





PyVISA: the Python instrumentation package

Hernán E. Grecco ^{1,2*}, Matthieu C. Dartailh ^{3,4,5*}, Gregor Thalhammer-Thurner ⁶, Torsten Bronger ⁷, and Florian Bauer ⁸

1 Universidad de Buenos Aires, Facultad de Ciencias Exactas y Naturales, Departamento de Física. Buenos Aires, Argentina. **2** CONICET - Universidad de Buenos Aires, Instituto de Física de Buenos Aires (IFIBA). Buenos Aires, Argentina **3** Laboratoire Pierre Aigrain, Ecole Normale Supérieure. Paris, France **4** Center for Quantum Phenomena, New York University. New-York, NY, USA **5** Institut Néel, CNRS. Grenoble, France **6** Institute for Biomedical Physics, Medical University of Innsbruck, Austria **7** Forschungszentrum Jülich GmbH (ROR 02nv7yv05). Jülich, Germany **8** Independent Researcher, Germany  Corresponding author * These authors contributed equally.

DOI: [10.21105/joss.05304](https://doi.org/10.21105/joss.05304)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [Matthew Feickert](#)  

Reviewers:

- [@aquilesC](#)
- [@sidihamady](#)

Submitted: 06 March 2023

Published: 27 April 2023

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](https://creativecommons.org/licenses/by/4.0/)).

Summary

Data-driven science requires reliable data generation, and in modern instrumentation software plays a central role to achieve this goal. Computer-controlled experiments allow for complex synchronization of sensors and actuators. Moreover, as it was already recognized decades ago, they enable on-line analysis routines to steer the experiment in real time ([Enke, 1982](#)). When combined with programming best practices, instrumentation software fosters reproducible, traceable, and open science.

An important step in the development of instrumentation software has been the definition of Virtual Instrument Software Architecture (VISA) ([Cheij, 2002](#)), which quickly became the most common API for test and measurement and is widely used both in industry and academia. It includes specifications for communication with resources over a variety of interfaces such as GPIB, PXI, VXI, SERIAL, TCP/IP, and USB. VISA abstracts away the (often complex) idiosyncrasies of those low-level protocols, so that the scientist can focus on the instrument-specific command set.

The Python package PyVISA provides an easy-to-use interface to software libraries that implement the VISA standard, enabling communication between a host computer and instruments or other devices.

Statement of need

PyVISA has become the *de facto* standard for instrument control in Python. At the heart of every PyVISA program is a `ResourceManager` instance, which allows to list available resources (e.g., devices, instruments, boards). Opening one of such devices returns a `Resource` instance with methods to send and receives streams of data, control registers and device specific flags. While this high-level API serves most use cases, PyVISA still provides a convenient way to make low-level calls for those few applications that require it.

PyVISA was designed to be used in both simple acquisition scripts and also within large and complex codebases. It takes care of resource allocation, cleaning, stream parsing, data conversion, and provides introspection capabilities. Resource instances are fully configurable in terms of termination characters and format of the data stream, defaulting to sensible values while allowing full control if necessary. Logging is used extensively throughout the library to help with debugging.

Usage

PyVISA can be installed using pip or conda. PyVISA-py and PyVISA-sim can be installed using the same tools. PyVISA currently supports Python 3.8+.

While the PyVISA documentation ([The PyVISA authors, 2020a](#)) provides a full description of its API and several typical use cases, we provide here a few simple code samples to illustrate the package. The first step is to create a ResourceManager, which can be used to list available devices:

```
>>> import pyvisa
>>> rm = pyvisa.ResourceManager()
>>> rm.list_resources()
('ASRL1::INSTR', 'ASRL2::INSTR', 'USB0::0x0699::0x0363::C065089::INSTR')
```

We open a connection to the resource:

```
>>> osci = rm.open_resource('USB0::0x0699::0x0363::C065089::INSTR')
```

Basic methods for communication with a resource are write() and read(), which send and receive data (as strings). As this is often performed together, the query() method provides this with a single call.

Next we ask for an identification by sending the *IDN? command, which is defined in the IEEE 488.2 standard and also belongs to the standard commands for programmable instruments (SCPI); most instruments support this command.

```
>>> osci.query("*IDN?")
TEKTRONIX,TDS 1002B,C102220,CF:91.1CT FV:v22.11\n
```

The instrument is a Tektronix oscilloscope. We set the channel 1 vertical scale to 100 mV/div and request the curve on the display.

```
>>> osci.write("CH1:SCALE 100E-3")
>>> data = osci.query_ascii_values('CURV?')
```

PyVISA parses the instrument's response and returns the data as a Python list containing the values from the device.

This oscilloscope also supports binary data transfer. We activate this mode at the device with two commands and then request the binary values to be returned in a NumPy array.

```
>>> osci.write('DAT:ENC RPB')
>>> osci.write('DAT:WID 1')
>>> data = osci.query_binary_values('CURV?',
                                   datatype='B',
                                   container=np.array)
```

Structure of PyVISA

Backends: To provide the main functionality for communication with external devices PyVISA calls code in external libraries. Typically these are provided by vendors that implement the VISA library specification. PyVISA is not limited to a specific VISA vendor, but is rather a frontend for multiple backends. The default backend wraps the Interchangeable Virtual Instrumentation (IVI) VISA library provided by vendors like National Instruments, Keysight VISA, R&S VISA, tekVISA and others. Alternative backends have also been developed. For example, PyVISA-py uses Python packages such as PySerial, PyUSB and others to create a VISA backend. This has been very useful to avoid relying on proprietary libraries, and to support OS or architectures not supported by commercial vendors. PyVISA-sim is a backend that provides limited support

to simulated instruments, thereby allowing testing complex application even if the device is not available.

The interface of PyVISA is organized in several layers providing different levels of abstraction:

Layer 1 (low level): provides bindings to a compiled library preserving the original function names. The wrapper defines the argument types and response types of each function, as well as the conversions between Python objects and foreign types. Some backends, like PyVISA-py, do not define this layer as they rely on other Python packages. Direct access to these functions is neither required nor recommended.

Layer 2 (mid level): provides a function-oriented Pythonic API with comprehensive and Python-friendly documentation. Access to this layer is rarely needed, only used to control certain specific aspects of the VISA library which are not implemented by the corresponding resource class. In the default backend (which wraps the IVI-VISA binary library), these functions call the low-level counterparts, adding some code to deal with type conversions for functions that return values by reference. In other backends like PyVISA-py, the backend implements the protocols defined in the VISA standard and calls other Python libraries (like PyUSB) to communicate.

Layer 3 (high level) defines the previously mentioned ResourceManager class and the Resource class and its specific subclasses (GPIBInstrument, USBInstrument, etc). It also defines VisaLibraryBase, which serves as a modular interchangeable abstraction layer. Specific backends implement communication with devices in subclasses of VisaLibraryBase (e.g. IVIVisaLibrary and PyVisaLibrary), and contain layer 2 functions as bound methods.

PyVISA also implements routines to parse and filter resources names as defined in the VISA standard. Moreover, it provides functions to convert between data structures and data streams used by most equipment. While these routines are found in many IVI binary libraries, providing a pure Python version lowers the entry barrier to generate new backends.

Finally, at the very early stages of PyVISA development, it became clear that a **command line tool** was necessary to help with bug reporting and community testing. Indeed, one of the challenges when supporting users was reproducing the issue, given the broad range of software and hardware being used with PyVISA. For tackling this problem, the `visa` command line tool was born. `visa info` provides information about the running machine (OS, version, processor, etc), Python interpreter (version, compiler), PyVISA and its backends, and the Python and binary libraries it depends on. While gathering this information, it tries to instantiate all backends for obtaining valuable information. `visa shell` provides an interactive text-based session that allows listing resources, opening them, and sending and receiving messages.

However, this name of this tool conflicted with the `visa` package related to credit cards, so `visa info` became `pyvisa-info` and `visa shell` became `pyvisa-shell`.

Future of PyVISA

Like other projects in the Python scientific ecosystem, PyVISA started and leveraged Python's success as a glue language (Oliphant, 2015), being a simple wrapper around NI-VISA. But Python grew beyond, and so did PyVISA. One the biggest most recent additions to Python is the native support to write concurrent code using the `async/await` syntax (Selivanov, 2015). This is well-suited for IO-bound code with the need for concurrency, and most instrumentation code fits this description. While IVI libraries have supported `async` reads and writes for a long time, this has not reached most instrumentation code. Most applications are synchronous or use the less appropriate multithreading or multiprocessing Python libraries. By providing `async/await` versions of its functions and methods, PyVISA will allow to write safe, concurrent instrumentation code even in complex programs.

However, evolving PyVISA is a slow process due to the difficulty to thoroughly test it. Typical continuous integration (CI) systems cannot provide access to actual instruments to test

against. Through a cooperation with Keysight, PyVISA developers now have access to a virtual instrument that is more flexible to perform tests than a real instrument, however robust CI has not yet been set up.

Acknowledgements

PyVISA has been developed over the course of many years. Following open source best practices (Raymond, 1999), we fostered a community of users that became developers, and passed project leadership onto the people that can devote time and energy. This vision has been achieved by the establishment of a PyVISA organization on GitHub, moving the project out of a personal repository (The PyVISA authors, 2020b). We appreciate greatly the contributions of the PyVISA global community that made all this possible.

References

- Cheij, D. (2002). Software architecture for building. *IEEE Aerospace and Electronic Systems Magazine*, 17(1), 27–30. <https://doi.org/10.1109/62.978361>
- Enke, C. G. (1982). Computers in scientific instrumentation. *Science*, 215(4534), 785–791. <https://doi.org/10.1126/science.215.4534.785>
- Oliphant, T. E. (2015). *Guide to NumPy: 2nd edition*. CreateSpace Independent Publishing Platform.
- Raymond, E. (1999). The cathedral and the bazaar. *Know Techn Pol*, 12(3), 23–49. <https://doi.org/10.1007/s12130-999-1026-0>
- Selivanov, Y. (2015). PEP 492 – coroutines with async and await syntax. In *Python*. Python. <https://peps.python.org/pep-0492/>
- The PyVISA authors. (2020b). *PyVISA: A Python package for support of the Virtual Instrument Software Architecture*. GitHub. <https://github.com/pyvisa/pyvisa>
- The PyVISA authors. (2020a). *PyVISA: A Python package for support of the Virtual Instrument Software Architecture*. Read the Docs. <https://pyvisa.readthedocs.io>