PyVISA

Release 1.3

Torsten Bronger

26 March 2008

Aachen, Germany
bronger@physik.rwth-aachen.de

Abstract

PyVISA enables you to control your measurement and test equipment – digital multimeters, motors, sensors and the like. This document covers the easy-to-use `visa` module of the PyVISA package. It implements control of measurement devices in a straightforward and convenient way. The design goal is to combine HT Basic's simplicity with Python's modern syntax and powerful set of libraries.

PyVISA doesn't implement VISA itself. Instead, PyVISA provides bindings to the VISA library (a DLL or "shared object" file). This library is usually shipped with your GPIB interface or software like LabVIEW. Alternatively, you can download it from your favourite equipment vendor (National Instruments, Agilent, etc).

PyVISA is free software under the terms of the GPL. It can be downloaded at PyVISA's project page. You can report bugs there, too. Additionally, I'm happy about feedback from people who've given it a try. So far, we have positive reports of various National Instruments GPIB adapters (connected through PCI, USB, and RS 232), the Agilent 82357 A, and SRS lock-in amplifiers, for both Windows and Linux. However, I'd be really surprised about negative reports anyway, due to the high abstraction level of PyVISA.

As far as USB instruments are concerned, you must make sure that they act as ordinary USB devices and not as so-called HDI devices (like keyboard and mouse).

Contents

1 An example

Let's go *in medias res* and have a look at a simple example:

```
from visa import *

my_instrument = instrument("GPIB::14")
my_instrument.write("*IDN?")
print my_instrument.read()
```

This example already shows the two main design goals of PyVISA: preferring simplicity over generality, and doing it the object-oriented way.

Every instrument is represented in the source by an object instance. In this case, I have a GPIB instrument with instrument number 14, so I create the instance (i. e. variable) called *my_instrument* accordingly:

```
my_instrument = instrument("GPIB::14")
```

`"GPIB::14"` is the instrument's *resource name*. See section 1.3 for a short explanation of that.

Then, I send the message "*IDN?" to the device, which is the standard GPIB message for "what are you?" or – in some cases – "what's on your display at the moment?":

```
my_instrument.write("*IDN?")
```

Finally, I print the instrument's answer on the screen:

```
print my_instrument.read()
```

1.1 Example for serial (RS232) device

The only RS 232 device in my lab is an old Oxford ITC 4 temperature controller, which is connected through COM 2 with my computer. The following code prints its self-identification on the screen:

```
from visa import *

itc4 = instrument("COM2")
itc4.write("V")
print itc4.read()
```

Instead of separate write and read operations, you can do both with one `ask()` call. Thus, the above source code is equivalent to

```
from visa import *

itc4 = instrument("COM2")
print itc4.ask("V")
```

It couldn't be simpler. See section 2.2 for further information about serial devices.

1.2 A more complex example

The following example shows how to use SCPI commands with a Keithley 2000 multimeter in order to measure 10 voltages. After having read them, the program calculates the average voltage and prints it on the screen.

I'll explain the program step-by-step. First, we have to initialise the instrument:

```
from visa import instrument

keithley = instrument("GPIB::12")
keithley.write("*rst; status:preset; *cls")
```

Here, we create the instrument variable *keithley*, which is used for all further operations on the instrument. Immediately after it, we send the initialisation and reset message to the instrument.

The next step is to write all the measurement parameters, in particular the interval time (500 ms) and the number of readings (10) to the instrument. I won't explain it in detail. Have a look at an SCPI and/or Keithley 2000 manual.

```
interval_in_ms = 500
number_of_readings = 10

keithley.write("status:measurement:enable 512; *sre 1")
keithley.write("sample:count %d" % number_of_readings)
keithley.write("trigger:source bus")
keithley.write("trigger:delay %f" % (interval_in_ms / 1000.0))

keithley.write("trace:points %d" % number_of_readings)
keithley.write("trace:feed sense1; feed:control next")
```

Okay, now the instrument is prepared to do the measurement. The next three lines make the instrument waiting for a trigger pulse, trigger it, and wait until it sends a "service request":

```
keithley.write("initiate")
keithley.trigger()
keithley.wait_for_srq()
```

With sending the service request, the instrument tells us that the measurement has been finished and that the results are ready for transmission. We could read them with '`keithley.ask("trace:data?")`' however, then we'd get

```
NDCV-000.0004E+0,NDCV-000.0005E+0,NDCV-000.0004E+0,NDCV-000.0007E+0,
NDCV-000.0000E+0,NDCV-000.0007E+0,NDCV-000.0008E+0,NDCV-000.0004E+0,
NDCV-000.0002E+0,NDCV-000.0005E+0
```

which we would have to convert to a Python list of numbers. Fortunately, the `ask_for_values()` method does this work for us:

```
voltages = keithley.ask_for_values("trace:data?")
print "Average voltage: ", sum(voltages) / len(voltages)
```

Finally, we should reset the instrument's data buffer and SRQ status register, so that it's ready for a new run. Again, this is explained in detail in the instrument's manual:

```
keithley.ask("status:measurement?")
keithley.write("trace:clear; feed:control next")
```

That's it. 18 lines of lucid code. (Well, SCPI is awkward, but that's another story.)

1.3 VISA resource names

If you use the function `instrument()`, you must tell this function the *VISA resource name* of the instrument you want to connect to.

Generally, it starts with the bus type, followed by a double colon '`::`', followed by the number within the bus. For example,

```
GPIB::10
```

denotes the GPIB instrument with the number 10. If you have two GPIB boards and the instrument is connected to

board number 1, you must write

```
GPIB1::10
```

As for the bus, things like "GPIB", "USB", "ASRL" (for serial/parallel interface) are possible. So for connecting to an instrument at COM2, the resource name is

```
ASRL2
```

(Since only one instrument can be connected with one serial interface, there is no double colon parameter.) However, most VISA systems allow aliases such as 'COM2' or 'LPT1'. You may also add your own aliases.

The resource name is case-insensitive. It doesn't matter whether you say 'ASRL2' or 'asrl2'.

For further information, I have to refer you to a comprehensive VISA description like http://www.ni.com/pdf/manuals/370423a.pdf.

2 visa — module contents

This section is a reference to the functions and classes of the visa module, which is the main module of the PyVISA package.

2.1 Module functions

get_instruments_list([*use_aliases*])
: returns a list with all instruments that are known to the local VISA system. If you're lucky, these are all instruments connected with the computer.

  The boolean *use_aliases* is True by default, which means that the more human-friendly aliases like "COM1" instead of "ASRL1" are returned. With some VISA systems[1] you can define your own aliases for each device, e. g. "keithley617" for "GPIB0::15::INSTR". If *use_aliases* is False, only standard resource names are returned.

instrument(*resource_name*[, ***keyw*])
: returns an instrument variable for the instrument given by *resource_name*. It saves you from calling one of the instrument classes directly by choosing the right one according to the type of the instrument. So you have *one* function to open *all* of your instruments.

  The parameter *resource_name* may be any valid VISA instrument resource name, see section 1.3. In particular, you can use a name returned by get_instruments_list() above.

  All further keyword arguments given to this function are passed to the class constructor of the respective instrument class. See section 2.2 for a table with all allowed keyword arguments and their meanings.

2.2 Module classes

General devices

class Instrument(*resource_name*[, ***keyw*])
: represents an instrument, e. g. a measurement device. It is independent of a particular bus system, i. e. it may be a GPIB, serial, USB, or whatever instrument. However, it is not possible to perform bus-specific operations

[1] such as the "Measurement and Automation Center" by National Instruments

on instruments created by this class. For this, have a look at the specialised classes like `GpibInstrument` (section 2.2).

The parameter *resource_name* takes the same syntax as resource specifiers in VISA. Thus, it begins with the bus system followed by "`::`", continues with the location of the device within the bus system, and ends with an optional "`::INSTR`".

Possible keyword arguments are:

Keyword	Description
timeout	timeout in seconds for all device operations, see section 3.1. Default: 5
chunk_size	Length of read data chunks in bytes, see section 3.2. Default: 20 kB
values_format	Data format for lists of read values, see section 3.3. Default: `ascii`
term_char	termination characters, see section 3.4. Default: `None`
send_end	whether to assert END after each write operation, see section 3.4. Default: `True`
delay	delay in seconds after each write operation, see section 3.4. Default: 0
lock	whether you want to have exclusive access to the device. Default: `VI_NO_LOCK`

For further information about the locking mechanism, see *The VISA library implementation*.

The class `Instrument` defines the following methods and attributes:

`write`(*message*)
writes the string *message* to the instrument.

`read`()
returns a string sent from the instrument to the computer.

`read_values`([*format*])
returns a list of decimal values (floats) sent from the instrument to the computer. See section 1.2 above. The list may contain only one element or may be empty.

The optional *format* argument overrides the setting of *values_format*. For information about that, see section 3.3.

`ask`(*message*)
sends the string *message* to the instrument and returns the answer string from the instrument.

`ask_for_values`(*message*[, *format*])
sends the string *message* to the instrument and reads the answer as a list of values, just as `read_values()` does.

The optional *format* argument overrides the setting of *values_format*. For information about that, see section 3.3.

`clear`()
resets the device. This operation is highly bus-dependent. I refer you to the original VISA documentation, which explains how this is achieved for VXI, GPIB, serial, etc.

`trigger`()
sends a trigger signal to the instrument.

`read_raw`()
returns a string sent from the instrument to the computer. In contrast to `read()`, no termination characters are checked or stripped. You get the pristine message.

`timeout`
The timeout in seconds for each I/O operation. See section 3.1 for further information.

`term_chars`
The termination characters for each read and write operation. See section 3.4 for further information.

`send_end`
Whether or not to assert EOI (or something equivalent, depending on the interface type) after each write operation. See section 3.4 for further information.

`delay`
Time in seconds to wait after each write operation. See section 3.4 for further information.

values_format
The format for multi-value data sent from the instrument to the computer. See section 3.3 for further information.

GPIB devices

class GpibInstrument (*gpib_identifier*[, *board_number*[, ***keyw*]])
represents a GPIB instrument. If *gpib_identifier* is a string, it is interpreted as a VISA resource name (section 1.3). If it is a number, it denotes the device number at the GPIB bus.

The optional *board_number* defaults to zero. If you have more that one GPIB bus system attached to the computer, you can select the bus with this parameter.

The keyword arguments are interpreted the same as with the class `Instrument`.

Note: Since this class is derived from the class `Instrument`, please refer to section 2.2 for the basic operations. `GpibInstrument` can do everything that `Instrument` can do, so it simply extends the original class with GPIB-specific operations.

The class `GpibInstrument` defines the following methods:

wait_for_srq([*timeout*])
waits for a serial request (SRQ) coming from the instrument. Note that this method is not ended when *another* instrument signals an SRQ, only *this* instrument.

The *timeout* argument, given in seconds, denotes the maximal waiting time. The default value is 25 (seconds). If you pass `None` for the timeout, this method waits forever if no SRQ arrives.

class Gpib([*board_number*])
represents a GPIB board. Although most setups have at most one GPIB interface card or USB-GPIB device (with board number 0), theoretically you may have more. Be that as it may, for board-level operations, i. e. operations that affect the whole bus with all connected devices, you must create an instance of this class.

The optional GPIB board number *board_number* defaults to 0.

The class `Gpib` defines the following method:

send_ifc()
pulses the interface clear line (IFC) for at least 0.1 seconds.

Note: You needn't store the board instance in a variable. Instead, you may send an IFC signal just by saying '`Gpib().send_ifc()`'.

Serial devices

Please note that "serial instrument" means only RS232 and parallel port instruments, i. e. everything attached to COM and LPT. In particular, it does not include USB instruments. For USB you have to use `Instrument` instead.

class SerialInstrument (*resource_name*[, ***keyw*])
represents a serial instrument. *resource_name* is the VISA resource name, see section 1.3.

The general keyword arguments are interpreted the same as with the class `Instrument`. The only difference is the default value for *term_chars*: For serial instruments, `CR` (carriage return) is used to terminate readings and writings.

Note: Since this class is derived from the class `Instrument`, please refer to section 2.2 for all operations. `SerialInstrument` can do everything that `Instrument` can do.

The class `SerialInstrument` defines the following additional properties. Note that all properties can also be given as keyword arguments when calling the class constructor or `instrument()`.

baud_rate
The communication speed in baud. The default value is 9600.

data_bits
Number of data bits contained in each frame. Its value must be from 5 to 8. The default is 8.

stop_bits
Number of stop bits contained in each frame. Possible values are 1, 1.5, and 2. The default is 1.

parity
The parity used with every frame transmitted and received. Possible values are:

Value	Description
no_parity	no parity bit is used
odd_parity	the parity bit causes odd parity
even_parity	the parity bit causes even parity
mark_parity	the parity bit exists but it's always 1
space_parity	the parity bit exists but it's always 0

The default value is *no_parity*.

end_input
This determines the method used to terminate read operations. Possible values are:

Value	Description
last_bit_end_input	read will terminate as soon as a character arrives with its last data bit set
term_chars_end_input	read will terminate as soon as the last character of *term_chars* is received

The default value is *term_chars_end_input*.

3 Common properties of instrument variables

3.1 Timeouts

Very most VISA I/O operations may be performed with a timeout. If a timeout is set, every operation that takes longer than the timeout is aborted and an exception is raised. Timeouts are given per instrument.

For all PyVISA objects, a timeout is set with

```
my_device.timeout = 25
```

Here, *my_device* may be a device, an interface or whatever, and its timeout is set to 25 seconds. Floating-point values are allowed. If you set it to zero, all operations must succeed instantaneously. You must not set it to `None`. Instead, if you want to remove the timeout, just say

```
del my_device.timeout
```

Now every operation of the resource takes as long as it takes, even indefinitely if necessary.

The default timeout is 5 seconds, but you can change it when creating the device object:

```
my_instrument = instrument("ASRL1", timeout = 8)
```

This creates the object variable *my_instrument* and sets its timeout to 8 seconds. In this context, a timeout value of `None` is allowed, which removes the timeout for this device.

Note that your local VISA library may round up this value heavily. I experienced this effect with my National Instruments VISA implementation, which rounds off to 0, 1, 3 and 10 seconds.

3.2 Chunk length

If you read data from a device, you must store it somewhere. Unfortunately, PyVISA must make space for the data *before* it starts reading, which means that it must know how much data the device will send. However, it doesn't know a priori.

Therefore, PyVISA reads from the device in *chunks*. Each chunk is 20 kilobytes long by default. If there's still data to be read, PyVISA repeats the procedure and eventually concatenates the results and returns it to you. Those 20 kilobytes are large enough so that mostly one read cycle is sufficient.

The whole thing happens automatically, as you can see. Normally you needn't worry about it. However, some devices don't like to send data in chunks. So if you have trouble with a certain device and expect data lengths larger than the default chunk length, you should increase its value by saying e. g.

```
my_instrument.chunk_size = 102400
```

This example sets it to 100 kilobytes.

3.3 Reading binary data

Some instruments allow for sending the measured data in binary form. This has the advantage that the data transfer is much smaller and takes less time. PyVISA currently supports three forms of transfers:

ascii This is the default mode. It assumes a normal string with comma- or whitespace-separated values.

single The values are expected as a binary sequence of IEEE floating point values with single precision (i. e. four bytes each).[2]

double The same as **single**, but with values of double precision (eight bytes each).

You can set the form of transfer with the property `values_format`, either with the generation of the object,

```
my_instrument = instrument("GPIB::12", values_format = single)
```

or later by setting the property directly:

```
my_instrument.values_format = single
```

Setting this option affects the methods `read_values()` and `ask_for_values()`. In particular, you must assure separately that the device actually sends in this format.

In some cases it may be necessary to set the *byte order*, also known as *endianness*. PyVISA assumes little-endian as default. Some instruments call this "swapped" byte order. However, there is also big-endian byte order. In this case you have to append '`| big_endian`' to your values format:

```
my_instrument = instrument("GPIB::12", values_format = single | big_endian)
```

Example

In order to demonstrate how easy reading binary data can be, remember our example from section 1.2. You just have to append the lines

[2] All flavours of binary data streams defined in IEEE 488.2 are supported, i. e. those beginning with "⟨*header*⟩ #⟨*digit*⟩", where ⟨*header*⟩ is optional, and ⟨*digit*⟩ may also be "0".

```
keithley.write("format:data sreal")
keithley.values_format = single
```

to the initialisation commands, and all measurement data will be transmitted as binary. You will only notice the increased speed, as PyVISA converts it into the same list of values as before.

3.4 Termination characters

Somehow the computer must detect when the device is finished with sending a message. It does so by using different methods, depending on the bus system. In most cases you don't need to worry about termination characters because the defaults are very good. However, if you have trouble, you may influence termination characters with PyVISA.

Termination characters may be one character or a sequence of characters. Whenever this character or sequence occurs in the input stream, the read operation is terminated and the read message is given to the calling application. The next read operation continues with the input stream immediately after the last termination sequence. In PyVISA, the termination characters are stripped off the message before it is given to you.

You may set termination characters for each instrument, e. g.

```
my_instrument.term_chars = CR
```

Alternatively you can give it when creating your instrument object:

```
my_instrument = instrument("GPIB::10", term_chars = CR)
```

The default value depends on the bus system. Generally, the sequence is empty, in particular for GPIB. For RS232 it's `CR`.

Well, the real default is not `""` (the empty string) but `None`. There is a subtle difference: `""` really means the termination characters are not used at all, neither for read nor for write operations. In contrast, `None` means that every write operation is implicitly terminated with `CR+LF`. This works well with most instruments.

All CRs and LFs are stripped from the end of a read string, no matter how *term_chars* is set.

The termination characters sequence is an ordinary string. `CR` and `LF` are just string constants that allow readable access to `"\r"` and `"\n"`. Therefore, instead of `CR+LF`, you can also write `"\r\n"`, whichever you like more.

"delay" and "send_end"

There are two further options related to message termination, namely `send_end` and `delay`. `send_end` is a boolean. If it's `True` (the default), the EOI line is asserted after each write operation, signalling the end of the operation. EOI is GPIB-specific but similar action is taken for other interfaces.

The argument `delay` is the time in seconds to wait after each write operation. So you could write:

```
my_instrument = instrument("GPIB::10", send_end = False, delay = 1.2)
```

This will set the delay to 1.2 seconds, and the EOI line is omitted. By the way, omitting EOI is *not* recommended, so if you omit it nevertheless, you should know what you're doing.

4 Mixing with direct VISA commands

You can mix the high-level object-oriented approach described in this document with middle-level VISA function calls in module `vpp43` as described in *The VISA library implementation* which is also part of the PyVISA package. By doing so, you have full control of your devices. I recommend to import the VISA functions with

```
from pyvisa import vpp43
```

Then you can use them with '`vpp43.function_name(...)`'.

The VISA functions need to know what session you are referring to. PyVISA opens exactly one session for each instrument or interface and stores its session handle in the instance attribute `vi`. For example, these two lines are equivalent:

```
my_instrument.clear()
vpp43.clear(my_instrument.vi)
```

In case you need the session handle for the default resource manager, it's stored in `resource_manager.session`:

```
from visa import *
from pyvisa import vpp43
my_instrument_handle = vpp43.open(resource_manager.session, "GPIB::14",
                                  VI_EXCLUSIVE_LOCK)
```

5 Installation

5.1 Prerequisites

PyVISA needs Python version 2.3 or newer.

The PyVISA package doesn't include a low-level VISA implementation itself. You have to get it from one of the VISA vendors, e. g. from the National Instruments VISA pages. NI sells its VISA kit for approx. $ 400. However, it's bundled with most of NI's hardware and software. Besides, the download itself is free, and one user reported that he had successfully installed VISA support without buying anything.

I can't really tell about other vendors but well-equipped labs probably have VISA already (even if they don't know). Please install VISA properly before you proceed.

Additionally, your Python installation needs a fresh version of ctypes. By the way, if you use Windows, I recommend to install Enthought Python. It is a special Python version with all-included philosophy for scientific and engineering applications.[3]

5.2 Setting up the module

Windows

PyVISA expects a file called 'visa32.dll' in the PATH. For example, on my system you find this file in 'C:\WINNT\system32\'. Either copy it there or expand your PATH. Alternatively, you can create an INI file. You must do this anyway if the file is not called 'visa32.dll' on your system.

[3] Of course, it's highly advisable not to have installed another version of Python on your system before you install Enthought Python.

Linux

For Linux, the VISA library is by default at '/usr/local/vxipnp/linux/bin/libvisa.so.7'. If this is not the case on your installation, you have to create an INI file.

INI file for customisation

If the VISA library file is not at the default place, or doesn't have the default name for your operating system (see above), you can tell PyVISA by creating a file called '.pyvisarc' (mind the leading dot).

Another motivation for setting up an INI file is that you have more than one VISA library, e. g. because two GPIB interfaces of two different vendors are connected with the computer. However, in this case I'd try to use both interfaces with one library because sometimes you're lucky and it works. Note that PyVISA is currently not able to switch between DLLs while the program is running.

For Windows, place it in your "Documents and Settings" folder,[4] e. g.

```
C:\Documents and Settings\smith\.pyvisarc
```

if "smith" is the name of your login account. For Linux, put it in your home directory.

This file has the format of an INI file. For example, if the library is at '/usr/lib/libvisa.so.7', the file '.pyvisarc' must contain the following:

```
[Paths]

VISA library: /usr/lib/libvisa.so.7
```

Please note that "`[Paths]`" is treated case-sensitively.

You can define a site-wide configuration file at '/usr/share/pyvisa/.pyvisarc'. (It may also be '/usr/local/...' depending on the location of your Python.) Under Windows, this file is usually placed at 'c:\Python24\share\pyvisa\.pyvisarc'.

Setting the VISA library in the program

You can also set the path to your VISA library at the beginning of your program. Just start the program with

```
from pyvisa.vpp43 import visa_library
visa_library.load_library("/usr/lib/libvisa.so.7")
from visa import *
...
```

Keep in mind that the backslashes of Windows paths must be properly escaped, or the path must be preceeded by '`r`':

```
from pyvisa.vpp43 import visa_library
visa_library.load_library(r"c:\WINNT\system32\agvisa32.dll")
from visa import *
...
```

[4] its name depends on the language of your Windows version

6 About PyVISA

PyVISA was originally programmed by Torsten Bronger, Aachen/Germany and Gregor Thalhammer, Innsbruck/Austria. It bases on earlier experiences by Thalhammer.

Its homepage is http://sourceforge.net/projects/pyvisa/. Please report bugs there. **I'm also very keen to know whether PyVISA works for you or not. Thank you!**

Index

W