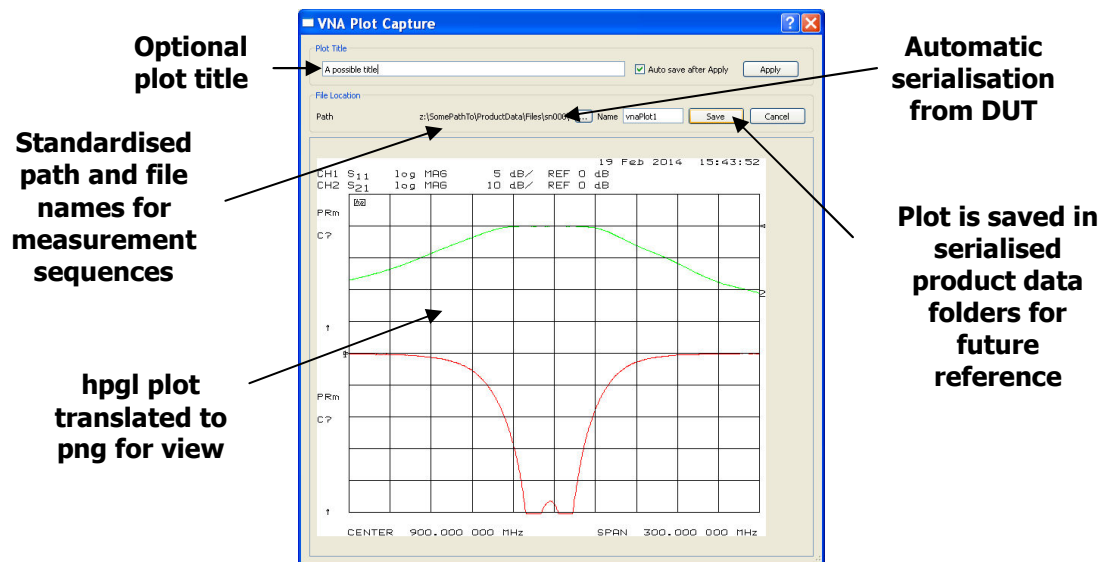


Figure 6, GUI example of a plot capture dialog.

Within this application it is worth noting another excellent example of the power of python and the standard libraries. The plot capture data from the hp8753, like many instruments is a standard vector graphics format called HPGL. This was designed for HP paper plotters and is rather old with no support in the Qt GUI framework. But there is a command line utility called hp2xx that converts HPGL into a number of other graphics formats including png that can be displayed in a Qt, GUI widget. The code to do this is rather complicated because of the cumbersome interface to windows API calls, but is sketched in Figure 7. Essentially the HPGL data read from the network analyser via the `UsbGpib` instance is written to a temporary file stream created from the python `tempfile` module. Then the `subprocess` module gives access to the windows API, spawning a separate process to run the `hp2xx.exe` file translation utility with the required command line parameters. Then by piping the data in via the temp file and out via the stdout pipe, the output png file format can be loaded into a Qt image widget and displayed in the dialog as seen in Figure 6. This is a very elegant solution to help make sure the plot looks right before saving the HPGL data and moving on to the next measurement. The alternative, creating a dedicated HPGL command interpreter just to display the plot on the screen would have been very complex and time consuming.

```

hpgl = ib.query('outpplot')
from tempfile import SpooledTemporaryFile as STF
spf =STF()
spf.write(hpgl)
import subprocess
pipe = subprocess(['hp2xx.exe', '-q', '-f-', '-mpng', '-c12345611'], stdin=spf,
                  stdout=subprocess.PIPE, stderr=subprocess.PIPE)
spf.close()
out, err = pipe.communicate()

```

Figure 7, outline of hpgl plot translation code.

Conclusion

The combination of a simple USB to GPIB adaptor and Python provides a low cost, yet highly effective tool for automated instrumentation control. The Prologix device is easy to use with a full set of gpib compatible functions. The Pythonxy distribution provides a fully featured programming environment free for all use under the GNU license. The python language itself is straightforward to learn and use as well as being cross platform, highly extendable, object oriented, with an enthusiastic on-line support community. The extensive libraries provide tremendous power, with many specifically for scientific engineering.

The interpreted nature of python, with a well implemented interactive shell, makes it possible to develop applications quickly and easily. Working with data and objects interactively, is particularly suitable for instrumentation automation, proving a simple and elegant way to develop code without the labourious write, compile, run and debug cycle common to other environments. Finally, once written, code can be wrapped into a GUI shell using other free tools in the pythonxy distribution such as Qt Designer and Spyder.

References

1. Core language and libraries available from www.python.org
2. Pythonxy is a distribution that installs the core with additional tools and libraries suitable for scientific applications.
<http://code.google.com/p/pythonxy/>
3. On line book, Dive into Python, <http://www.diveintopython.net/>
4. Prologix USB to GPIB controller, see <http://prologix.biz/>.
5. Programmer's Guide, HP 8753D Network Analyzer, 08753-90256
6. The starting point for online documentation,
<http://docs.python.org/2/index.html>
7. An index to modules in the standard distribution.
<http://docs.python.org/2/py-modindex.html>
8. For matplotlib, see <http://matplotlib.org/>
9. Python Qt Class Reference
<http://pyqt.sourceforge.net/Docs/PyQt4/classes.html>
- 10.

Appendix

There is as wealth of information available on line to get you started with python [6]. The GPL nature of the license seems to encourage a helpful community of enthusiasts that are also highly knowledgeable.

This section shows some screen captures from an IPython session, just to illustrate some of the ideas that will be used in the lecture.

In [1]: <code>z1 = 1 + 1j</code>	Interactive session showing a complex variable z1 and simple operation on that value.
In [2]: <code>z1*z1</code> Out[2]: <code>2j</code>	The last result is available as the variable named <code>'_'</code> (underscore)
In [3]: <code>z2 = _</code>	
In [4]: <code>import cmath</code>	The import command adds a library to the current namespace.
In [5]: <code>cmath.sqrt(z2)</code> Out[5]: <code>(1+1j)</code>	cmath includes additional functionality to the existing sqrt function for complex values.

Figure 8, simple interactive session examples.

In python, modules are a form of library that contain code such as functions and class definitions that encapsulate higher level concepts. The module `cmath` is the complex mathematics library and is just one of numerous modules that come with python and extend the functionality into a huge diversity of applications.[7] All the modules in the standard distributions are open source, thoroughly tests and highly functional. They often borrow ideas from other programming environments, giving python a rich set of tried and tests tools and concepts, so that is rarely necessary to re-invent a feature from other languages.

The idea of *namespace*, is common concept in modern computer languages. It is analogous to *scope*, from the algol, pascal world but more powerful. At a given level of the program, the import statement makes all symbol names (including internal import statements) from that library available for use. But to avoid conflicts and confusion, the imported names are prefaced by the name of the library. Hence `sqrt` from the `cmath` module is invoked with `cmath.sqrt()`. Note that in this demonstration, the IPython shell imports several modules, including `cmath` into its own name-space; so in this case `sqrt(z2)`, without the `cmath`, will also work, but never-the-less the example illustrates the principle behind the import statement. There are also other forms of the import function that will be used and explained in other examples.

The other key ideas that will be used in the examples are the built in types called *tuple*, *list* and *dict*. (*dictionary*). They have much in common, but are subtly different. Lists and tuples contain an ordered list of any number of

items of any type; individual items are accessed by a zero based positional index and segments are accessed by *slicing*. But, lists can be altered, so elements can be over-written as well as added or removed, while tuples are fixed once created. Dictionaries are an un-ordered list of (key, value) pairs. The key is the index to the value and has to be unique and countable. The value can be any type including lists and dictionaries themselves. Many powerful concepts are based on these types. For examples, program loops operate by iterating over every element of a list, tuple or dict; strings are tuples of characters. Also the parameters for a function can be fixed in length a tuple or variable via a dict, with the result from the function any of the three.

<pre>In [1]: s = 'a string of characters'</pre>	<p>s is a tuple of characters (zero based)</p>
<pre>In [2]: s[2:8] Out[2]: 'string'</pre>	<p>[2:8] is a slice from 2 of length (8-2)=6</p>
<pre>In [3]: s[12:] Out[3]: 'characters'</pre>	<p>While [12:] is a slice from 12 to the end</p>
<pre>In [4]: s[-10:] Out[4]: 'characters'</pre>	<p>And [-10:] is 10 from the end to the end</p>
<pre>In [5]: s[0] Out[5]: 'a'</pre>	<p>[0] is the first individual item</p>
<pre>In [7]: l = [1, 'two', 3.0]</pre>	<p>L is a list of 3 different items</p>
<pre>In [8]: l.append('infinity')</pre>	<p>Append adds to the end</p>
<pre>In [9]: l Out[9]: [1, 'two', 3.0, 'infinity']</pre>	
<pre>In [10]: l.pop(2) Out[10]: 3.0</pre>	<p>Pop() will remove an element and return it</p>
<pre>In [11]: l Out[11]: [1, 'two', 'infinity']</pre>	<p>L is now 3 items</p>
<pre>In [12]: for x in l: ...: print x ...: 1 two infinity</pre>	<p>Multiline command example. For loop on each element in the list l Commands that enter a block end in : Tab indents for all the block commands See the result</p>

Figure 9, tuple and list examples with slice, add and remove operations.

This is just a flavour of the essentials that will be used in the demonstration. There is some specific syntax to learn, but once that is grasped, most of the other ideas will be familiar to anyone with some programming experience.

This is also only scratching the surface; the language is much more sophisticated and powerful than these simple examples illustrate. There is also an enormous amount of information and examples on-line as well as many ready made solutions in the wealth of tried and tested modules and extensions already written.