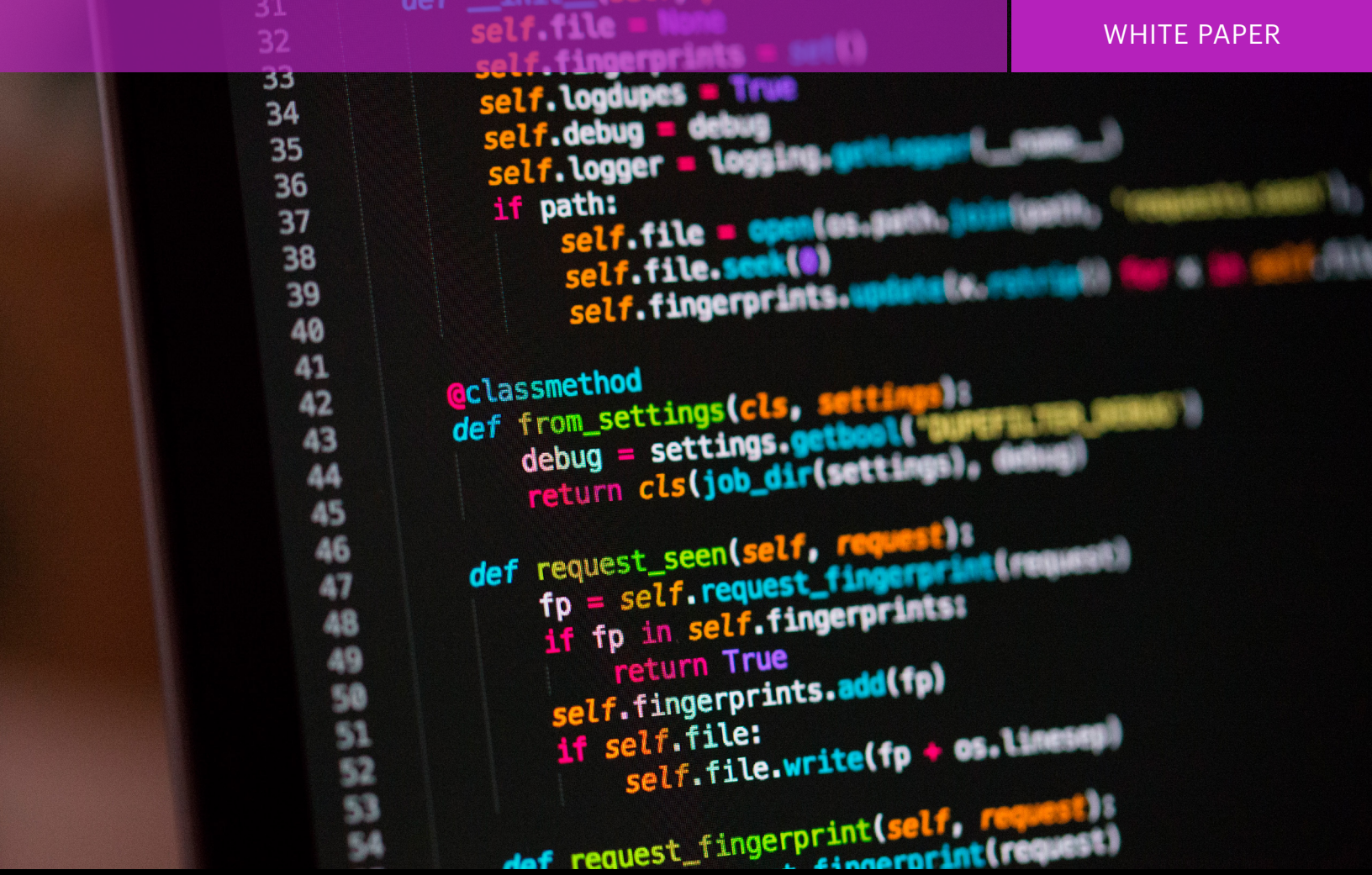
WHITE PAPER



Instrument Automation with Python

A Practical Guide

KEYSIGHT

As technology marches forward and the world becomes more automated, connected, and integrated, expectations and workflows in the professional landscape change along with it. Certain skills that used to be considered specialties become commonplace and other proficiencies previously taken for granted fall by the wayside. Automation is a skill set that is highly desirable in tomorrow's businesses, whether that means streamlining data entry, reducing time spent on timekeeping, simplifying file management, or automating time-consuming measurements.

In pursuit of this skill, the next generation of engineers are abandoning closed-source, highly-controlled legacy programming languages and are flocking toward open-source, community-oriented, agile languages like Python, Perl, Ruby, Julia, and R.

There is a discontinuity between the traditional approaches used by the test and measurement industry and the power, flexibility, and development speed of Python. The goal of this paper is to bridge that gap and help engineers learn how Python can be used effectively in instrument automation. After reading this paper, you should have a good understanding of remote instrument control and be able to write a Python script that automates a simple measurement on an oscilloscope.

> Python is heavily utilized by tech giants. Google and Facebook use Python for backend utility work, Spotify and Netflix use Python for big data analysis and recommendation services, Instagram, Quora, and Reddit use Python to build and manage their web frameworks.

Python Introduction

Python is free and open-source, which provides accountability for core developers, an enormous support base, and the ability for Python users to both inspect and improve its codebase. The Python Package Index hosts over 100 thousand Python packages that extend Python's base functionality and make it very easy to set up a Python environment to tackle virtually any programming problem. Software in the Python Package Index can be added to a Python installation using its package management tool, called "pip." Python is cross-platform, meaning it supports Windows, Mac OS, and Linux. Python's primary competitors in the instrument control space require the purchase of licenses for the base environment and additional licenses to do unlock additional capabilities. Python requires no licensing and can be installed and configured very easily. Python is easy to learn, and its syntax and structure is very human and intuitive. Many Python users describe the experience of programming in Python as writing pseudocode that actually runs.

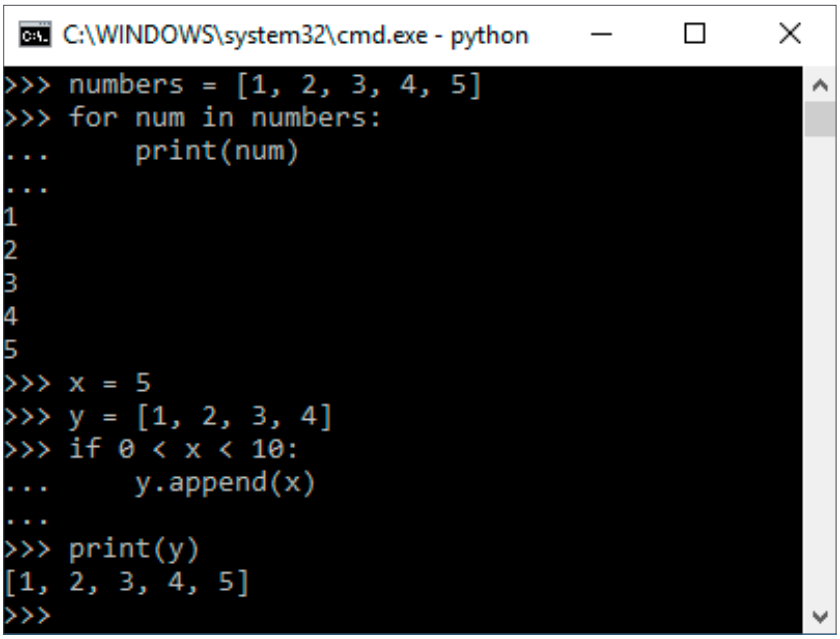


Figure 1. Python – Pseudocode that Actually Runs

Instrument Automation

Instrument automation involves writing a script or application on a computer that controls test equipment by sending ASCII messages to it. Each instrument has its own set of ASCII messages defined using the Standard Commands for Programmable Instruments (or SCPI, often pronounced “skippy”) protocol.

SCPI

Every instrument that is remotely controllable has a set of documented SCPI commands that allow the user to control the instrument using a programmatic interface rather than using front panel controls or a graphical user interface. In some cases, the SCPI command set for instruments is included in its user manual, but often the manufacturer provides a standalone programmer manual that documents all the available commands. Some standard commands are available on every SCPI-enabled instrument, including *RST (reset/default setup), *ESR? (check error status register), *OPC? (operation complete query), *CLS (clear status register), and *IDN? (identification query), but most commands are instrument-specific.

VISA

Instrument communication is generally facilitated by the VISA standard. VISA stands for Virtual Instrument Software Architecture, and it is a standardized mechanism for communication between test equipment and a controlling computer. The VISA standard is maintained by the IVI Foundation and is implemented by major test and measurement companies. Keysight’s implementation is called Keysight IO Libraries.

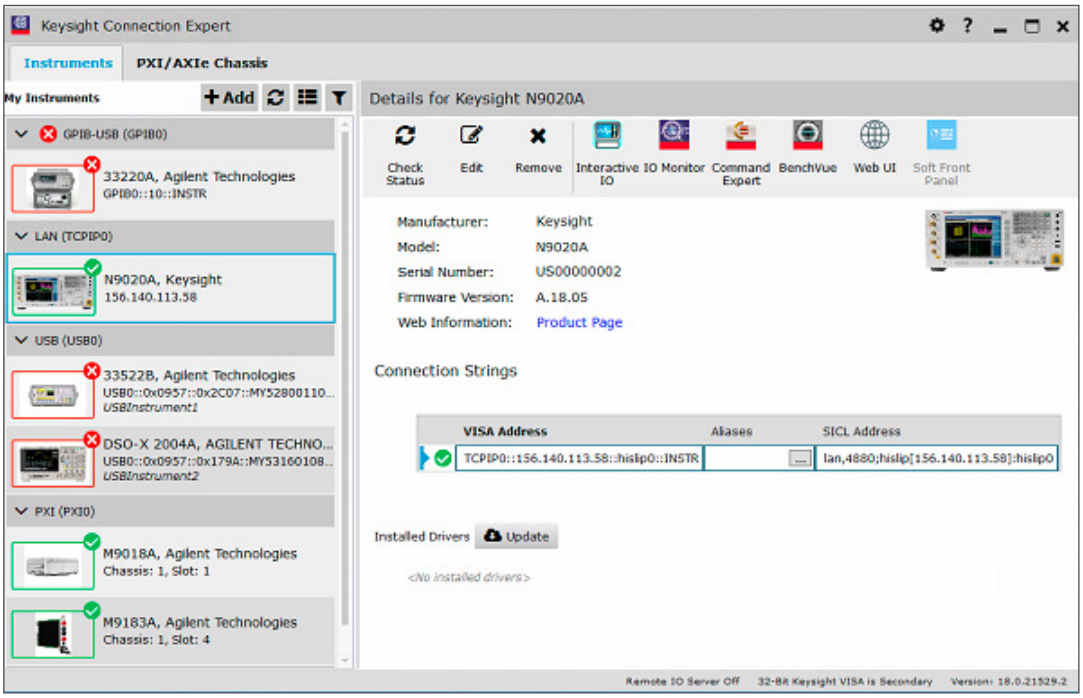


Figure 2. Keysight IO Libraries GUI

Several hardware interfaces can be used to communicate with an instrument, including USB, serial (RS-232), GPIB, and Ethernet, but VISA abstracts this away and allows the user to interact with the equipment the same way regardless of the physical hardware used to interface with the test equipment.

In most cases instruments are controlled through an Ethernet connection. The flexibility and widespread popularity of this interface allows for fast communication and abstraction of location, meaning that while both the controlling computer and test equipment are connected to the same network, they can be located an arbitrary distance from each other while maintaining communication. The throughput and latency provided by Ethernet communication is also fast enough for the overwhelming majority of use cases.

TCP/IP communication methods

There are three main protocols that can be used over an Ethernet connection to automate test equipment: VXI-11, Raw Socket, and HiSLIP (High-Speed LAN Instrument Protocol).

VXI-11 and HiSLIP are additional layers on top of the TCP/IP protocol that allow a user to communicate with an instrument over a LAN connection. They are supported by most VISA software packages and provide features like GPIB emulation and instrument locking in addition to basic communication and control. HiSLIP is significantly faster than the much older VXI-11 protocol. Raw Socket communication is a little different. It provides a lightweight LAN interface for communicating with an instrument, but little else. Notably, it is the fastest interface because it has almost no overhead. Most modern test and measurement equipment allows the user to choose between the three interfaces.

Regardless of the interface used, a few basic functions are required to successfully control an instrument: *Write, read/query, binary block read, and binary block write* functions.

- *Write* sends an ASCII command string to the instrument.
- *Read* gets whatever information is in the instrument's output buffer.
- *Query* is a write immediate followed by a read.
- *Binary block write* and *binary block read* transfer definite-length blocks of raw binary data between instrument and computer. There is a standardized format for these binary data blocks defined by IEEE 488.2 that is used throughout the test and measurement industry. Binary block read and write should not be used for normal ASCII commands.

How Does Python Help?

The functions listed above have already been implemented in Python through a widely-used Python package called PyVISA, which supports instrument communication by exposing a simple Pythonic interface to the code libraries used by the VISA standard. PyVISA also includes useful customization features, extensive documentation, and useful examples.

In addition to providing instrument communication support with PyVISA, the Python Package Index also contains powerful signal processing and data visualization tools. NumPy and SciPy are mature and full-featured numerical processing packages that make it easy to analyze and manipulate captured data. These packages include signal processing tools like a robust FFT computation engine, filter design and application features, and highly optimized vector and matrix multiplication operations, all with intuitive syntax and easy-to-use documentation. Matplotlib is a full-featured data visualization package whose commands are very similar to those of Matlab's plotting library. Matplotlib is very easy to learn, but it also has deep customization options for a huge variety of data visualization tasks.

Code example prep

Prior to controlling an instrument with PyVISA, a VISA application must be configured and the instrument must be added to the VISA resource list. This is generally done using a manufacturer-specific VISA application such as Keysight IO Libraries. Most instruments on the local area network are automatically discovered in Keysight IO Libraries, and adding a new instrument to the resource list is as simple selecting it from the list of discovered instruments. More generally, an instrument's IP address can be manually entered in the "Add Instrument" dialog for a given vendor-specific VISA implementation.

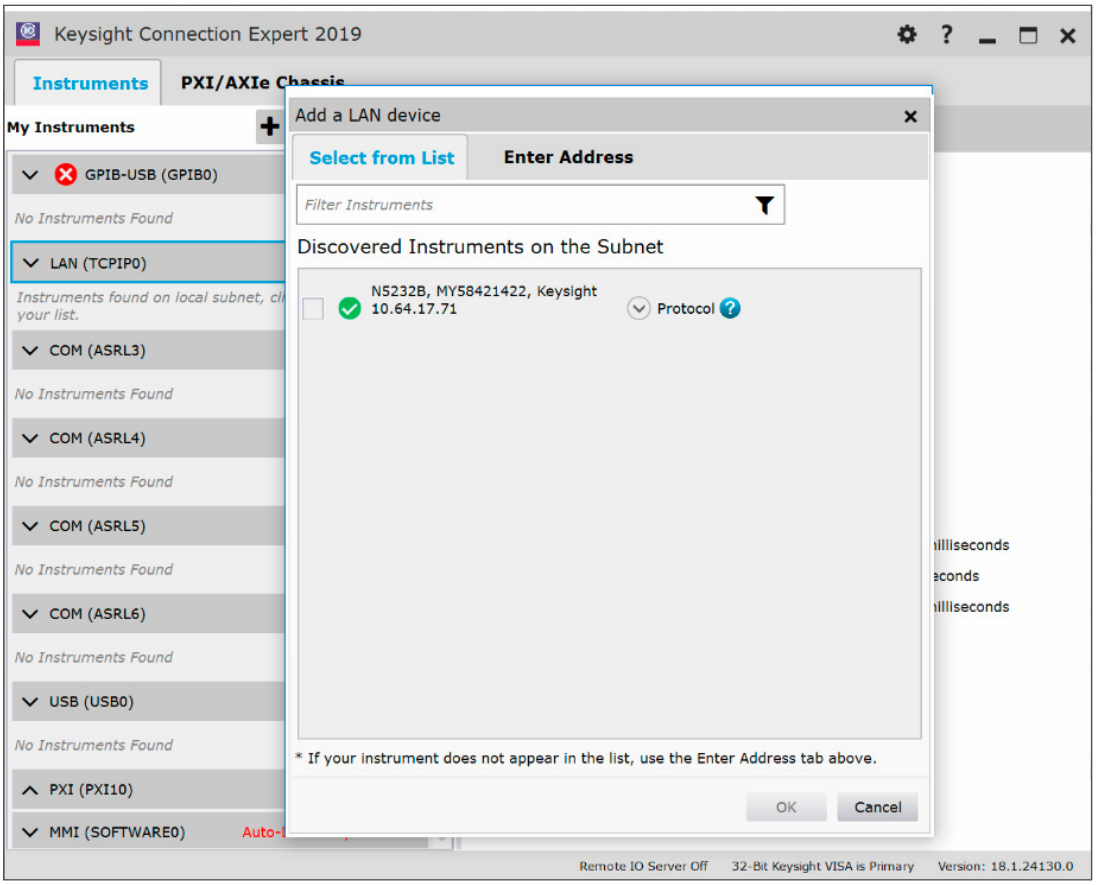


Figure 3. Keysight IO Libraries Instrument Connection

When writing test automation scripts, the programmer manual for the test equipment in question should always be available for reference. Keysight also has a free software called Command Expert that acts as an interactive programmer manual, complete with a graphical tree diagram for the command set, a handy search feature, intelligent argument insertion, real-time command testing, and script exporting.

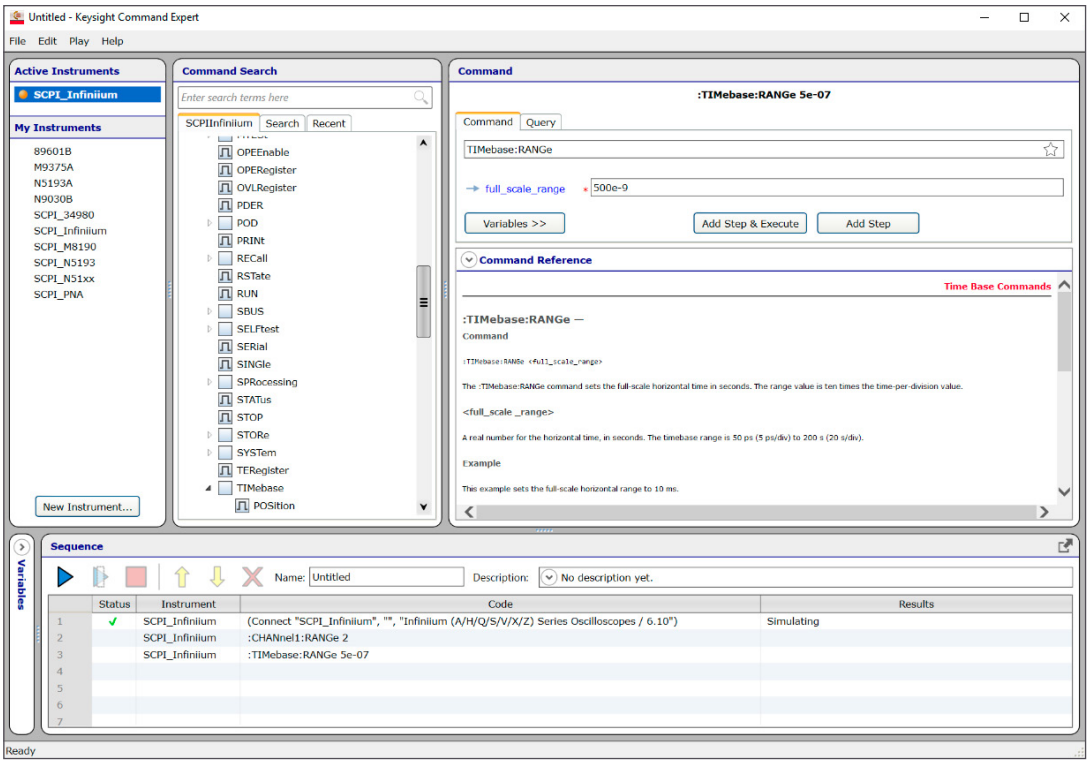


Figure 4. Keysight Command Expert GUI

Code example

Now that the basics of instrument communications have been discussed, it's time to explore a Python script that communicates with a real instrument using PyVISA. The script will set up an oscilloscope, extract waveform data, and plot it. Although this short walkthrough will touch on programming techniques, is not intended to *teach* programming and assumes a basic knowledge of Python syntax.

An instrument control script includes the following steps:

- Import required Python packages.
- Make connections to instruments.
- Define measurement variables.
- Send commands and acquire data.
- Process and visualize data.

Generally, the first few lines of code in a Python script import any Python packages required by the script. A file containing Python code has a *.py* extension and is also known as a Python *module*. A Python *package* is a hierarchical structure of multiple Python modules, and Python packages can be added to a Python installation from the Python Package Index. An *import* statement allows a Python script to use any of the code in the package or module being imported.

The first line in the example file imports the PyVISA package. This second line imports the pyplot *module* from the matplotlib *package* and assigns an alias to the module using the "as" keyword. This is purely for convenience, and it allows the user to write *plt* instead of m*atplotlib.pyplot* every time they want to use something from that Python module.

```
import visa
import matplotlib.pyplot as plt
```

A connection to the oscilloscope must be made using the PyVISA package. This script uses two pieces of code from PyVISA: the *ResourceManager* class and its *open_resource* method. The *ResourceManager* can list all the available VISA resources and create a connection to those resources. The script creates an instance of the *ResourceManager* class and assigns it to the variable *rm*. *ResourceManager* has a method called *open_resource*, which takes an instrument's VISA identifier as the argument and returns a VISA instrument object, which can be used to communicate with the instrument. In this example, the computer and the oscilloscope are connected through a local area network and the scope's IP address is known. VISA identifiers are in the form <connection type>::<address>::<interface>::INSTR. So for a TCP/IP instrument with a 192.168.1.3 IP address using the HiSLIP interface, it's VISA identifier would be TCPIP::192.168.1.3::hislip0::INSTR. This VISA identifier can also be copied directly from the VISA software running on the PC. The VISA object is assigned to a variable called scope.

```
# Make connection to instrument
rm = visa.ResourceManager()
scope = rm.open_resource('TCPIP::192.168.1.3::hislip0::INSTR')
```

It is useful to create variables containing measurement setup information that can be inserted into SCPI command strings. If these parameters need to be changed, it is far easier to access them in one place rather than searching through the script to find where specific numerical values were entered in SCPI commands. This is a simple matter of creating variables and assigning values to them. In this case, the oscilloscope's vertical range, time (horizontal) range, trigger level, and channel of interest are defined.

```
# Measurement setup variables
vRange = 2
tRange = 500e-9
trigLevel = 0
ch = 1
```

After these variables are defined, the script can begin communicating with the scope. A VISA instrument object has three basic communication methods: *write(), read(),* and *query(). write()* takes a string as an argument and sends that string to the instrument, *read()* gets whatever information is placed in the instrument's output buffer, and *query()* concatenates a *write()* and *read()* back-to-back.

Years of experience suggests that an instrument control script should start with a **RST* command. This command is defined by the SCPI protocol and is common to all test and measurement equipment. It returns the instrument to its default configuration. This is desirable because the script will configure the oscilloscope the same way regardless of any previous settings. The **OPC?* query is another incredibly useful command that enables synchronization. It waits until previous commands have finished executing before moving forward in the script.

```
# Preset and wait for operation to complete
scope.write('*rst')
scope.query('*opc?')
```

Once the scope has been brought to a default state, vertical, horizontal, and trigger settings can be configured to match the waveform in question. Vertical range, time range, and trigger level were defined earlier in the program, so those values will be inserted into their respective SCPI command stings. The values in this example were chosen for the specific conditions of this exercise.

There are many ways to format strings in Python, but this example uses F-strings. F-strings replace any expression placed inside curly braces with a value at runtime.

```
print(f'One plus two = {1 + 2}')
>>> 'One plus two = 3'
```

A quick note on SCPI commands. SCPI commands are *not* case-sensitive, but each command has an abbreviated version that is denoted by uppercase letters. Take the "AUToscale" command, for example. Writing either 'aut' or 'autoscale' to the scope will have the same effect. The scope's SCPI parser recognizes either text string as a valid command to automatically scale the horizontal and vertical ranges of the scope to match the waveform on screen.

Many SCPI commands will have arguments. As defined by the SCPI standard, arguments are either numbers or strings separated from the main command by a space. Some string arguments require double quotes around them (e.g., when specifying a save directory), and this requirement will be specified in the command's documentation.

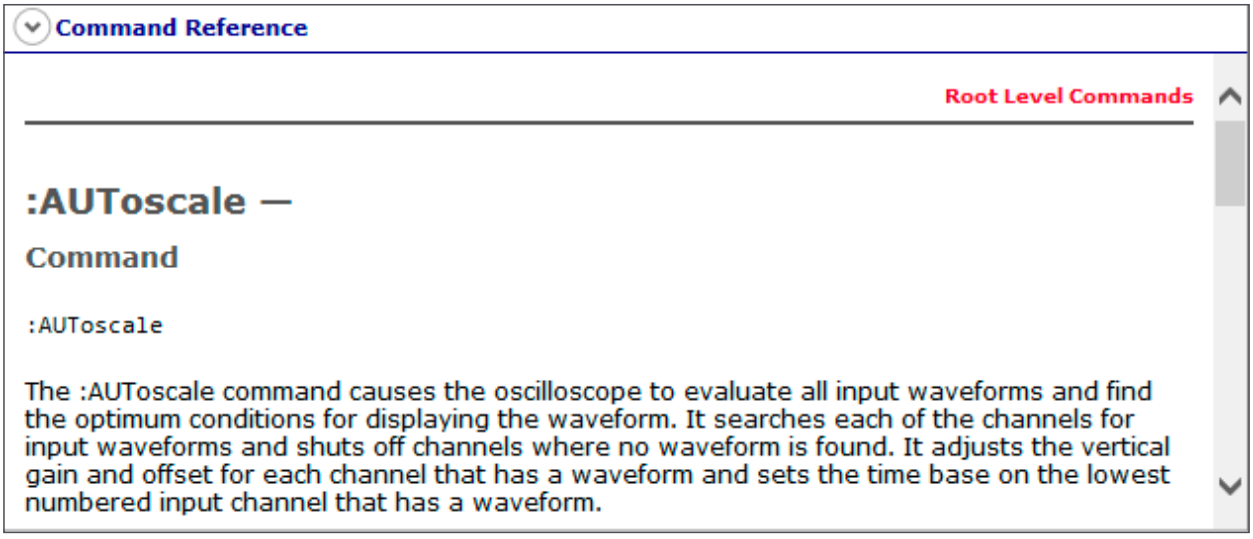


Figure 5. AUToscale command documentation

Back to the real script. The "CHANnel<N>:RANGe <range_value>" SCPI command accepts an argument that specifies the total vertical range for a given channel in volts. The vertical scale was defined in the variable *vRange*, so that variable will be inserted into the SCPI command being sent to the instrument. This process will be repeated with the "TRIGger:MODE <mode>", "TIMebase:RANGe <full_scale_range>", and "TRIGger:LEVel CHANnel<N>, <level>" commands. These commands set the vertical range of channel 1 to 2 V, the acquisition time to 500 ns, the trigger type to "Edge", and the trigger level on channel 1 to 0 V.

```
# Setup up vertical and horizontal ranges
scope.write(f'channel{ch}:range {vRange}')
scope.write(f'timebase:range {tRange}')
# Set up trigger mode and level
scope.write('trigger:mode edge')
scope.write(f'trigger:level channel{ch}, {trigLevel}')
```

It is useful to explicitly select the scope channel being used as the data source. In the case of this example, channel 1 is automatically selected by the default setup, but there are cases where other channels are of interest.

```
# Set waveform source
scope.write(f'waveform:source channel{ch}')
```

Defining the waveform format on the instrument side is an important but often overlooked step, as it determines how the script must interpret the waveform data. In this case, the "byte" argument in the "WAVeform:FORMat <format>" specifies that any waveform data sent from the scope is formatted as signed 8-bit integers. There are other formats, but "byte" results in the fastest waveform transfer rates and has sufficient resolution for this example.

```
# Specify waveform format
scope.write('waveform:format byte')
```

After the acquisition parameters have been configured, the scope needs to acquire the waveform. The "DIGitize" SCPI command makes a single acquisition and then stops.

```
# Capture data
scope.write('digitize')
```

The IEEE 488.2 standard defines a format for sending and receiving fixed-length data blocks to and from test and measurement equipment. This format includes a header that specifies how many bytes or characters are contained in the block of data, and this header must be parsed to send and receive this data correctly. PyVISA's *query_binary_values()* method parses the header and handles the transfer of data automatically. In this case it takes two arguments: the SCPI query that instructs the instrument to generate data and a datatype argument, which determines the data type used to interpret the data in the block. It is critical in this case to ensure that the instrument and the script are using the same data type. It is easy to accidentally swap signed and unsigned data types, and this confusion will lead to incorrect results.

```
# Transfer binary waveform data from scope
data = scope.query_binary_values('waveform:data?', datatype='b')
```

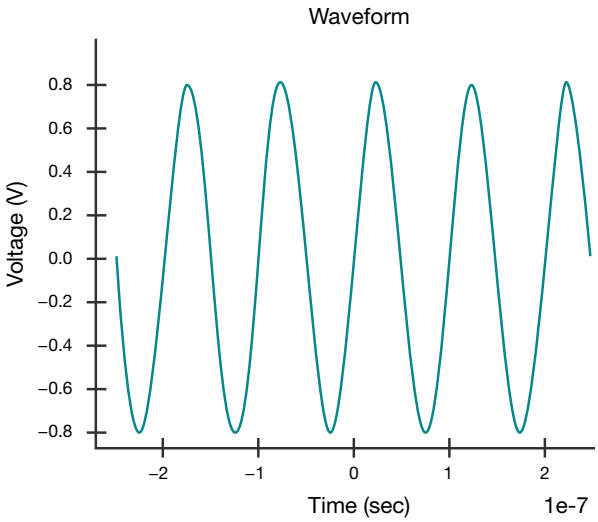


Figure 6. Script and instrument data type match

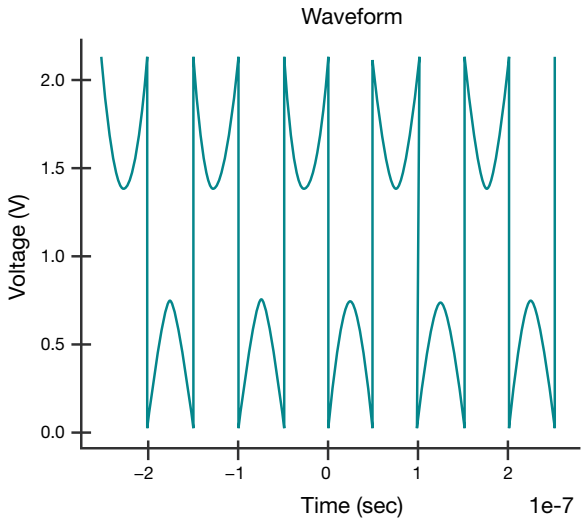


Figure 7. Script and instrument data type mismatch

Although irrelevant for the single-byte data type used in this example, big endian and little endian confusion can lead to similar situations. This is more prevalent on instruments that use higher resolution digitizers that require two or more bytes to represent each sample in the data. Fortunately, *query_binary_values()* also has an *is_big_endian* argument that allows the user to specify the incoming data's endianness.

Now that the data has been transferred to the computer, it is useful to visualize it. Generally, the data transferred from the instrument is in an unscaled binary format that must be multiplied by a scaling factor for visualization and calculation purposes. Fortunately, most instruments' SCPI command sets include queries to get the required scaling factors.

Because VISA objects return data from normal queries as strings, they must be converted to numerical data types prior to being used for numerical processing. Python makes type conversions ridiculously simple. To convert a compatible string or integer value to a floating point data type, simply call the *float()* function on the data to be converted. In this case, *float()* is called on the SCPI queries to convert the returned scaling factors to floats.

X increment, Y increment, X origin, and Y origin are needed to scale the data correctly, and there are individual commands for each. The length of the raw data returned from the scope is calculated in this block as well.

```
# Query x and y values to scale the data appropriately for plotting
xIncrement = float(scope.query('waveform:xincrement?'))
xOrigin = float(scope.query('waveform:xorigin?'))
yIncrement = float(scope.query('waveform:yincrement?'))
yOrigin = float(scope.query('waveform:yorigin?'))
length = len(data)
```

The X values can be used to create a time vector for plotting the waveform data, and the Y values can be used to convert the raw binary data from the "WAVeform:DATA?" query into voltages. In addition to standard for() loops, Python also has a concise list generation syntax called *list comprehension* that is used to scale the data here.

```
# Apply scaling factors
# Standard syntax:
time = []
wfm = []
for t in range(length):
    time.append((t * xIncrement) + xOrigin)
for d in data:
    wfm.append((d * yIncrement) + yOrigin)
# List comprehension syntax:
time = [(t * xIncrement) + xOrigin for t in range(length)]
wfm = [(d * yIncrement) + yOrigin for d in data]
```

After the appropriate scaling has been applied, the data can be plotted. Matplotlib will be used for the data visualization task. Recall at the beginning of the script where the "import matplotlib.pyplot as plt" statement was written. This means that "plt" can be used in place of "matplotlib.pyplot". Matplotlib's syntax closely mirrors Matlab's plotting syntax. The code below plots the scaled waveform data vs time, adds a title and axis labels, and displays the plot in a new window.

```
# Plot waveform data
plt.plot(time, wfm)
plt.title('Waveform')
plt.xlabel('Time (sec)')
plt.ylabel('Voltage (V)')
plt.show()
```

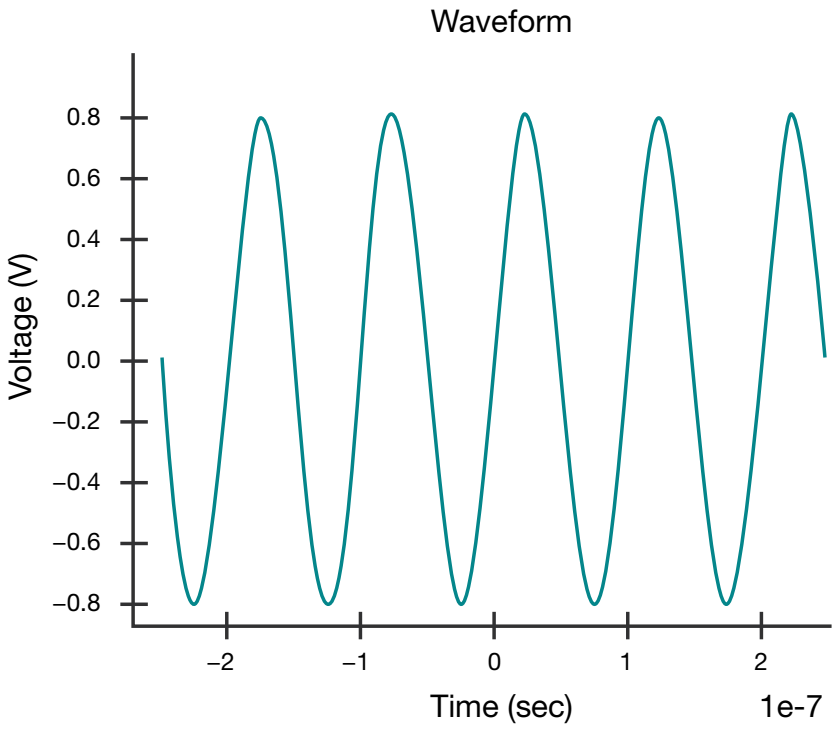


Figure 8. Plotted waveform

```
import visa
import matplotlib.pyplot as plt

# Make connection to instrument
rm = visa.ResourceManager()
scope = rm.open_resource('TCPIP::192.168.1.3::hislip0::INSTR')

# Measurement setup variables
vRange = 2
tRange = 500e-9
trigLevel = 0
ch = 1

# Preset and wait for operation to complete
scope.write('*rst')
scope.query('*opc?')

# Setup up vertical and horizontal ranges
scope.write(f'channel{ch}:range {vRange}')
scope.write(f'timebase:range {tRange}')
# Set up trigger mode and level
scope.write('trigger:mode edge')
scope.write(f'trigger:level channel{ch}, {trigLevel}')

# Set waveform source
scope.write(f'waveform:source channel{ch}')
# Specify waveform format
scope.write('waveform:format byte')

# Capture data
scope.write('digitize')
# Transfer binary waveform data from scope
data = scope.query_binary_values('waveform:data?', datatype='b')

# Query x and y values to scale the data appropriately for plotting
xIncrement = float(scope.query('waveform:xincrement?'))
xOrigin = float(scope.query('waveform:xorigin?'))
yIncrement = float(scope.query('waveform:yincrement?'))
yOrigin = float(scope.query('waveform:yorigin?'))
length = len(data)

# Apply scaling factors
# Standard syntax:
time = []
wfm = []
for t in range(length):
    time.append((t * xIncrement) + xOrigin)
for d in data:
    wfm.append((d * yIncrement) + yOrigin)
# List comprehension syntax:
time = [(t * xIncrement) + xOrigin for t in range(length)]
wfm = [(d * yIncrement) + yOrigin for d in data]

# Plot waveform data
plt.plot(time, wfm)
plt.title('Waveform')
plt.xlabel('Time (sec)')
plt.ylabel('Voltage (V)')
plt.show()
```

Figure 9. Complete Python script

Conclusion

In about 60 lines of code using free tools, this Python script has established a connection with an oscilloscope, configured acquisition settings, captured and transferred a waveform, and visualized the acquisition data.

This example only scratches the surface of Python's capabilities for test automation. In addition to remotely controlling test equipment, Python can further enhance workflows with digital signal processing, complex data visualization, or integration with machine learning and data analytics.

Resources

- 8 World-Class Software Companies That Use Python:
  https://realpython.com/world-class-companies-using-python/
- Official Python Website:
  https://www.python.org/
- Python Package Index:
  https://pypi.org/
- What Is Pip? A Guide for New Pythonistas:
  https://realpython.com/what-is-pip/
- IVI Foundation Specifications (VISA is near the bottom of the page):
  http://www.ivifoundation.org/specifications/default.aspx
- Keysight IO Libraries:
  https://www.keysight.com/en/pd-1985909/io-libraries-suite?cc=US&lc=eng
- Getting Data From the Analyzer (binary block data format):
  http://na.support.keysight.com/vna/help/latest/Programming/Learning_about_GPIB/Getting_Data_from_the_Analyzer.htm#block
- PyVISA Documentation:
  https://pyvisa.readthedocs.io/en/latest/
- Keysight Command Expert:
  https://www.keysight.com/en/pd-2036130/command-expert?cc=US&lc=eng