KEYSIGHT

Keysight Lunch and Learn :
Automating High-Speed Digital Systems with Python: When, Why, and How?

Tim Fairfield Senior Solutions Engineer
January 2026

Premise

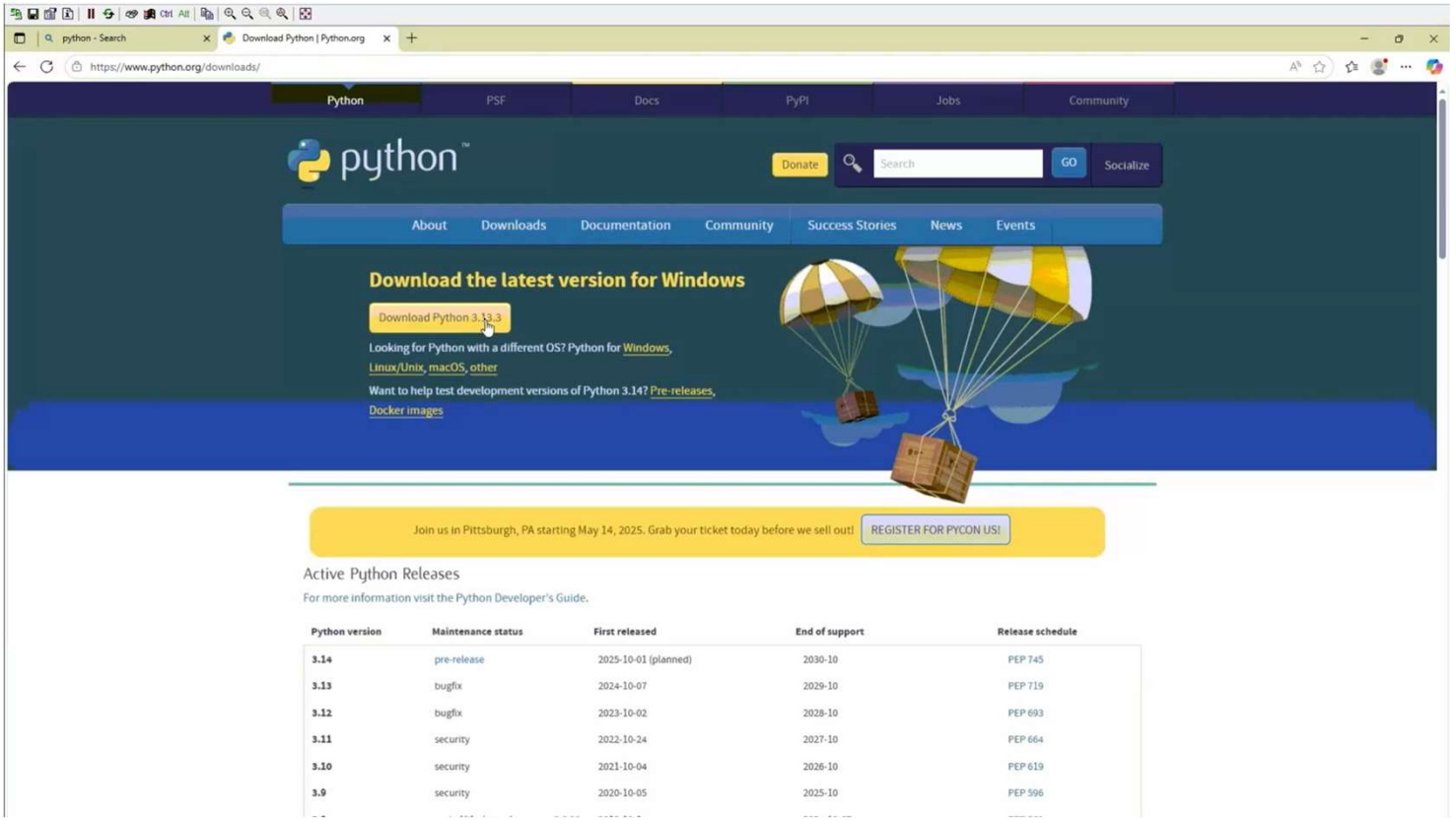
Instrumentation Automation is a vast world, This will help you get started!

- This Presentation focused on a few digital products (Scope and Bert) but applies to all product lines (RF, Power supply) and even other instruments in the industry
- Demonstrates how to get started, how to initially test communication, Keysight toolsets and procedures to get code running quickly and reliably
- Focuses on SCPI calls using Python programming using Pyvisa, There are many other ways to program (Matlab, C/C++ C#, IVI, .NET) however this is the predominant method of programming we see used daily in major companies today.
- Uses these tools to show you how to “fish for yourself”
- Use AI to speed up coding and assist you along the way
- Issues that may crop up and solutions to avoid these issues
- Get started now , whatever your coding level is.

Prerequsite Install Python 3.12

Python.org/downloads Python v3.12 or higher recommended

- Install Python 3.12,
- add to environment variables PATH (this makes package installing and running code much easier)
- Make sure to download the 64 bit version
- IDLE is the default programming editor but I prefer either Visual Studio code Or Pycharm as they are more powerful debug tools
- Install the package pyvisa from command line “pip install pyvisa”
- Verify PYVISA is installed . From interactive mode “import pyvisa”
- Run a basic Hello World make sure things are running



Why Automate in the first Place??

- **Eliminate Repetitive Tasks**
  Automate recurring procedures across one or multiple instruments to save time and reduce human error.
- **Increase Efficiency**
  Run tests faster and in parallel across multiple systems—one person can manage several setups simultaneously.
- **Scalability**
  Easily scale operations by renting or adding more systems without increasing manual workload.
- **Flexible Deployment**
  Create internal tools for personal use or deploy automation solutions to field teams or manufacturing lines.
- **Consistency & Accuracy**
  Ensure repeatable and reliable results every time, minimizing variability in testing or data collection.
- You may **no**t have **physical access** to the instrument (up a tower, in a clean room, in a hazardous environment, in a remote office i.e. Covid restrictions motivated many to do remote testing)

Why SCPI?

Standard Commands for Programmable Instruments

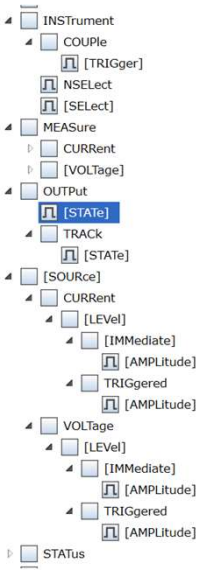
- Industry Standard for Decades
  SCPI has been a long-standing standard for instrument communication—used reliably for over 20 years.
- Universal Command Set
  Most instruments support a common set of commands (e.g., identification, querying options), making automation more consistent and predictable.
- Simplifies Integration
  Familiarity with SCPI commands allows easier scripting and control across different instruments and vendors.

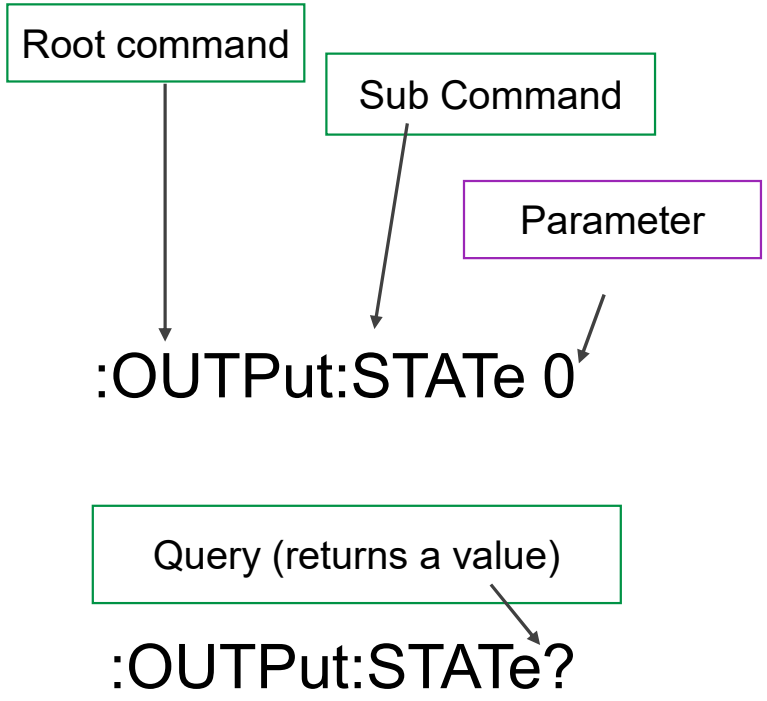
SCPI Fundamentals

If you can press a button or turn a dial on the instrument or select it in a menu, you can likely automate it via SCPI.

Syntax:

Colon (:): Separates command levels.
:OUTPut:STATe 0
Question Mark (?): Indicates a query.
:OUTPut:STATe?





scope.write("*RST;:AUT;:SING;*OPC?")

Compound statements Separated by semicolon ;
Reset the instrument (*RST)
Autoscale (:AUT)
Trigger a single acquisition (:SING)
Wait for operation complete (*OPC?)

KEYSIGHT

Common SCPI Commands

- General (applies to any Instrument)

Command	Description
*IDN?	Identify the instrument (manufacturer, model, serial, firmware)
*RST	Reset the instrument to its default state
*CLS	Clear status and error queues
*TST?	Perform self-test and return result
*OPC	Set Operation Complete flag when tasks finish
*OPC?	Query if all pending operations are complete
*SRE <value>	Set Service Request Enable register
*ESE <value>	Set Event Status Enable register
*ESR?	Query Event Status Register
*STB?	Query Status Byte Register
*WAI	Wait until all operations are complete before continuing

- Instrument Specific (Scope)

Command	Description
ACQuire:TYPE <type>	Set acquisition type (e.g., NORMAL, AVERAGE)
CHANnel<n>:SCALe <value>	Set vertical scale (volts/div) for channel *n*
CHANnel<n>:COUPling <type>	Set input coupling (AC, DC, GND)
TRIGger:MODE <type>	Set trigger mode (EDGE, PULSE, etc.)
TRIGger:EDGE:LEVel <value>	Set trigger level
MEASure:FREQuency?	Query signal frequency
MEASure:PK2Pk?	Query peak-to-peak voltage
WAVeform:SOURce <chan>	Select channel for waveform data
WAVeform:DATA?	Query waveform data
DIGitize	Start a single acquisition

Command	Description
VOLTage <value>	Set output voltage
CURRent <value>	Set output current limit
OUTPut <ON>	Turn on supply output
MEASure:VOLTage?	Measure actual output voltage
MEASure:CURRent?	Measure actual output current
SOURce:VOLTage:PROTection <value>	Set over-voltage protection limit
SOURce:CURRent:PROTection <value>	Set over-current protection limit
OUTPut:PROTection:TRIPped?	Query if protection has tripped
APPLy <voltage>, <current>	Set voltage and current in one command

- Vendor and Product line Specific Commands

KEYSIGHT

SCPI program flow & waiting mechanisms (used for reliable instrument sync/control):

Very important as some instrument operations take time and you don't want to proceed if not ready

- Default Timeout: Sets max time to wait for instrument response (prevents hangs). [Blocking]
- *OPC?: Query if all prior operations are complete; blocks until done. [Blocking]
- *OPC: Sets internal flag when operations finish; does not block. [Non-blocking]
- *WAI: Pauses program until all previous commands are finished. [Blocking]
- Instrument-specific methods: Some instruments provide unique sync/status queries (e.g., :STATus:OPERation:CONDition?, :STATus:INSTrument:RUN?)

SCPI Error Handling

Examples of errors and what they mean

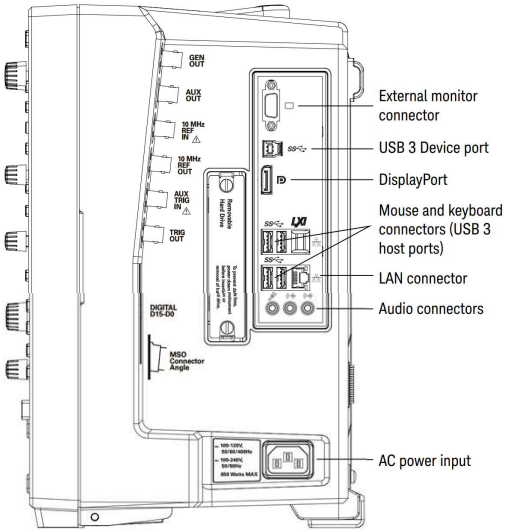
1. **-113: Undefined header**:
    1. Indicates that the command sent to the instrument does not contain a recognizable command name **1**
    2. .
2. **-410: Query INTERRUPTED**:
    1. Occurs when a valid query is sent to the instrument, but another command or query is sent before the response is fully received **1**
    2. .
3. **-420: Query UNTERMINATED**:
    1. Happens when the instrument is addressed to talk but has no response to send, often due to an invalid or missing query **1**
    2. .
4. **-100: Command error**:
    1. Indicates a syntax error or an unrecognized command **2**
    2. .
5. **-200: Execution error**:
    1. Occurs when the instrument cannot execute the command due to its current state or configuration

- To handle and clear query and command errors in SCPI, you can use the following commands:
- Clear Status:
- ***CLS**: Clears the status byte, error queue, and all event registers 1.
- Query Errors:
- **SYSTem:ERRor?:** Queries the error queue to retrieve the next error message 2.
- Standard Event Status Register:
- ***ESR?:** Queries the Standard Event Status Register to check for errors 1.
- Enable Error Reporting:
- ***ESE** <value>: Configures the Standard Event Status Enable Register to monitor specific errors 1.
- Power-On Status Clear:
- ***PSC ON|OFF**: Controls automatic clearing of event registers upon power-on 1.
- These commands help manage and resolve errors effectively in SCPI-based communication

Physical Connection Options

Ethernet is Most common, Check user guide or back panel for options

Pick 1 if an instrument has more than 1 interface and check if you need to turn the interface on in the instrument



VISA Address

TCPIP0::192.168.50.126::hislip0::INSTR

VISA Address

USB0::0x2A8D::0x4704::MY64390177::0::INSTR

VISA Address

ASRL6::INSTR

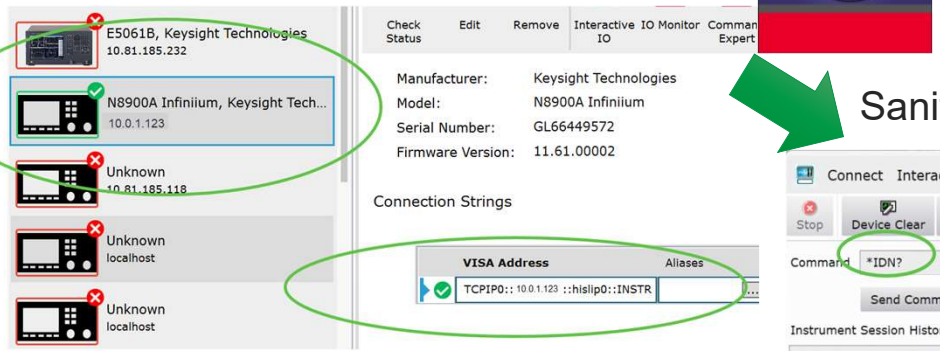
VISA Address

GPIB1::10::INSTR

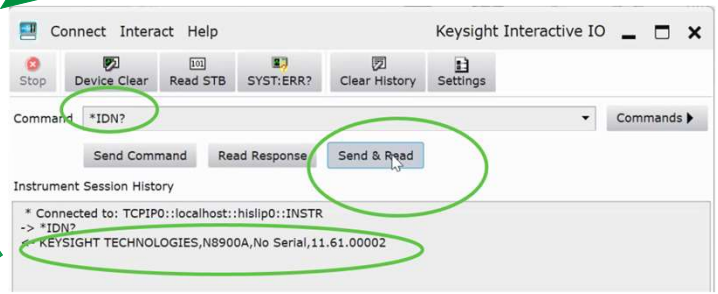
General Workflow

Simple process

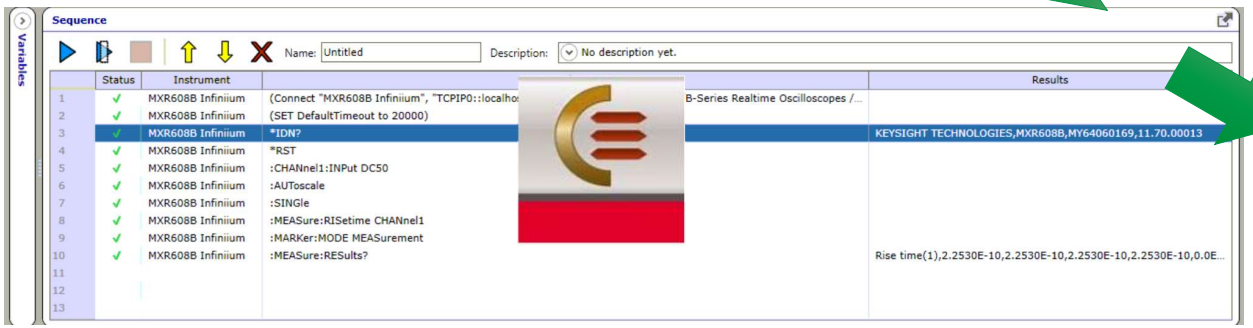
Connect with IOLIBS (Connection Expert)



Sanity Check communication *IDN?

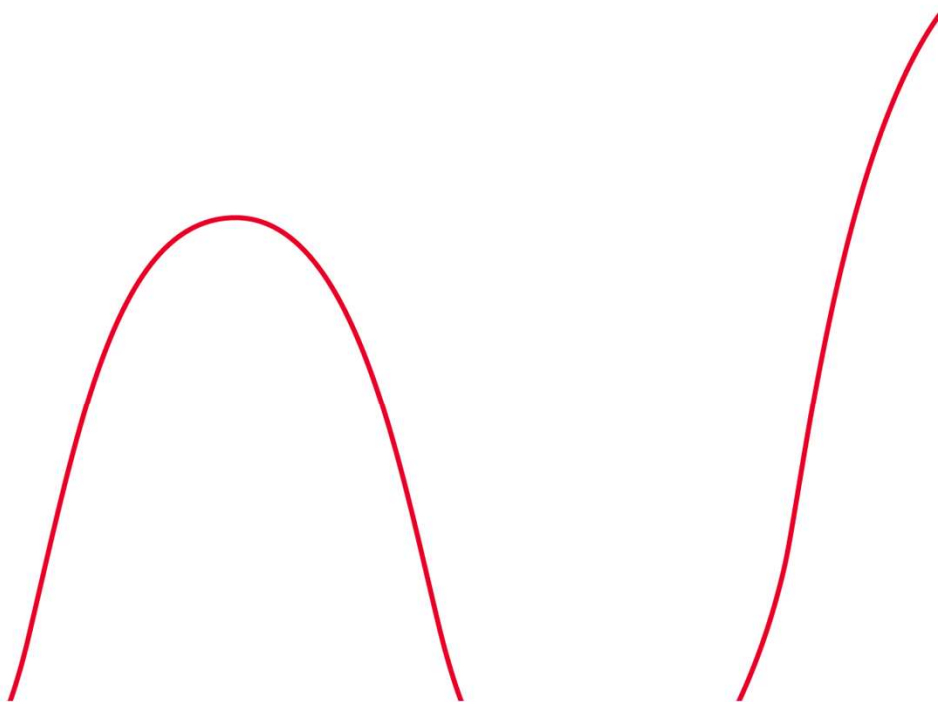


Create a sequence of commands (Command Expert)



Export to Python

Automating an MXR Infiniium Scope



Workflow

Walkthrough of a Risetime measurement Automation

Run Manual Steps

Connection, Interactive IO, IO Monitor

Commands, Sequencing, Profiling, Exporting

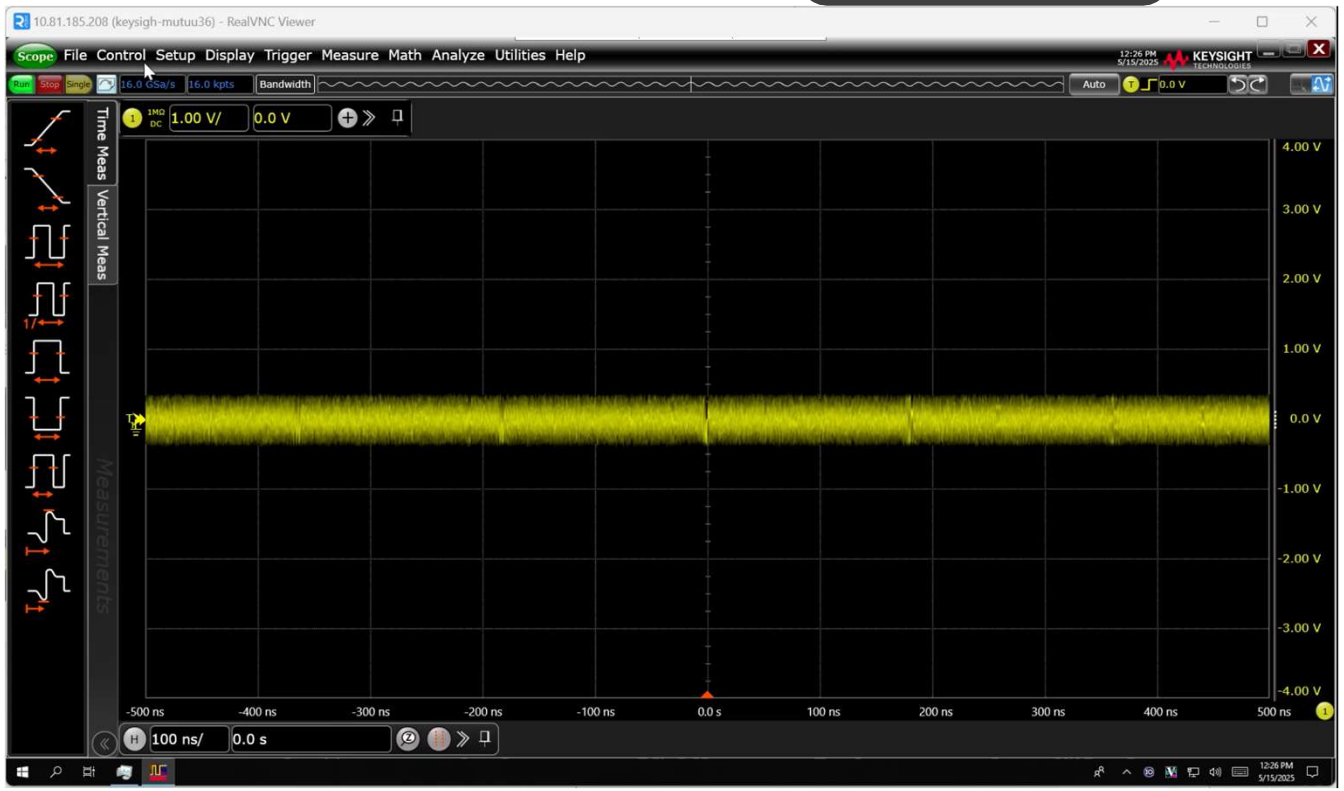
Python Code

Define your first Program

First, run through the steps manually at the scope

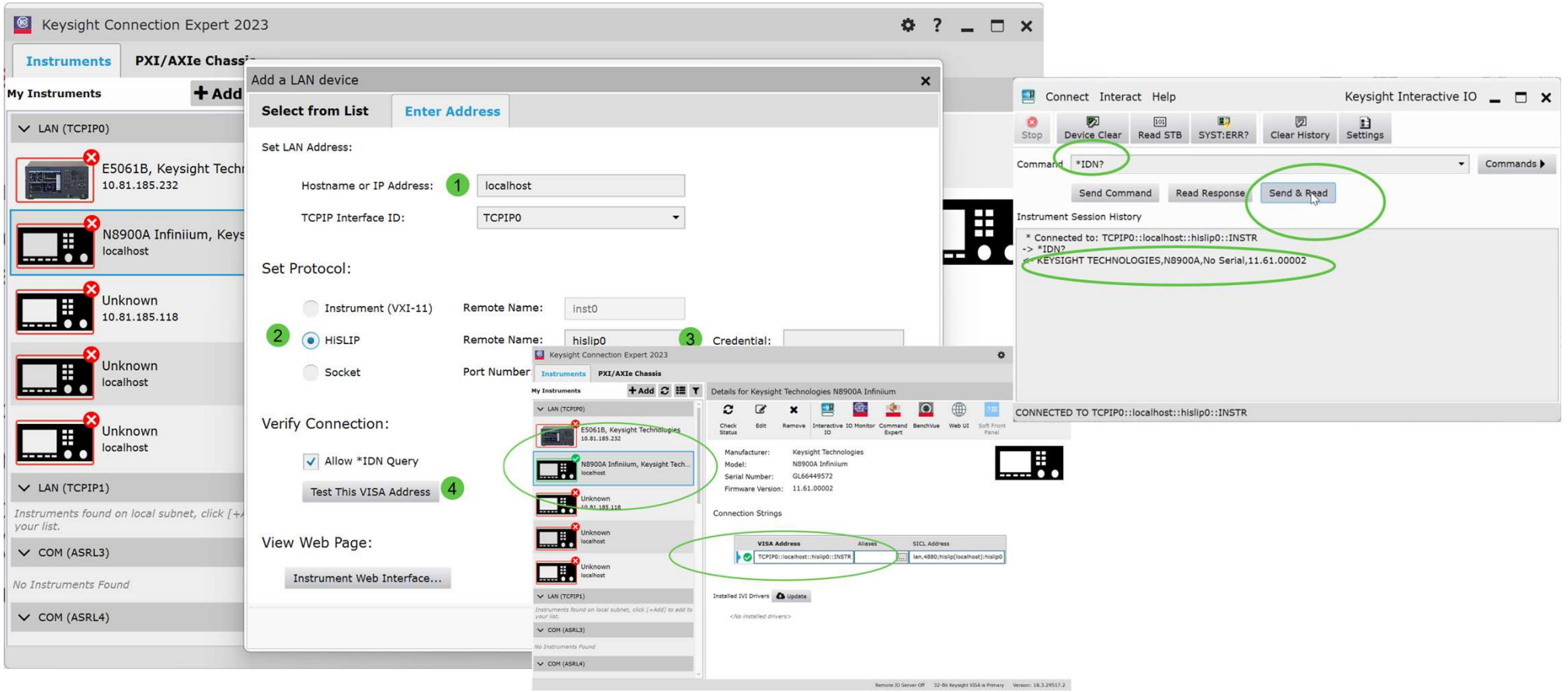
Video demo
Mxr Scope interface
Manual steps

- Goal: Make a risetime measurement on MXR scope
- Steps:
  - Default setup
  - Set input impedance
  - Autoscale
  - Add Risetime measurement
  - Press Single Acquisition
- Remember: Automation is a sequence of something you would do manually

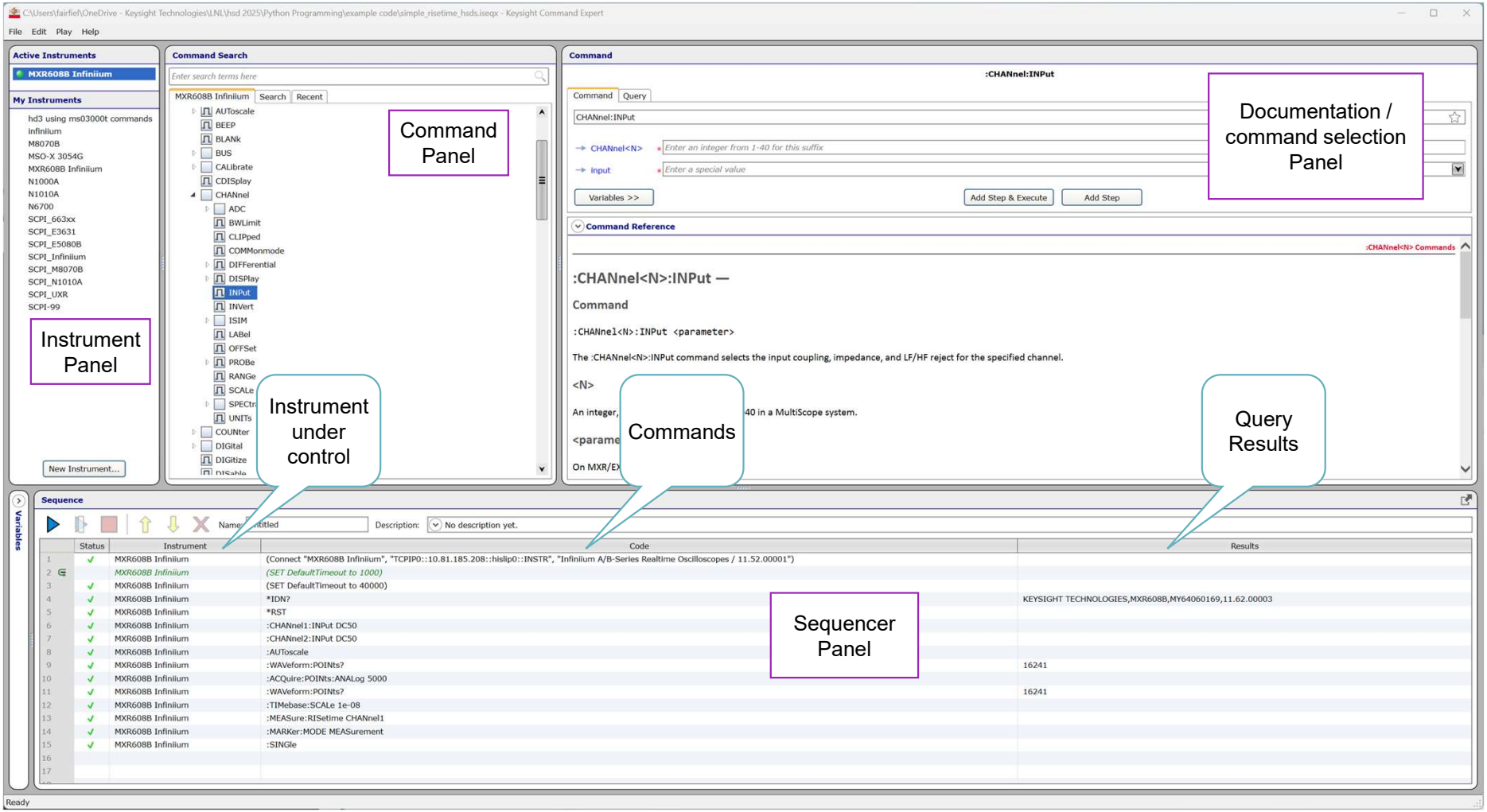


IO Libraries Connection Expert checking connection (localhost)

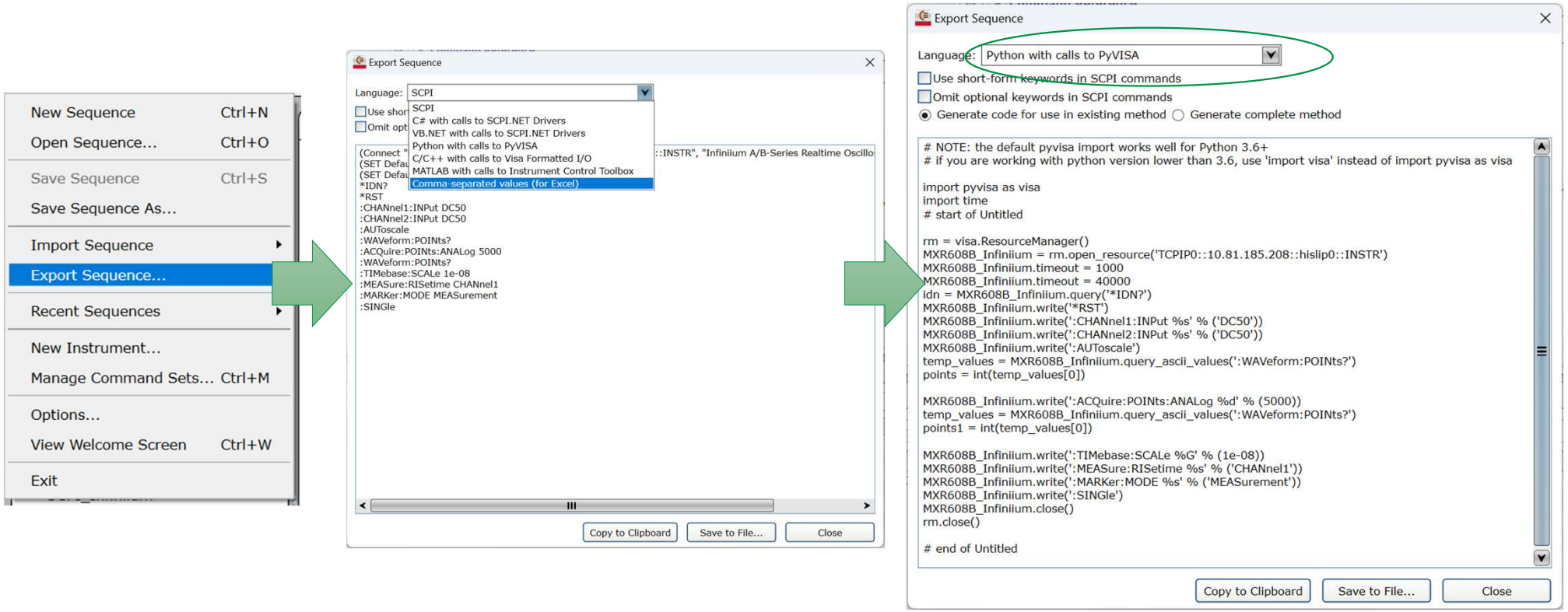
***IDN? Should always be your first command, sanity check for send and reads and make sure you are controlling the right instrument.**



Command Expert



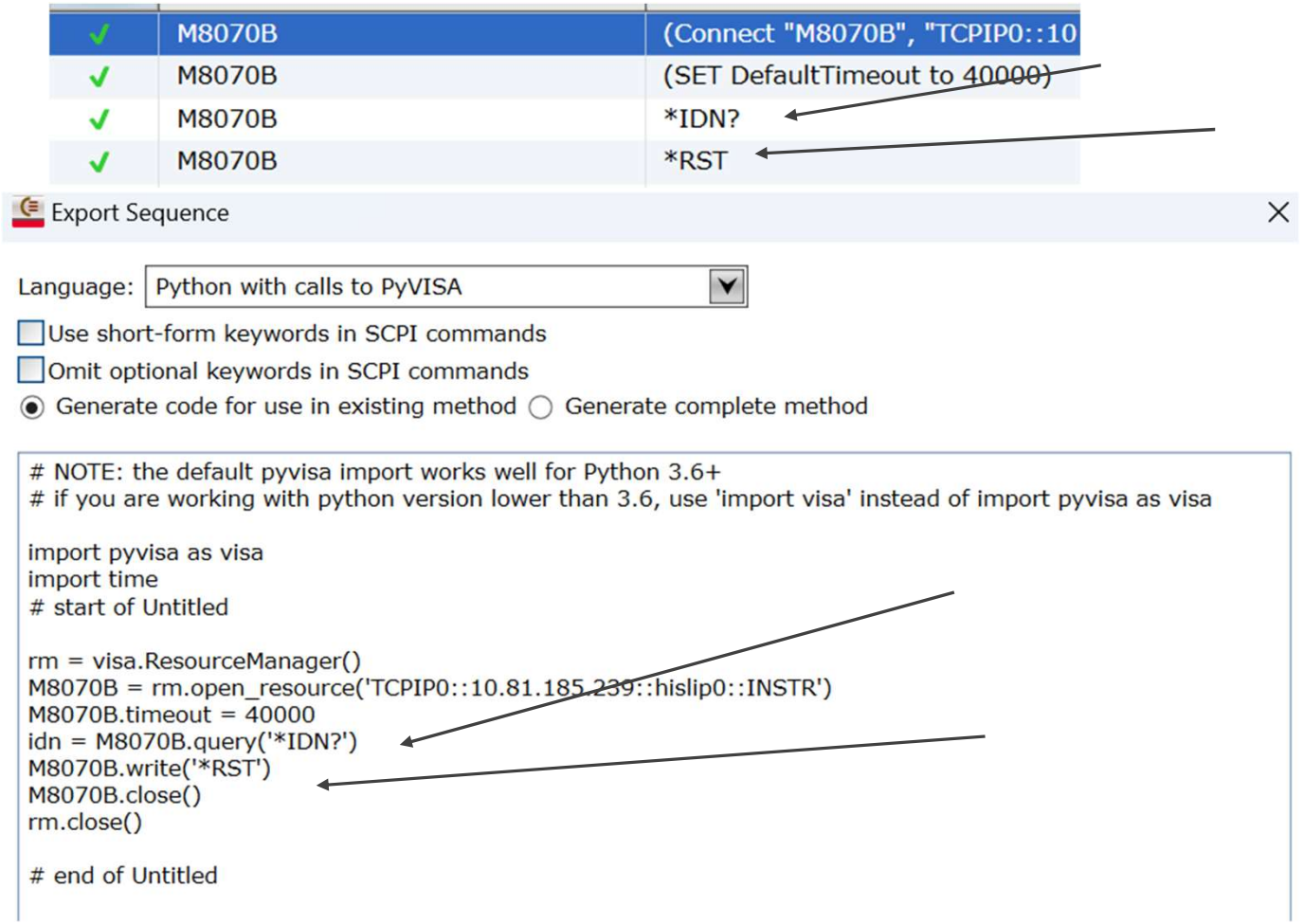
Exporting Code from Command Expert



How to Convert SCPI Recorded to Python

Use Command expert to get the Instrument object created then Glue in your code

- Create a simple sequence in Command expert that includes
  - a command
  - a prompt
- Export to Python
- Apply the appropriate command or prompt format to the raw SCPI (copy/paste)



Other Coding options: Manually Converting SCPI to Python

```
:OUTPut:STATe 'M1.DataOut1’,1
M8070B.write(":OUTPut:STATe 'M1.DataOut1’,1")
```

Command

```
:OUTPut:STATe? 'M1.DataOut1'
Value=M8070B.query("OUTPut:STATe? 'M1.DataOut1’ “)
print(f“Output state={Value}”)
```

Query

Lets try it out
(Walkthrough of MXR Risetime)

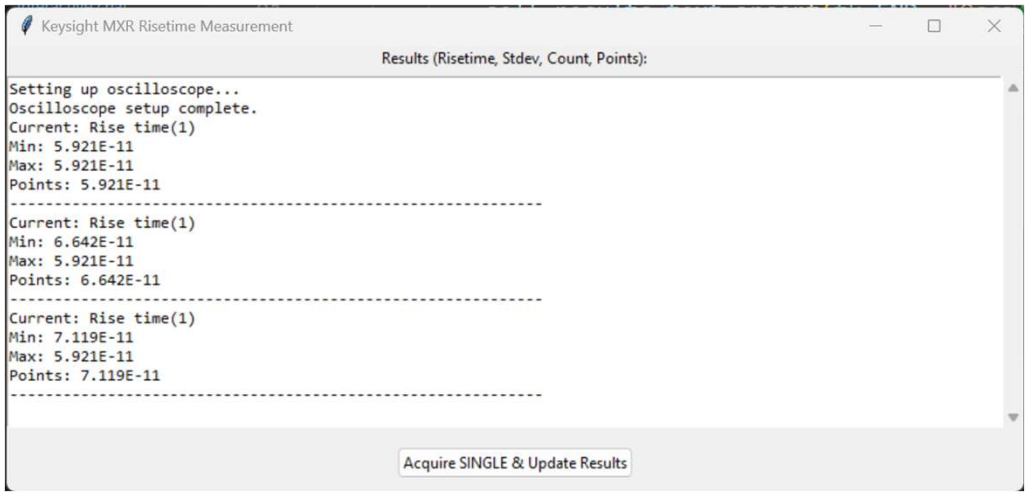
SCPI Command	Python Implementation
`*RST`	`scope.write('*RST')`
`:CHANnel1:INPut DC50`	`scope.write(':CHANnel1:INPut DC50')`
`:CHANnel2:INPut DC50`	`scope.write(':CHANnel2:INPut DC50')`
`:CHANnel1:DISPlay ON`	`scope.write(':CHANnel1:DISPlay ON')`
`:CHANnel2:DISPlay ON`	`scope.write(':CHANnel2:DISPlay ON')`
`:AUToscale`	`scope.write(':AUToscale')`
`*OPC?`	`scope.query('*OPC?')`
`:CHANnel1:SCALe 0.2`	`scope.write(':CHANnel1:SCALe 0.2')`
`:CHANnel2:SCALe 0.2`	`scope.write(':CHANnel2:SCALe 0.2')`
`:ACQuire:POINts:ANALog 5000`	`scope.write(':ACQuire:POINts:ANALog 5000')`
`:TIMebase:SCALe 1e-08`	`scope.write(':TIMebase:SCALe 1e-08')`
`:MEASure:RISetime CHANnel1`	`scope.write(':MEASure:RISetime CHANnel1')`
`:MARKer:MODE MEASurement`	`scope.write(':MARKer:MODE MEASurement')`
`:SINGle`	`scope.write(':SINGle')`
`:MEASure:RESults?`	`scope.query(':MEASure:RESults?')`

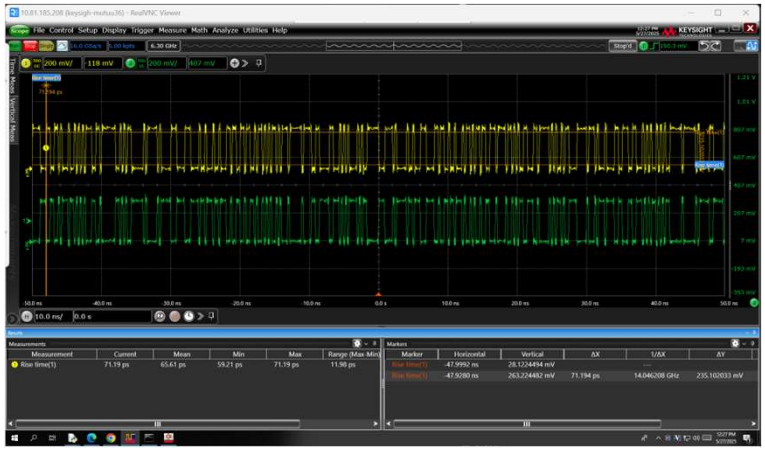
- simple_risetime_hsds_gui_version.py

Using AI to assist In Making a Simple GUI

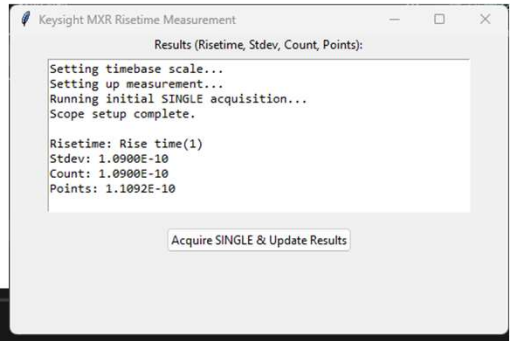
Give it a copy of your code , give it context, and iterate (you may not get the desired results right away)

- Prompt: “make this a gui , create a button that every time pressed the SINGLE command is executed and the results update in a text box below after each press. Make sure the text fits in the box “ <<and copy in code>>
- Follow up prompt: “make the display of text wider and allow scrolling also dont display the acquire button until setup is complete”
- 2nd follow up Prompt: “append new results to end instead of overwriting”





Demo 2
Lets try out the final Code
(running the Gui App)

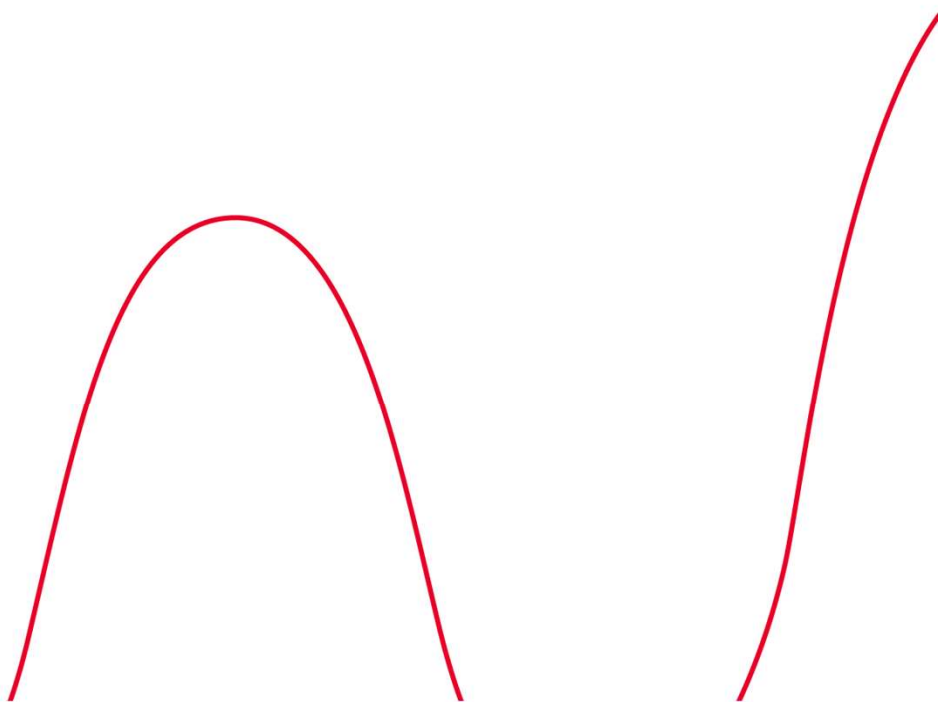



```python
import pyvisa as visa
import time

rm = visa.ResourceManager()
scope = rm.open_resource('TCPIP0::10.81.185.208::his
scope.timeout = 40000

scope.write('*RST')
scope.write(':CHANnel1:INPut DC50')
scope.write(':AUToscale')
scope.write(':SINGle')
results = scope.query(':MEASure:RESults?')
print(results)
scope.close()
rm.close()
```

Quick Reference: GUI Elements Added

GUI Element	Example Code Snippet	Purpose
Tkinter Import	`import tkinter as tk` `from tkinter import ttk`	Enables GUI creation in Python
Main Window	`class MXRGuiApp(tk.Tk):` `super().__init__()`	Main application window
Window Title/Size	`self.title("Keysight MXR Risetime Measurement")` `self.geometry("500x300")`	Set window title and size
Label	`self.results_label = ttk.Label(self, text="Results ...")`	Display static text above results
Text Box	`self.results_text = tk.Text(self, height=10, width=60, ...)`	Show results and status messages
Button	`self.button = ttk.Button(self, text="Acquire...", command=...)`	User triggers measurement
Dynamic Button	`self.button.pack_forget()` `self.button.pack()`	Hide/show button based on setup status
Status Logging	`self.log_stage(message)`	Append setup/status to text box
Window Close Handling	`self.protocol("WM_DELETE_WINDOW", self.on_close)`	Clean up VISA resources on close
Event Loop	`app.mainloop()`	Start the GUI

Waveform Data



Managing Waveform Data

Using python Matplotlib and extracting and scaling waveform

:Autoscale

*OPC?

:Single

:WAVeform:POINts 1000

:WAVeform:SOURce CHAN1

:WAVeform:FORMat ASCii

:WAVeform:PREamble?

:WAVeform:DATA?

The **preamble** can be used to translate raw data into time and voltage values. The following lists the elements in the **preamble**.

<preamble_ data>

<format>, <type>, <points>, <count> , <X increment>, <X origin>, <X reference>,
<Y increment>, <Y origin>, <Y reference>, <coupling>, <X display range>, <X display origin>,
<Y display range>, <Y display origin>, <date>, <time>, <frame model #>, <acquisition mode>,
<completion>, <X units>, <Y units>, <max bandwidth limit>, <min bandwidth limit>
[,<segment count>]

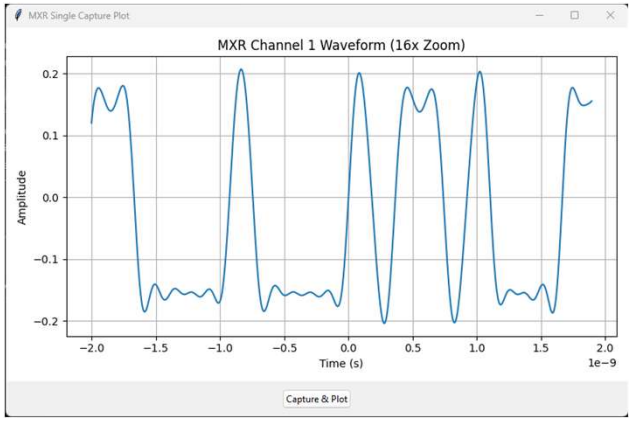
In your code, you use the following elements from the preamble:

- **num_points:** preamble_fields[2] — the number of data points in the waveform.
- **xincr:** preamble_fields[4] — the time increment between points (x-axis increment).
- **xorig:** preamble_fields[5] — the starting time value (x-axis origin).

These are used to construct the time axis for plotting the waveform:

```
num_points = int(float(preamble_fields[2]))
xincr = float(preamble_fields[4])
xorig = float(preamble_fields[5])
time_axis = xorig + np.arange(num_points) * xincr
```

Demo 3 Downloading and Plotting Scope Captured Waveform Data



Instrument Commands (Core)

```
oscilloscope.write(':AUToscale')          # Autoscale the scope
oscilloscope.query('*OPC?')               # Wait for operation complete
oscilloscope.write(':WAVeform:POINts 1000')  # Set number of waveform points
oscilloscope.write(':WAVeform:SOURce CHAN1') # Select channel 1
oscilloscope.write(':WAVeform:FORMat ASCii') # Set data format
oscilloscope.write(':SINGle')             # Single acquisition
oscilloscope.query('*OPC?')               # Wait for acquisition complete
preamble = oscilloscope.query(':WAVeform:PREamble?') # Get scaling info
waveform_data = oscilloscope.query(':WAVeform:DATA?') # Get waveform data
```

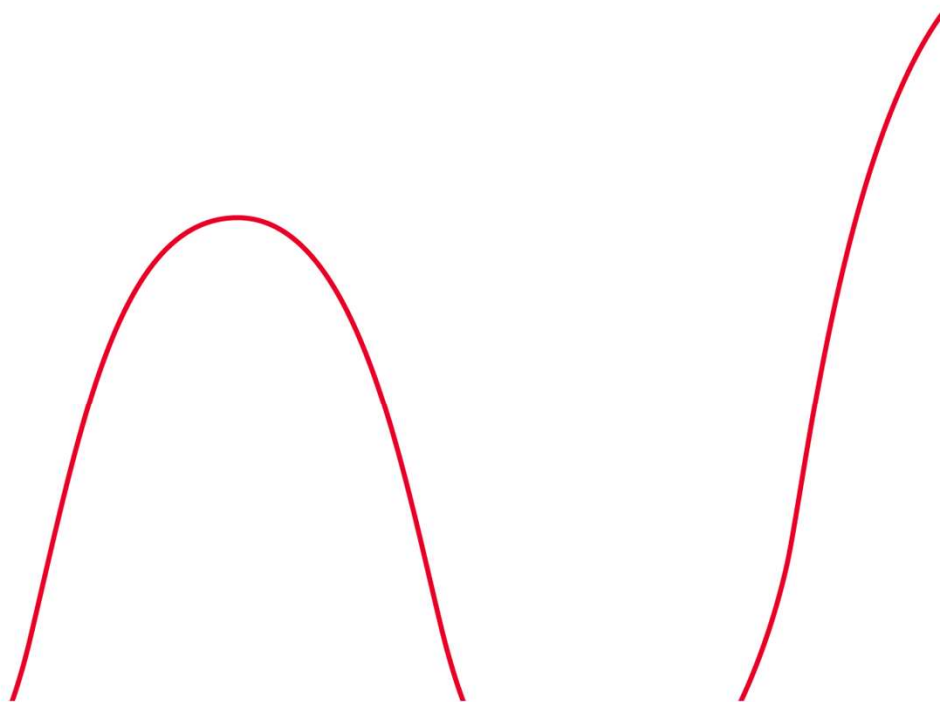
These commands set up the instrument, trigger a capture, and retrieve waveform data.

GUI & Plotting Elements

```
root = tk.Tk()                            # Create main window
fig, ax = plt.subplots(figsize=(8, 4))    # Create matplotlib figure
canvas = FigureCanvasTkAgg(fig, master=root) # Embed plot in window
canvas.get_tk_widget().pack(fill=tk.BOTH, expand=1) # Show plot
capture_btn = ttk.Button(root, text="Capture & Plot", command=capture_and_plot)
capture_btn.pack(side=tk.BOTTOM, pady=10) # Add capture button
root.mainloop()                           # Start GUI event loop
```

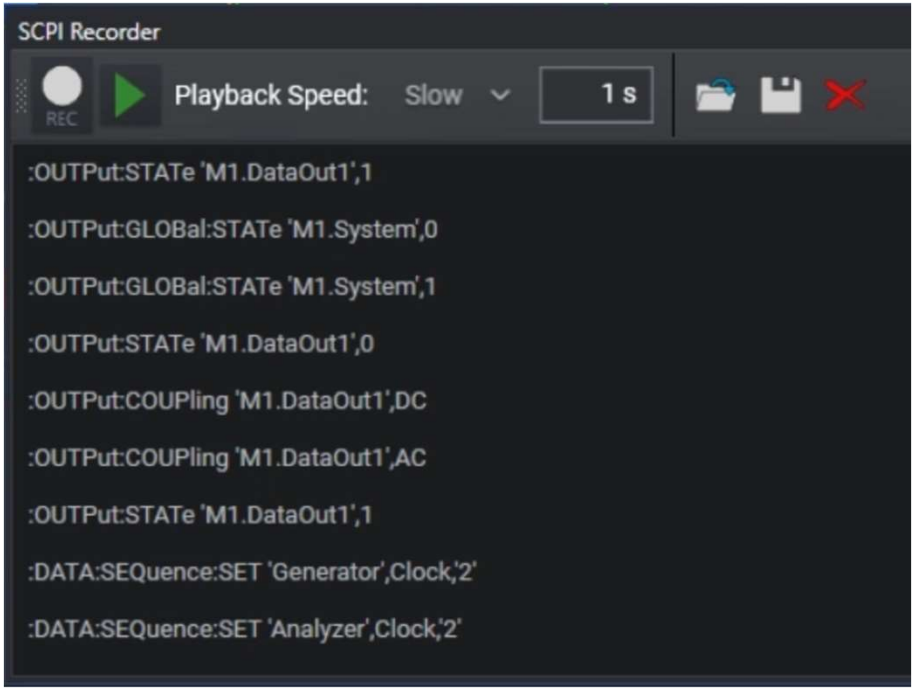
These elements create a window, embed a live plot, and add a button to trigger captures and update the plot.

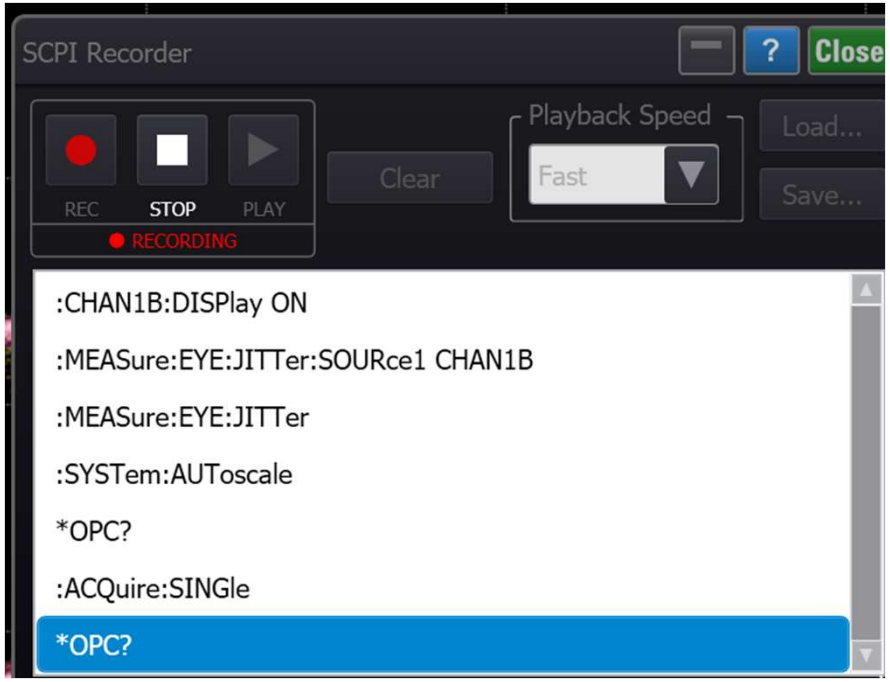
Writing and testing code without an Instrument



Instruments with SCPI Recorders (takes pain out of looking things up)

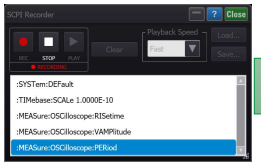
Many Keysight Instruments have SCPI recorders, M8070B and FlexDCA shown below





Coding in offline Mode (When no instrument is available)

Using SCPI recorder and AI to quickly Generate code



Run Offline mode, Walk through Steps

Obtain Address from Flex DCA

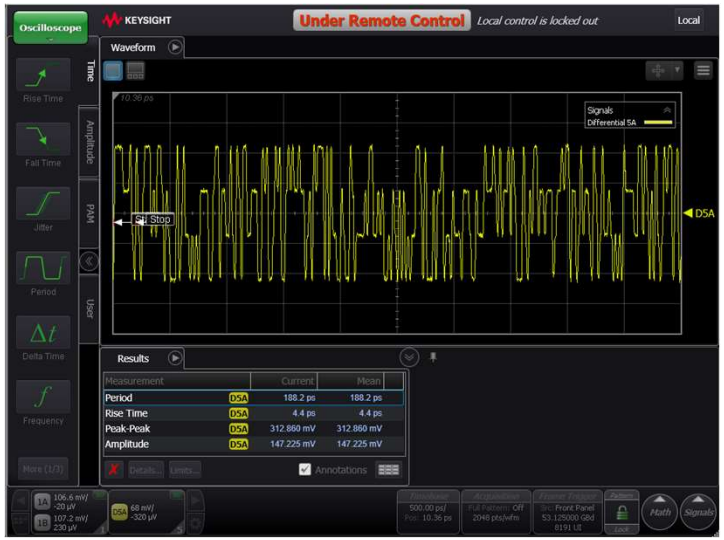
Use SCPI Recorder

Prompt AI to convert SCPI to Python

Python Code

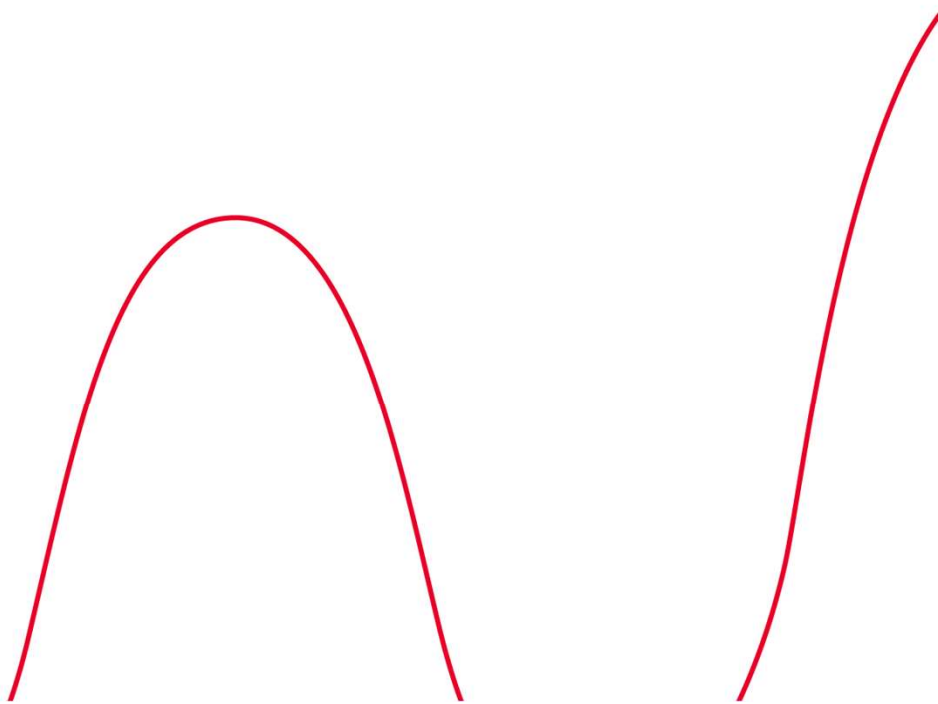
Offline Testing

When you have no physical instrument, You can still run code (Lets run generated code on Flex DCA Offline)



SCPI Command	Python Implementation
`:SYSTem:DEFault`	flexdca.write(":SYSTem:DEFault")
`:DISK:SETup:RECall "%USER_DATA_DIR%\Setups\HSDSDEMO7flexdca_complianceappSetup_2025-05-28_1.setx"`	flexdca.write(':DISK:SETup:RECall "%USER_DATA_DIR%\\Setups\\HSDSDEMO7flexdca_complianceappSetup_2025-05-28_1.setx"')
`*OPC?`	flexdca.query("*OPC?")
`:SYSTem:MODE OSCilloscope`	flexdca.write(":SYSTem:MODE OSCilloscope")
`:HISTogram1:DISPlay OFF`	flexdca.write(":HISTogram1:DISPlay OFF")
`:TIMebase:SCALe 2.0000E-10`	flexdca.write(":TIMebase:SCALe 2.0000E-10")
`:SYSTem:AUTOscale`	flexdca.write(":SYSTem:AUTOscale")
`*OPC?`	flexdca.query("*OPC?")
`:TIMebase:SCALe 1.0000E-10`	flexdca.write(":TIMebase:SCALe 1.0000E-10")
`:MEASure:OSCilloscope:VAMPlitude`	flexdca.write(":MEASure:OSCilloscope:VAMPlitude")
`:MEASure:OSCilloscope:VAMPlitude?`	flexdca.query(":MEASure:OSCilloscope:VAMPlitude?")
`:MEASure:OSCilloscope:VPP`	flexdca.write(":MEASure:OSCilloscope:VPP")
`:MEASure:OSCilloscope:VPP?`	flexdca.query(":MEASure:OSCilloscope:VPP?")
`:MEASure:OSCilloscope:RISetime`	flexdca.write(":MEASure:OSCilloscope:RISetime")
`:MEASure:OSCilloscope:RISetime?`	flexdca.query(":MEASure:OSCilloscope:RISetime?")
`:MEASure:OSCilloscope:PERiod`	flexdca.write(":MEASure:OSCilloscope:PERiod")
`:MEASure:OSCilloscope:PERiod?`	flexdca.query(":MEASure:OSCilloscope:PERiod?")
`:MEASure:ANNotations:STATe ON`	flexdca.write(":MEASure:ANNotations:STATe ON")
`:SYSTem:AUTOscale`	flexdca.write(":SYSTem:AUTOscale")
`*OPC?`	flexdca.query("*OPC?")
`:TIMebase:SCALe 5.0000E-10`	flexdca.write(":TIMebase:SCALe 5.0000E-10")
`:ACQuire:SINGle`	flexdca.write(":ACQuire:SINGle")
`*OPC?`	flexdca.query("*OPC?")

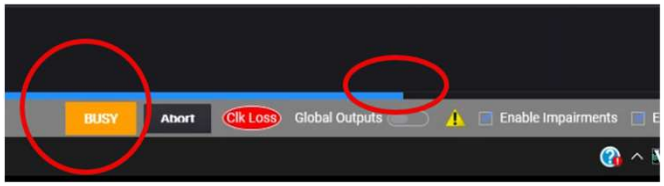
Some instruments have specific commands for checking readiness



The M8040 and similar Bert series has some unique challenges with Timing

Some operations take a long time to execute, if not handled correctly can be extremely frustrating

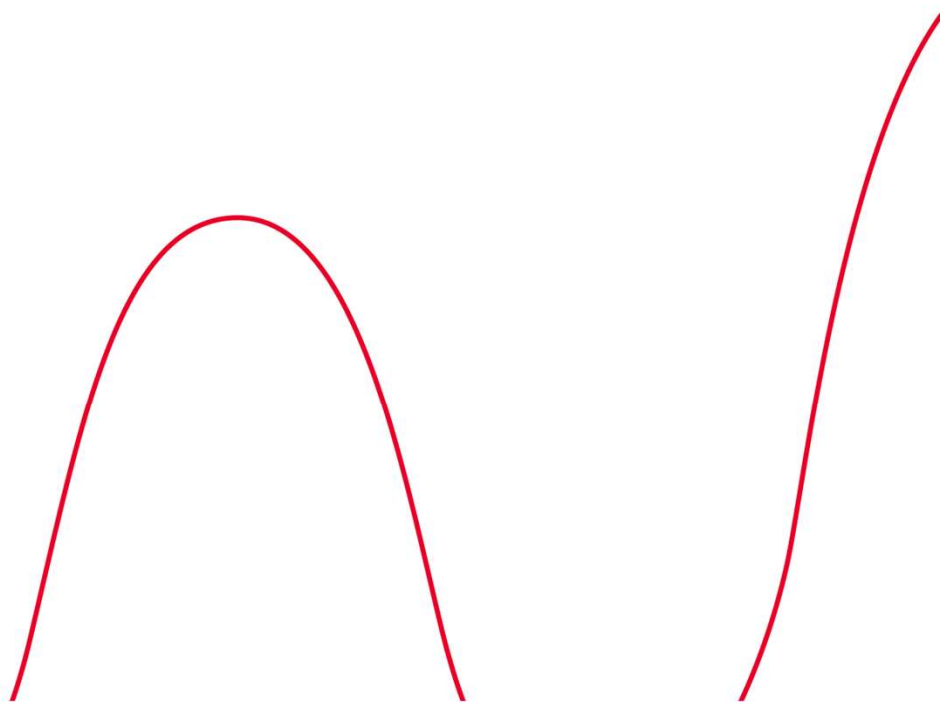
- *RST (reset or hard reset)
- Changing Clock Speeds
- Interference sources changing values



- It is imperative to have a reliable method to know when the BERT is finished setting up and is ready.
- OPC? Operation Complete works well as a blocking statement for Scopes but not here.
- The best to way to deal with this is to poll for Dataout ready.
- This is an Instrument Specific recommendation

```
while True:
        status = int(instr.query(':STATus:INSTrument:RUN? "M1.DataOut1"').strip())
        print(f"Polling DataOut1 ready status: {status}")
        if status == 1:
            print("Instrument is ready.")
            break
```

Automating 2 instruments at the same time



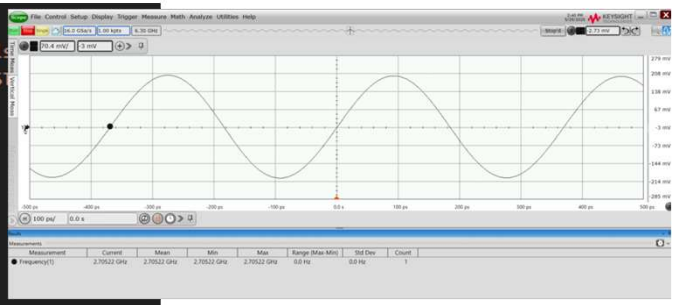
Adding a second instrument to your automation

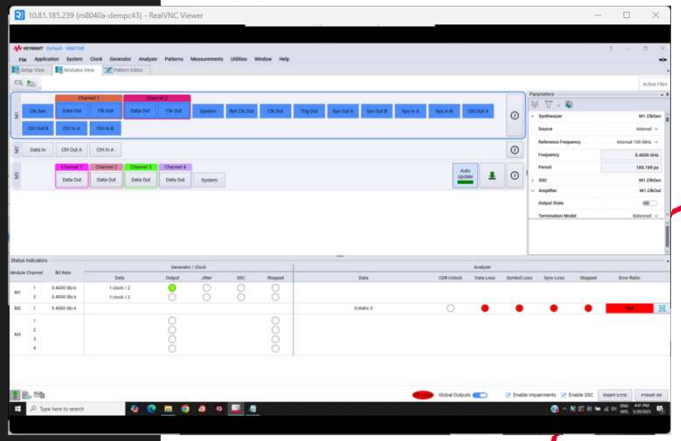
Defining our second program:

- We will Use M8040 and set 1 frequency in the M8070A Software
- Set default timeout to 10 seconds
- *IDN? Sanity check and also do this for the M8040A
- Report instruments installed options *OPT?
- Default Setup *RST
- Setup first frequency M8040A
- Autoscale
- Measure Frequency, Report it
- Change clock frequency on M8040A
- Autoscale
- Measure New Frequency, Report it
- Export sequence as Python Code
- Execute from Python IDE

A quick Look: Automating 2 Instruments MXR , M8040A, Note each unique resource name

```
rm = visa.ResourceManager()
MXR608B_Infiniium = rm.open_resource('TCPIP0::10.81.185.208::his
M8070B = rm.open_resource('TCPIP0::m8040a-dempc43::hislip0::INST
MXR608B_Infiniium.timeout = 20000
M8070B.timeout = 40000

idn = M8070B.query('*IDN?')
M8070B.write(":SOURce:FREQuency 'M1.ClkGen',5200000000")
wait_for_dataout1_ready(M8070B)
M8070B.write(":SOURce:VOLTage:AMPLitude 'M1.DataOut1',0.32")
M8070B.write(":DATA:SEQuence:SET 'Generator',Clock,'2' ")
M8070B.write(":OUTPut:COUPling 'M1.DataOut1',AC")
M8070B.write(":OUTPut:STATe 'M1.DataOut1',1")
M8070B.write(":OUTPut:GLOBal:STATe 'M1.System',1")
temp_values = M8070B.query(":SOURce:FREQuency? 'M1.ClkGen'")

idn1 = MXR608B_Infiniium.query('*IDN?')
MXR608B_Infiniium.write('*RST')
MXR608B_Infiniium.write(':CHANnel1:INPut DC50')
MXR608B_Infiniium.write(':AUToscale')
MXR608B_Infiniium.write(':SINGle')
MXR608B_Infiniium.write(':MEASure:Frequency CHANnel1')
MXR608B_Infiniium.write(':MARKer:MODE MEASurement')
results = MXR608B_Infiniium.query(':MEASure:RESults?')
first_value = results.split(',')[1].strip()

M8070B.write(":SOURce:FREQuency 'M1.ClkGen',5400000000")
wait_for_dataout1_ready(M8070B)
MXR608B_Infiniium.write(':SINGle')
MXR608B_Infiniium.write(':MEASure:Frequency CHANnel1')
results = MXR608B_Infiniium.query(':MEASure:RESults?')
second_value = results.split(',')[1].strip()

MXR608B_Infiniium.close()
M8070B.close()
rm.close()
```





```
Resource manager created.
Connected to Infiniium Oscilloscope.
Connected to M8070B BERT.
Timeouts set for both instruments.
M8070B IDN: Keysight Technologies,M8070B,MY62701793,11.0.150.12
Setting BERT clock frequency and waiting for outputs to be ready...
Polling DataOut1 ready status: 0
Polling DataOut1 ready status: 0
Polling DataOut1 ready status: 0
Polling DataOut1 ready status: 1
Instrument is ready.
Setting BERT output voltage amplitude...
Setting BERT data sequence...
Setting BERT output coupling to AC...
Enabling BERT output...
Enabling global output on BERT...
Querying BERT clock frequency...
Bert CLock Frequency set to: 5.20000000000000E+09

Querying oscilloscope IDN...
Connected to Infiniium Oscilloscope: KEYSIGHT TECHNOLOGIES,MXR608B,MY6406016
,11.62.00003
Resetting oscilloscope...
Setting oscilloscope channel 1 input to DC50...
Autoscaling oscilloscope...
Setting oscilloscope to single acquisition mode...
Setting up frequency measurement on channel 1...
Setting marker mode to measurement...
Querying oscilloscope measurement results...

###### Signal frequency measured by Infiniium: 2.60306E+09
--- Setting BERT frequency to 5.2 GHz and measuring again ---
Polling DataOut1 ready status: 0
Polling DataOut1 ready status: 0
Polling DataOut1 ready status: 0
Polling DataOut1 ready status: 0
Polling DataOut1 ready status: 0
Polling DataOut1 ready status: 1
Instrument is ready.
###### Signal frequency measured by Infiniium after increase: 2.69619E+09
Closing oscilloscope connection...
Closing BERT connection...
Closing VISA resource manager...
PS C:\Users\fairfiel\OneDrive - Keysight Technologies\LNL\hsd 2025\Python Pro
gramming\example code>
```

Best Practices

Always start with *RST from a know condition

Always sanity check with *IDN?

Make sure you understand the sequence of the measurement

Don’t rely on Autoscale functions, instead, set scales manually for repeatability

Make sure your programs live in User space in the OS otherwise security may block execution

Repurposing existing code

- If you have an automation setup for one Vendor’s Instrument that is of the same type as your instrument, , you can re-use a lot of the commands.
- You may have to make substitutions for Vendor specific ‘Custom” instrument commands.

Be aware that in a busy lab , LAN may slow down with multiple instruments

“But it worked last week”

- Some automation Can Move a lot of data and you may get added latency
- This becomes more of an issue when you haven’t considered synchronization Strategy between the program and instruments.
- A combination of localization and synchronization needs to be considered or you may run into software errors related to delay that world otherwise not show up in an isolated setup
- Not all your issues will be exposed in initial coding, may appear months later if you didn’t consider latency issues that are dynamic

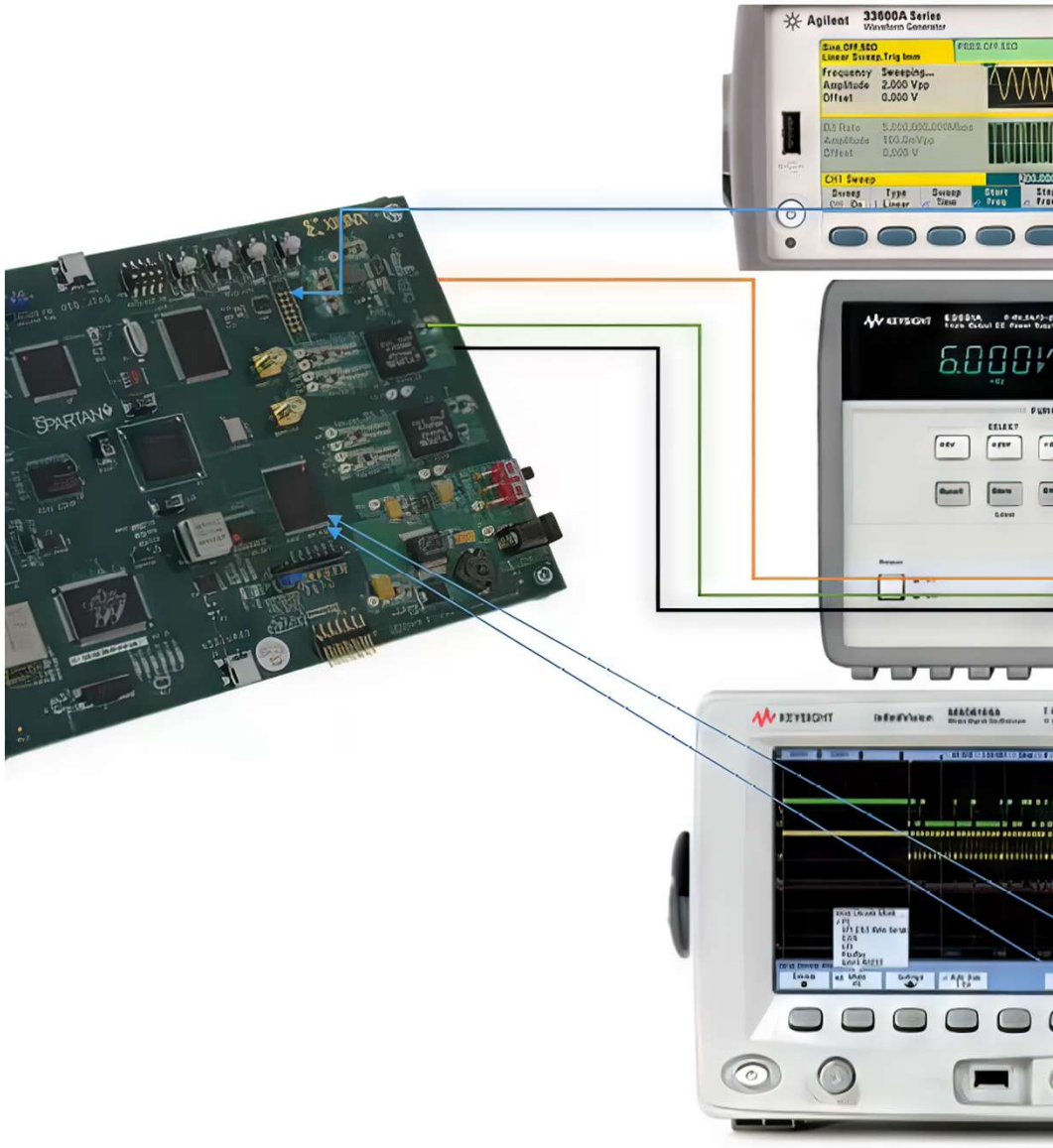
Summary: Think Like you are at the Bench!!

- Automation = Manual Process, Scripted
  You're replicating what you'd normally do by hand—just in code.

- Plan the Sequence
  Set up instruments in the right order:
    - Arm the oscilloscope
    - Prepare the DMM (if used)
    - Set voltage/current limits on the power supply
    - Power on the supply

- Synchronize Actions
  Timing matters—just like at the bench, everything must be ready before powering up.
    - You may have to wait until an instrument has executed a command (Autoscale may take a few seconds to complete)

- Understand Before You Automate
  Know the manual process well—it's the foundation for reliable automation.



KEYSIGHT

Summary Where to get Code , commands and Snippets

- Command expert
- Programmers reference for the product
- Example code
- Scpi recorders if the instrument supports it
- Porting code found on the web.

The current state of AI is powerful, but flawed. We are hitting a "Data Wall" and an "Abstraction Wall." Instead of waiting for a distant superintelligence (AGI), we must use today's AI as an Efficiency Multiplier.

AI's Core Limitation	The Problem	Your Essential Role
1. The Statistical Limit	Current models excel at **mimicry** (predicting the next word) but lack true **judgment** or the ability to handle completely **novel** problems	**Provide Judgment & Strategy** You set the goal, define the 'why,' and apply human principles (taste, ethics) to the generated output.
2. The Inconsistency Flaw	AI can be highly productive but is prone to unpredictable **inconsistency** (e.g., getting stuck in bug loops) and cannot reliably bridge the gap to the physical world	**Be the Quality Controller** You must verify, fact-check, and apply the final, critical accuracy review to ensure output is reliable and feasible.
3. The Efficiency Gap	Lacking the human **'value function'** (emotion/internal compass), AI must inefficiently calculate every option, leading to decision paralysis without human input	**Define Value & Priority** You must provide the emotional context and priority to guide the pure logic toward the *most useful* and efficient path.

This is Paraphrased from recent Youtube interviews with Geoffrey Hinton, **Ilya Sutskever and Alex Krizhevsy** about limitations where all the data in the world is not producing overwhelming results in business etc, not until we stop using LLMs

- **Takeaway:** AI handles the statistical *drudgery*. You provide the **Judgment, Quality, and Purpose.**

Vibe coding VS Planned Coding

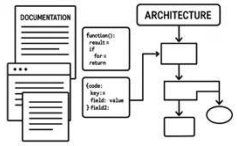
Best workflow: *Vibe to explore → Plan to stabilize / Refactor for reuse*

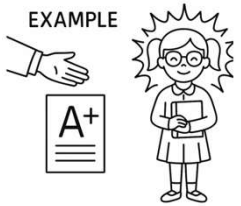
Aspect	**Vibe Coding**	**Planned Coding / Prompt Engineering**
Goal	Explore & get something working fast	Build correct, repeatable, scalable code
Speed to first result	⭐⭐⭐⭐⭐ Very fast	⭐⭐ Slower startup
Structure	Emerges organically (or not at all)	Defined upfront (functions, flow, contracts)
Memory / state assumptions	Often implicit	Explicit and documented
LLM interaction	Short, reactive prompts	Precise, constrained prompts
Scalability	Breaks down as code grows	Scales to large scripts/projects
Maintainability	Harder to debug & extend	Easier to read, test, and reuse
Best use cases	Demos, notebooks, learning, exploration	Production scripts, libraries, teaching
Risk	Hidden bugs, fragile logic	Over-design, slower iteration

Using AI to Help you :My Prompt Engineering Workflow for better success

Watching and Learning on youtube from different points of view I saw a common workflow being used

- **A Formal Workflow for Technical Work**
  1. **Define the Objective** — measurable, simple, testable
  2. **Provide Context** — documents/ user guides, code samples, architecture
  3. **Set Boundaries/Constraints** — no invention from the bot, tell it to ask questions if unclear
  4. **Define Success Rubric** — provide the context of what you consider success for the goal and have the AI evaluate itself Ask for Test Vectors to be run
  5. **Iterate to Completion** — refine until rubric passes 90%
- **Key insight:**
  Create meta prompts, chat with AI to create better prompts, have it ask you questions if some parameters are unclear.
- **Sandbox the Environment** — isolated run-space with logs, this provides a safe environment space for the AI to automatically run commands and interate the results (it can read the terminal responses)





Meta prompts: Using prompts to create better prompts

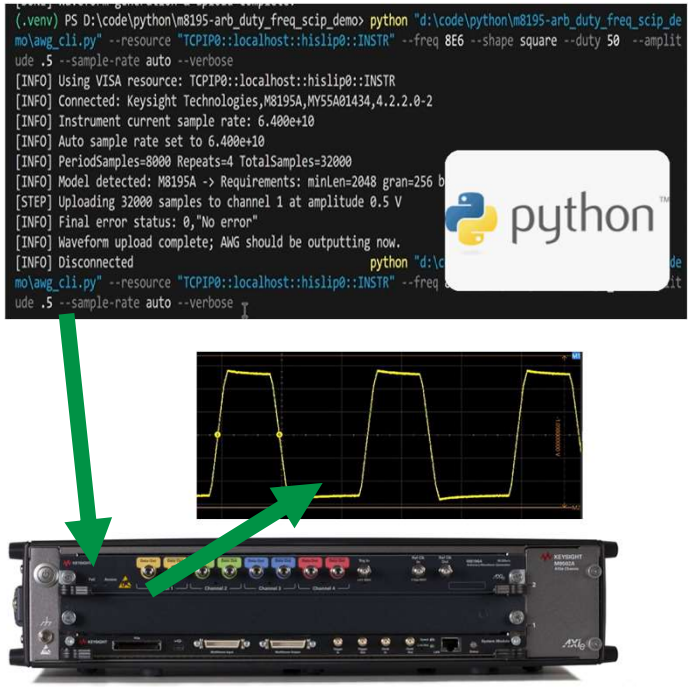
Example Prompt phrases to include which really have an impact

- ““You are a senior software engineer and data scientist. Write clean, production-quality Python using industry best practices (readable structure, clear naming, error handling, and logging). Prefer simple, maintainable solutions over clever ones.””
- “You are a meticulous programmer-data scientist. For every function you write, also generate unit tests and test vectors, including edge cases and failure scenarios. Explicitly show the expected inputs and outputs.”
- “You Are Alan Copeland..’nuff said

Applied Workflow: AI-Generated Python Control of M8196A

Customer wanted Python parameter-based signal generation, for the M8196A

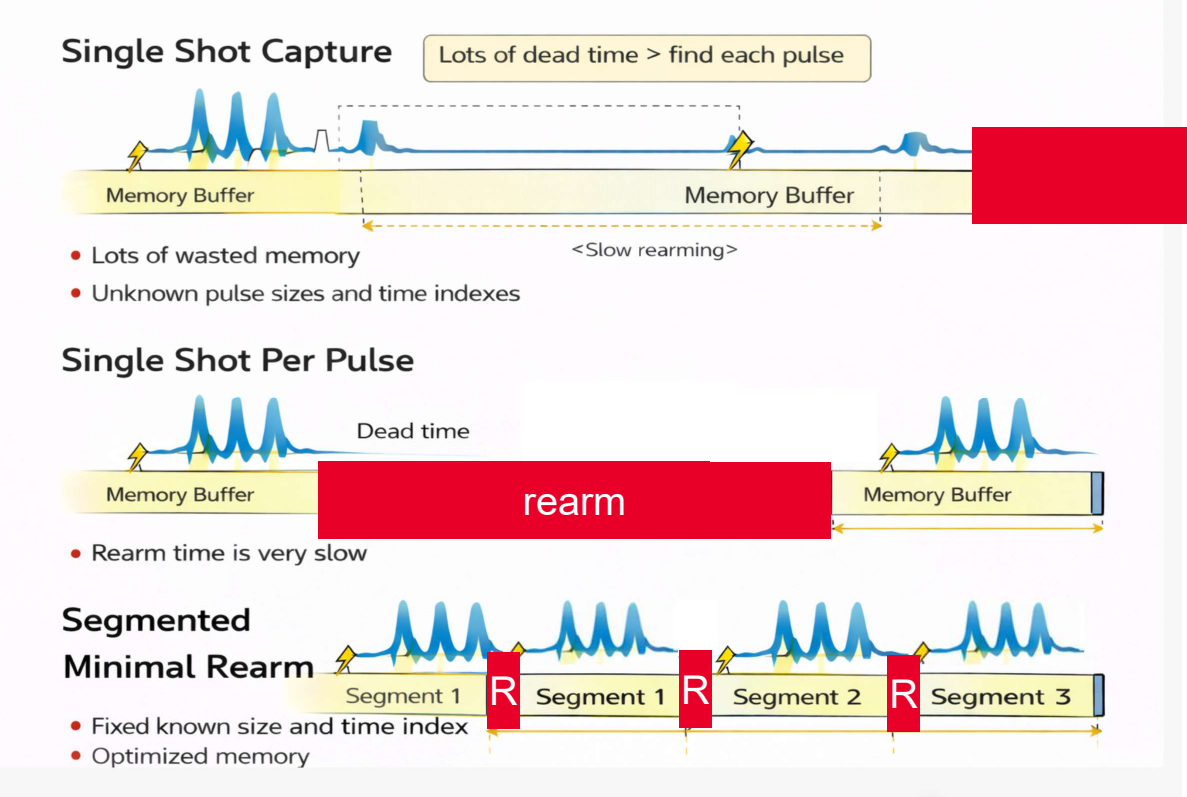
- **Customer challenge**
- Needed a **simple Python interface** for freq, duty cycle, sweeps, and multi-arb control.



Define Objective	**simple Python interface** for freq, duty cycle, sweeps, and multi-arb control, the key is memory calculation and fitting pattern memory boundaries exactly as to not cause spectral issues …
Provide Context	-M8195 Programmers reference -M8195 Users manual -Iqtools m files -Example python code
Set Contsraints	Create a VENV, and use a sandbox allow autonomous actions in this area only, min max frequency generation, min max duty cycle and steps
Define Success with a grade	check for crash, recovery app failures automatically then final quality: human pass/fail checks when prompted by AI
Define tests and Iterate until grade is met	Write test vectors for all channels, sweep multiple duty cycle values, figure out load time issues, various instrument starting states (including a parallel program changing values),

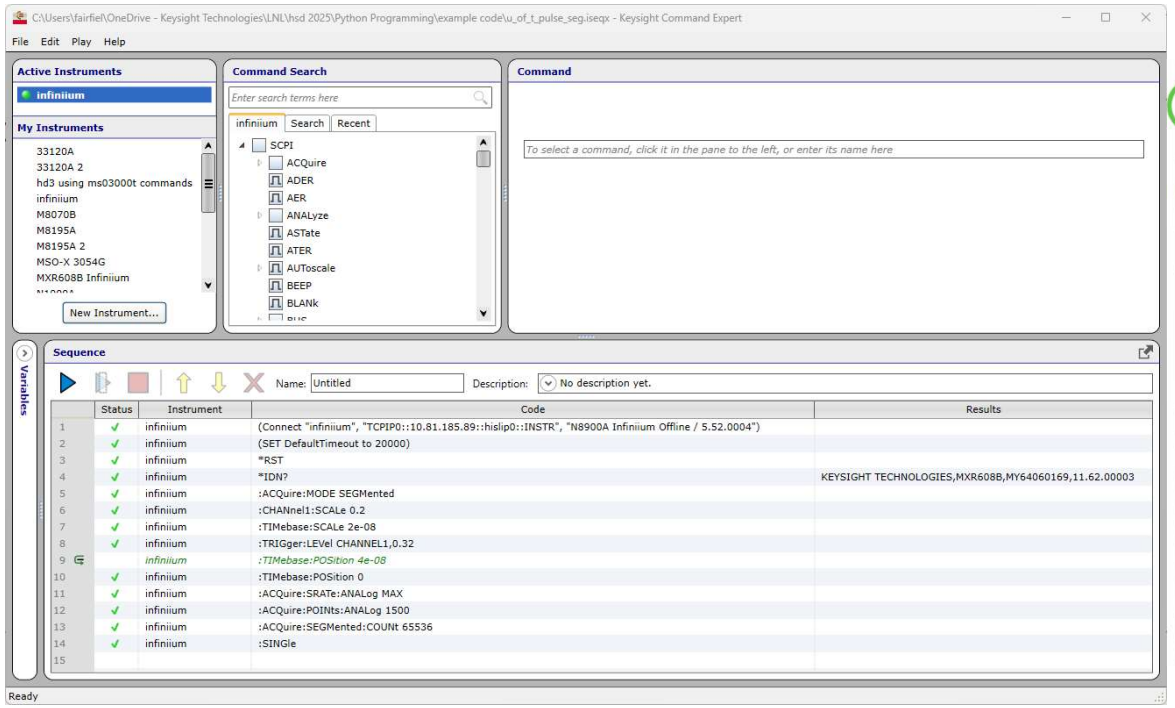
Segmented memory mode Advantages

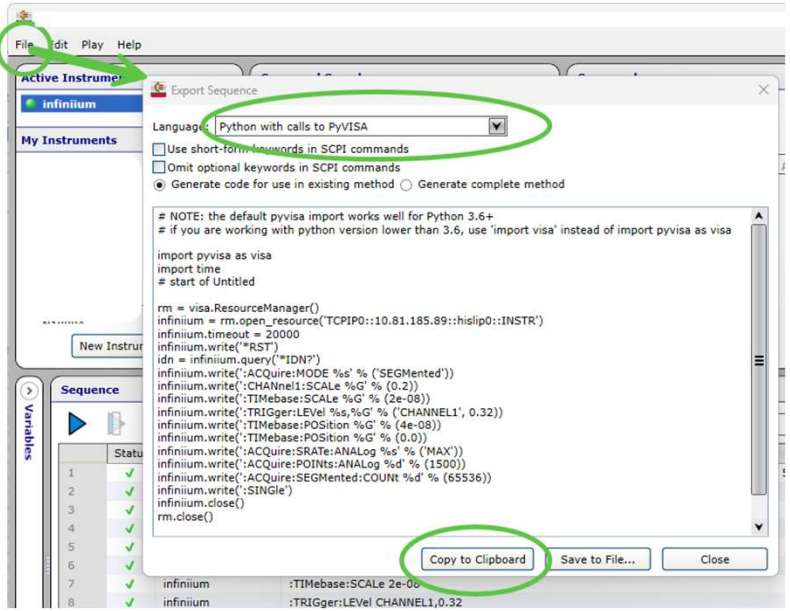
Single shot capture least optimized but does capture “everything” for that period



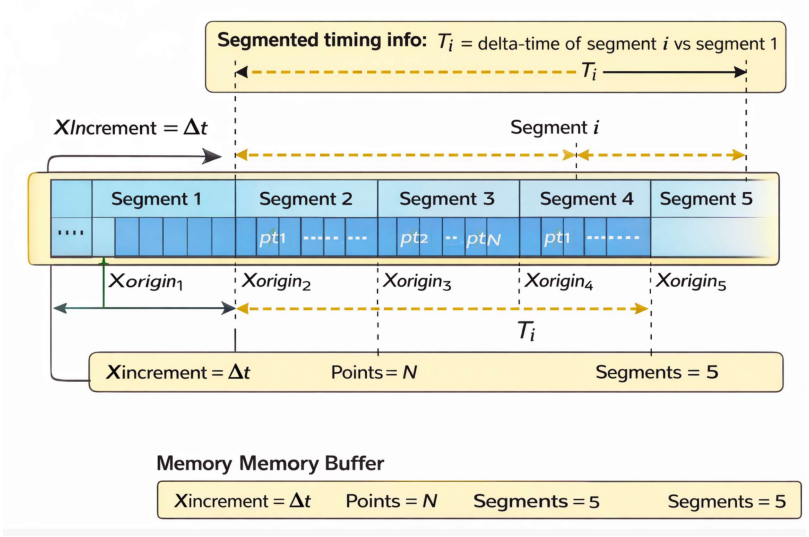
Simple shot pulse capture and waveform data transfer python demo

Start with Command Expert Sequence, Its simple and allows realtime checking then export





Two transfer modes: “one-at-a-time” vs “all-at-once”



- **Mode A: One segment at a time (recommended)**
- Loop: set INDex :WAVeform:DATA? :WAVeform:SEGMented:TTAG?
- Pros: memory-safe for huge segment counts, easy to stream/process
- Cons: more round trips

- **Mode B: Bulk transfer (all segments concatenated)**
- Enable :WAVeform:SEGMented:ALL ON then :WAVeform:DATA?
- Data comes back as [seg1][seg2]...[segN] with **fixed bytes per segment**
- Pros: fewer transactions, fast for small N
- Cons: can be massive; must compute segment boundaries using points & bytes/point

- **Best practice**
- Always set waveform source/format explicitly :WAVeform:SOURce :WAVeform:FORMat :WAVeform:BYTeorder,
- Always disable ALL mode afterward :WAVeform:SEGMented:ALL OFF,

Segment Viewer

list the scpi commands used

✓ Identified SCPI commands for oscilloscope

Here are all the SCPI commands used in the segment viewer:

Instrument Identification:

- `*IDN?` - Query instrument ID (manufacturer, model, serial)

Waveform Configuration:

- `:WAVeform:SOURce CHANnel1` - Set channel to read from
- `:WAVeform:FORMat WORD` - Set data format (16-bit integers)
- `:WAVeform:BYTeorder LSBF` - Set byte order (Least Significant Byte First)

Query Segment Information:

- `:WAVeform:SEGMented:COUNt?` - Get total number of captured segments
- `:WAVeform:XINCrement?` - Get time increment between samples (seconds)

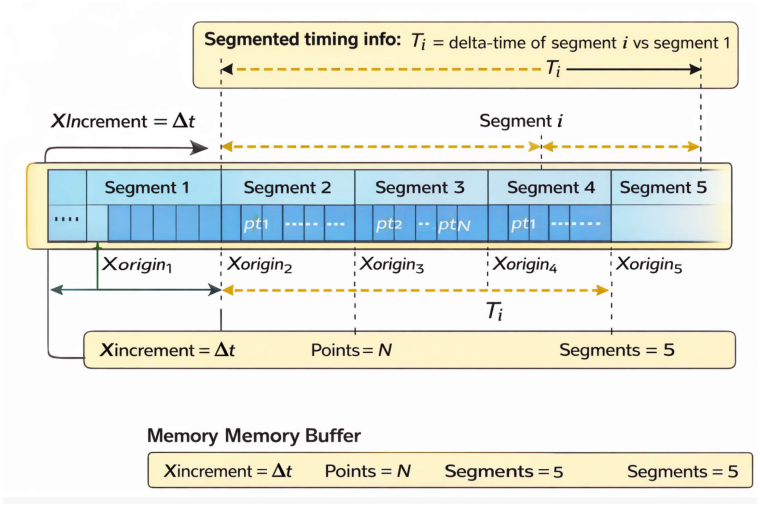
Segment Data Acquisition:

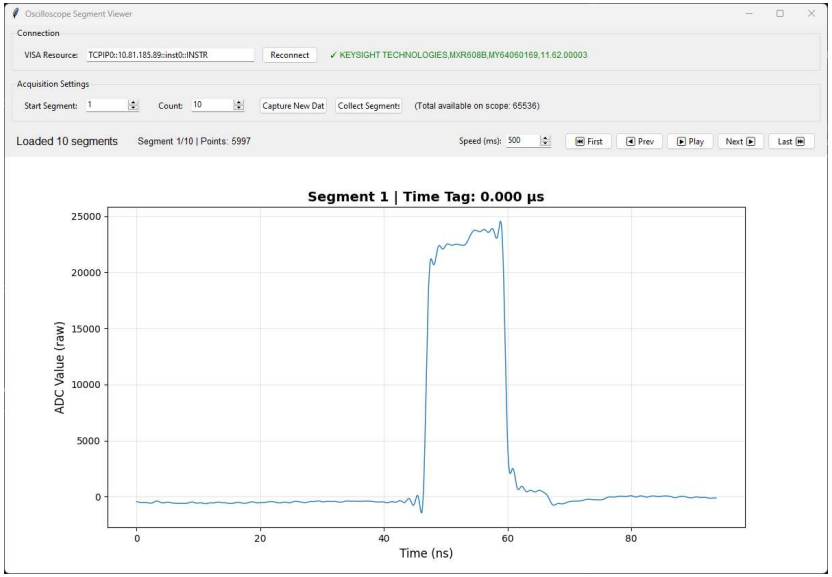
- `:ACQuire:SEGMented:INDex <n>` - Select segment number to read
- `:WAVeform:DATA?` - Read waveform data (returns IEEE 488.2 binary block)
- `:WAVeform:SEGMented:TTAG?` - Get time tag for current segment (seconds)

Data Flow per Segment:

1. Set segment index → `:ACQuire:SEGMented:INDex 1`
2. Request data → `:WAVeform:DATA?`
3. Get time tag → `:WAVeform:SEGMented:TTAG?`
4. Repeat for each segment

Claude Sonnet 4.5 • 1x





Making the app User Friendly Using AI

create a new gui app, add the scope setup logic from the super_simple app_ to the segment viewer gui app and in a setup section have the variables for the setup parameters, allow to change sample rate, segment size acquire points, segment count , channel scale, timebase scale , trigger level , timebase position

super_simple_pulse_from_command_expert.py segment_viewer_gui.py

Thinking...

- Sample rate
- Segment size/acquire points
- Segment count
- Channel scale
- Timebase scale
- Trigger level
- Timebase position

2. This should be a new file, combining the functionality

Let me create a comprehensive GUI that has both setup and viewing capabilities.

Created a new comprehensive GUI app! Run it with:

```
python scope_setup_and_viewer.py
```

Perfect! The new GUI is running with all setup parameters:

Scope Setup Parameters (left column):

- Channel 1 Scale (V/div) - Default: 0.2
- Timebase Scale (s/div) - Default: 2e-8
- Trigger Level (V) - Default: 0.32
- Timebase Position (s) - Default: 0.0

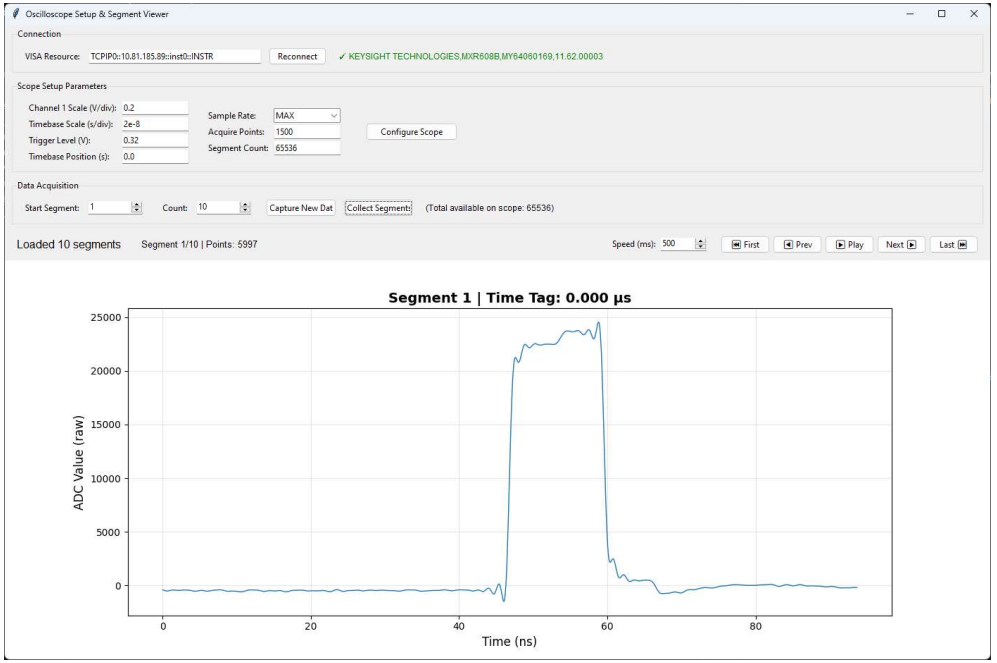
Scope Setup Parameters (right column):

- Sample Rate - Dropdown: MAX, 10e9, 5e9, 2.5e9, 1e9, 500e6
- Acquire Points - Default: 1500
- Segment Count - Default: 65536

Workflow:

1. **Connect** to scope
2. Adjust setup parameters as needed
3. **Configure Scope** - applies all settings with *RST
4. **Capture New Data** - triggers :SINGle
5. **Collect Segments** - downloads data
6. Use playback controls to view

All parameters are editable before configuration!



Resources used

- Code examples:
- https://github.com/fairdogJR/Segmented-memory-Automation-with-python-infiniium-scopes-

- Command Expert
- https://www.keysight.com/us/en/lib/software-detail/computer-software/command-expert-downloads-2151326.html
- Introduction to Python Automation (circa 2016 Python 2.7)
- https://www.youtube.com/playlist?list=PLMFn2UaPBVfWVjCbQh_DsJknlwpVkilhh

KEYSIGHT