

Instrument Control and Automation with Python

Easily control instruments and create test sequences with Python in a hybrid learning environment with Digital Learning Suite

Introduction

Test automation is heavily used in the R&D and production environment in the test and measurement industry. In education, educators and students use test automation to characterize electronic designs, completing group assignments or research projects. Without the support from an easy-to-use development environment, scripting test automation is challenging especially when it involves the integration of multiple remote instruments and DUT controls, interactive user interfaces, advanced data processing and charting. Often, students also need access to lab equipment out of lab hours to continue with their lab work. It is crucial to have a test automation environment that can be accessed anywhere and anytime, equipped with a remote lab management and scheduler.

Keysight enables innovators to push the boundaries of engineering by quickly solving design, emulation, and test challenges to create the best product experiences. Keysight is committed to ensure that students in the universities are exposed to the latest tools used in the industry. PathWave Test Automation (powered by OpenTAP) is the core automation engine powering many Keysight solutions in automating measurements in fields such as 4G, 5G, Automotive, Cybersecurity, etc. PathWave Test Automation OpenTAP engine is now built into the new web-based Digital Learning Suite (DLS) to ease the use of test automation for students and educators in a hybrid learning environment.

Python is a common programming language in the engineering education for device controls and data analysis. It is widely used in instrumentation and controls, data analysis, control systems, signal processing, statistical analysis, etc. With OpenTAP and DLS, professors can continue leveraging existing knowledge and code written in Python and have the frontend test automation handled entirely by DLS. The DLS can be accessed from a web browser which offers the greatest flexibility and mobility to the educators and students. The OpenTAP Python plugin makes it possible to use Python to program plugins for OpenTAP that can eventually be accessed from a browser with DLS. This application note will explain how you can easily construct a test automation with Python, controlling an instrument and performing data analysis with OpenTAP and DLS.

Setting Up the Environment

We will first set up the environment by installing a few tools and plugins. Then, we will go through two different ways of using Python in DLS. The first one will see a creation of a test sequence consisting of an instrument driver and control test step. The second method uses Python to do post data analysis on results coming from built-in instrument control test steps in DLS.

1. Download Python from <https://www.python.org/>
2. Install Keysight IO Libraries Suite, if not already installed from <https://www.keysight.com/find/iolibraries>
3. Visit OpenTAP Public Package Repository. <https://opentap.io>.
 - a. Search for "OpenTAP", select Windows for OS and version 9.20.4+c2191172 or above. Download the package. Once downloaded, use any archiver to extract the package (it is a zipped file) to your local folder. You can place the folder anywhere and the location of this folder is your OpenTAP installation directory. Launch a command prompt window in elevated mode and change directory to where OpenTAP package is extracted to.
 - b. Use a command prompt and run the following command lines inside the OpenTAP installation directory:
 - i. `tap package install OpenTAP`
 - ii. `tap package install Python`
 - iii. `tap package install REST-API` - Requires KS8400EDU license
 - iv. `tap package install "Keysight Licensing"`
 - v. `tap package install "KeyLab Test Sequencing and Control"` - Requires KS8400EDU license
 - c. Now you are ready to create an OpenTAP plugin with Python

Next, you can start developing Python plugin projects in two different ways. You will find more information about Python plugin creation here <https://doc.opentap.io/OpenTap.Python>

1. Develop the plugin in an isolated 'project' folder. This is recommended for bigger projects.
2. Develop the plugin inside the 'Packages' folder. This way it's easy to get started and make something quick, but maintaining the project becomes harder in the long run

In this app note, we will show you the second method by developing the code in a folder within %TAP_PATH%\Packages. %TAP_PATH% is your OpenTAP installation directory. To get started quickly, you can also copy all files in <https://www.keysight.com/us/en/lib/software-detail/programming-examples/sr101edua-digital-learning-suite-python-code-example.html> and paste them in your %TAP_PATH%\Packages\PythonExampleForDLS. This effectively creates a Python plugin called "PythonExampleForDLS". The plugin contains many examples of controlling Keysight and non-Keysight devices. The Python codes presented in the next sections can be found in this plugin folder.

To get started, we will use Keysight InfiniiVision Oscilloscope as an example. We will create an instrument driver with Python that contains the controls such as "Identify Instrument", "Auto Scale", "Measure Frequency", "Screenshot" and "Get Trace Data". We will also show you how you can easily publish results and transfer data between test steps using the custom "DLStep" class. Lastly, we will show you can quickly connect your plugin to DLS, running the test steps, plotting the data.

Oscilloscope Instrument Driver

OscilloscopeInstrument.py

```
from System import Double, String, Byte, Int32
from opentap import *
import OpenTap
from OpenTap import Log
import numpy as np
```

Import these modules to access the OpenTAP API that contains .NET types and OpenTAP classes. Numpy library can be used to ease data manipulation

```
@attribute(OpenTap.Display(Name="Oscilloscope", Groups= ["DLS Python Plugin"]))
class OscilloscopeScpiInstrument(OpenTap.ScpInstrument):
    def __init__(self):
        super(OscilloscopeScpiInstrument, self).__init__()
        self.Name = "Oscilloscope"
        self.IoTimeout = 10000
```

Create an oscilloscope instrument class that users can add as an instrument. SCPI commands are used to control the oscilloscope. Attributes (Name, Groups, etc) can be added to aid users when selecting the instrument

```
def GetIdnString(self):
    return self.ScpQuery[String]("*IDN?")

def AutoScale(self):
    return self.ScpQuery[String](":AUTOSCALE;*OPC?")

def MeasureFrequency(self, channel):
    return self.ScpQuery[String]("MEASURE:FREQUENCY? CHANNEL" + str(channel))

def GetScreenshot(self):
    data = self.ScpQueryBlock(":DISPlay:DATA? BMP,COLOR")
    print(":DISPlay:DATA? BMP,COLOR")
    return data

def GetTraceData(self, channel):
    self.ScpCommand("WAVEFORM:SOURCE CHANNEL" + str(channel))
    self.ScpCommand("WAVEFORM:POINTS:MODE MAXIMUM")
    self.ScpCommand("WAVEform:FORMat BYTE")
    preamble = self.ScpQuery[String](":WAVEFORM:PREAmble?").split(",")
    t = int(preamble[1])
    fPoints = int(preamble[2])
    fCount = int(preamble[3])
    fXincrement = float(preamble[4])
    fXorigin = float(preamble[5])
    fXreference = float(preamble[6])
    fYincrement = float(preamble[7])
    fYorigin = float(preamble[8])
    fYreference = float(preamble[9])
    rawdata = self.ScpQueryBlock("WAVEform:DATA?")
    x = np.empty(len(rawdata), dtype=np.double)
    y = np.empty(len(rawdata), dtype=np.double)
    for idx, d in enumerate(rawdata):
        x[idx] = (fXorigin + ((float(idx) - fXreference) * fXincrement))
        y[idx] = ((d - fYreference) * fYincrement) + fYorigin
    return [x, y]
```

Now, let us define a few oscilloscope functions that can be used to identify the instrument headers, auto scale, measure frequency. Get a screenshot and trace data. ScpiCommand is used to send a SCPI command without a returned value. To get a returned value, we will use ScpiQuery[String] for string value while ScpiQueryBlock for binary value. In "GetTraceData" function, numpy is used to construct arrays that contain the returned waveform point values.

Oscilloscope Test Step

OscilloscopeSteps.py

```
import sys
import openTap
from openTap import *
import OpenTap
import math
from OpenTap import Log, EnabledIfAttribute
import System
from System import Array, Double, Byte, Int32, String, Boolean
from System.ComponentModel import BrowseableAttribute
import System.Xml
from System.Xml.Serialization import XmlIgnoreAttribute
from OscilloscopeInstrument import *
from DLSOutputInput import *
```

```
@attribute(OpenTap.Display(Name="Identify", Description="", Groups= ["DLS Python Plugin", "Oscilloscope"]))
class OscilloscopeIdentify(TestStep):
    ScpiInst = property(OscilloscopeSCPIInstrument, None).add_attribute(OpenTap.Display( "Oscilloscope"))

    def __init__(self):
        super(OscilloscopeIdentify,self).__init__()

    def Run(self):
        idn = self.ScpiInst.GetIdnString()
        self.log.Info("IDN: "+str(idn))

@attribute(OpenTap.Display(Name="Autoscale", Description="", Groups= ["DLS Python Plugin", "Oscilloscope"]))
class OscilloscopeAutoScale(TestStep):
    ScpiInst = property(OscilloscopeSCPIInstrument, None).add_attribute(OpenTap.Display( "Oscilloscope"))

    def __init__(self):
        super(OscilloscopeAutoScale,self).__init__()

    def Run(self):
        self.ScpiInst.AutoScale()
```

```
@attribute(OpenTap.Display(Name="Measure Frequency", Description="", Groups= ["DLS Python Plugin", "Oscilloscope"]))
class OscilloscopeMeasureFrequency(TestStep):
    ScpiInst = property(OscilloscopeSCPIInstrument, None).add_attribute(OpenTap.Display( "Oscilloscope"))
    Channel = property(Int32, 1).add_attribute(OpenTap.Display( "Channel"))

    def __init__(self):
        super(OscilloscopeMeasureFrequency,self).__init__()

    def Run(self):
        freq = self.ScpiInst.MeasureFrequency(self.Channel)
        self.log.Info("Frequency: "+str(freq))
        self.PublishResult("Measure Frequency", [{"Frequency"}, {freq}]
```

```
@attribute(OpenTap.Display(Name="Screenshot", Description="", Groups= ["DLS Python Plugin", "Oscilloscope"]))
class OscilloscopeGetScreenshot(DLSStep):
    ScpiInst = property(OscilloscopeSCPIInstrument, None).add_attribute(OpenTap.Display( "Oscilloscope"))

    def __init__(self):
        super(OscilloscopeGetScreenshot,self).__init__()

    def Run(self):
        y = self.ScpiInst.GetScreenshot()
        super().OutputToDLS("Screenshot", [{"Screenshot"}], [y], True)

@attribute(OpenTap.Display(Name="Trace Data", Description="", Groups= ["DLS Python Plugin", "Oscilloscope"]))
class OscilloscopeGetTraceData(DLSStep):
    ScpiInst = property(OscilloscopeSCPIInstrument, None).add_attribute(OpenTap.Display( "Oscilloscope"))
    Channel = property(Int32, 1).add_attribute(OpenTap.Display( "Channel"))

    def __init__(self):
        super(OscilloscopeGetTraceData,self).__init__()

    def Run(self):
        x,y = self.ScpiInst.GetTraceData(self.Channel)
        super().OutputToDLS("TraceData1", [{"X", "Y"}], [x,y])
```

Now, we will use the instrument driver “OscilloscopeSCPIInstrument” (from OscilloscopeInstrument.py) created above. In DLS environment, you can also use the DLSStep (from DLSOutputInput.py) class to simplify data publishing and transfer.

Next, we will create a test step named “Identify”. A property named “Oscilloscope” is created for user to select the instrument. This test step will run the “GetIdnString()” function from the instrument driver to obtain the instrument header and print on the log panel. Follow the same process to create a function named “Autoscale” to run “AutoScale()” function from the driver.

You can also add more properties for users to select. In this case, to measure frequency, user can select the channel to measure. Create a property named “Channel”. The returned data can be further processed if needed and be published to the OpenTAP result listener. The result can later be exported or plotted in DLS

For more complicated returned data type such as binary and array, we will use the “DLSStep” instead. Inherit “DLSStep” to use the “OutputToDLS” function. Data can also be further manipulated before outputting to DLS if needed. The “OutputToDLS” function will format the data into a recognizable format in DLS where you can view the screenshot image or the data array in a tabular form that can be easily plotted on the DLS charts.

Connecting Python to DLS

Now, we have the plugin ready to connect to DLS. Open a command prompt and change the directory to your OpenTAP installation directory. Type “tap remote sessionmanager” to start the local session for DLS. In DLS, navigate to Local Test Automation to start the session.

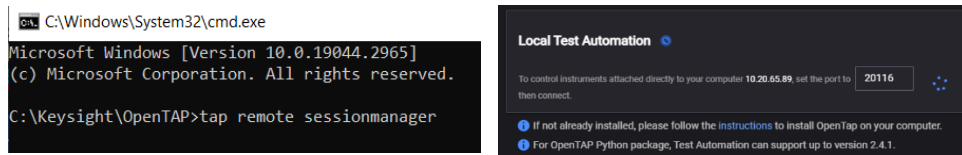


Figure 1. Launching DLS Test Automation

On the DLS Test Automation user interface, add a new “Oscilloscope” instrument under the “DLS Python Plugin” category and enter the instrument address. Go to Steps and expand the “DLS Python Plugin” to find all the test steps created in previous sections. Add and run all of them on the test plan.

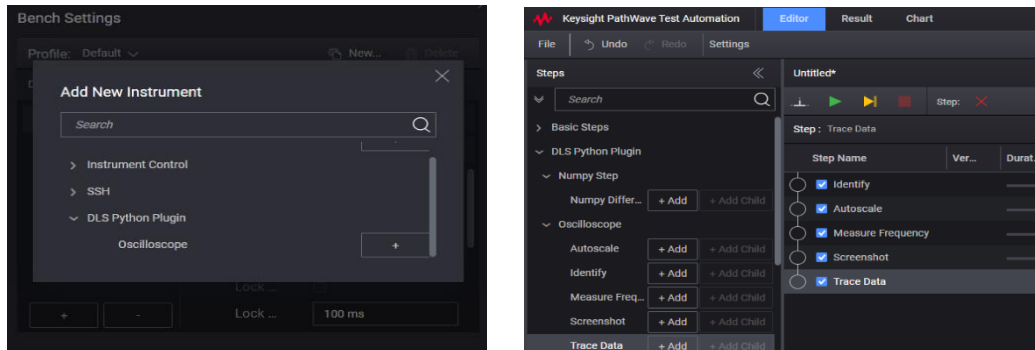


Figure 2. Creating a test plan and test steps

After completing the run, you can get the measured frequency value, screenshot image and trace data on the Result page. You can also plot the Trace Data on the Chart page by editing the X and Y text boxes.

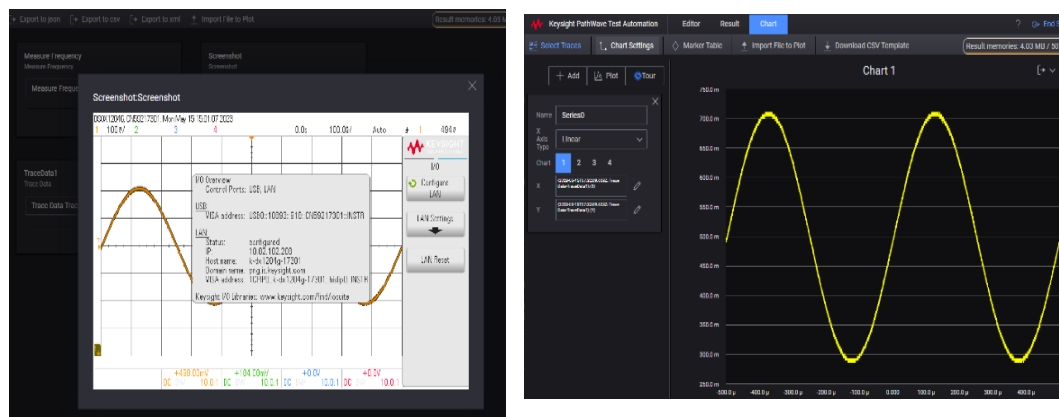


Figure 3. Viewing results

Data Transfer and Analysis

In DLS test automation, you can also obtain results from other test step to do post data analysis. DLS has hundreds of built-in test steps that you can readily use to control Keysight and non-Keysight instruments. With Python, you can easily get results from the built-in steps and perform further data analysis. Now, you can spend time focusing on creating data analysis algorithms and DLS will manage the control and data extraction from instruments.

Each test step in DLS has three additional properties: “ResultNamesList”, “ColumnNamesList” and “DataList”. These properties are used to store the returned data after each successful test step run. A test step can store one or more ResultNames with each ResultName containing one or more columns of data array stored in “DataList” where each array corresponds to a ColumnName. These properties can be renamed by users. “DLSSStep” comes with default properties for users to choose the data array based on the chosen InputTestStep, ResultName and ColumnName. The example below shows how you can obtain results from other built-in DLS test steps using the above properties, apply mathematical operation using NumPy (A powerful Python library used for numerical computing and scientific computing) and plot the data on DLS charts.

Data Transfer and Analysis with Numpy

NumpySteps.py

```
from System.Collections.Generic import List
from opentap import *
import OpenTap
```

```
from OpenTap import TestStep
```

```
import numpy as np
from scipy import stats
```

```
from .DLSOutputInput import *
```

```
@attribute(OpenTap.Display(Name="Numpy Difference", Description="", Groups= ["DLS Python Plugin", "Numpy Step"]))
```

```
class NumpyDifference(DLSSStep):
    ResultName1 = property(String, 'Get Measurement').add_attribute(OpenTap.Display( 'Result Name 1', Order=10))
    ColumnName1 = property(String, 'Data').add_attribute(OpenTap.Display( 'Column Name 1', Order=11))
    ResultName2 = property(String, 'Get Measurement (1)').add_attribute(OpenTap.Display( 'Result Name 2', Order=12))
    ColumnName2 = property(String, 'Data').add_attribute(OpenTap.Display( 'Column Name 2', Order=13))
    def __init__(self):
        super().__init__()
```

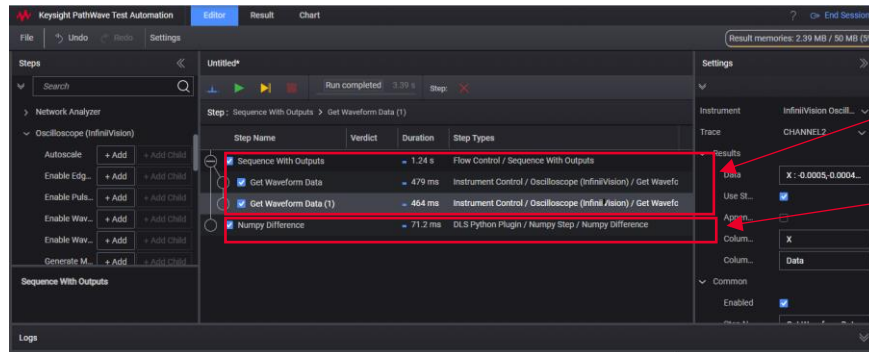
```
    def Run(self):
        [SelectedData1, SelectedData2] = super().GetStepData1(self.ResultName1, self.ColumnName1, self.ResultName2, self.ColumnName2)
        Data1 = np.fromiter(SelectedData1, dtype=np.float)
        Data2 = np.fromiter(SelectedData2, dtype=np.float)
        super().OutputToDLS('NumpyDifference', [Data2-Data1], [Data2-Data1])
```

First, import from “DLSOutputInput.py”

Inherit DLSSStep. The DLSSStep class contains a default InputTestStep selection by default for users to choose. Now, you can create a few more properties for users to select ResultName and ColumnName. In this case, we are going to get the results from two test steps.

Now, we will use “GetStepData1” to get the data from the two test steps by selecting the result name and column name. If only data from one test step is required, you can use “GetStepData” instead. We will do a simple subtraction here. You can further process the data if needed.

In this section, we will look at the use of NumpySteps.py described previously by extracting trace data from two different DLS built-in test steps. Create a new test plan and add two “Get Waveform Data” steps (from built-in Oscilloscope – InfiniiVision category) with each of them getting data from Channel 1 and Channel 2 respectively. Alternatively, you can also choose to use your Python-written “Trace Data” steps. Both test steps must be included as children of “Sequence With Outputs” step obtained from the Flow Control. Add “Numpy Difference” step created earlier at the bottom.

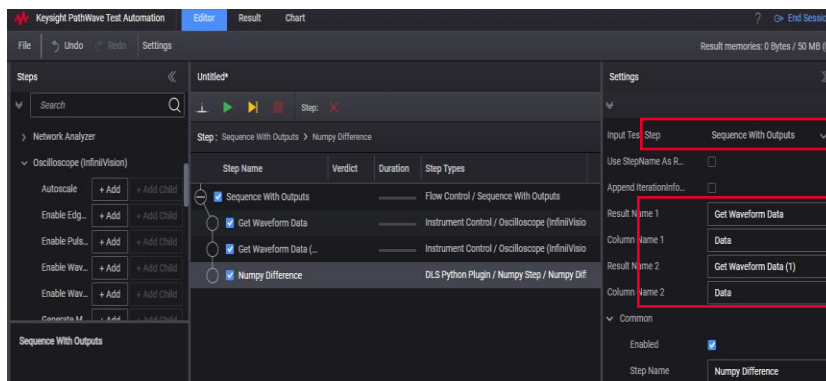


These are the built-in steps in DLS.

This is “Numpy Difference” step created with Python to process the data from the above built-in steps

Figure 4. Creating another test plan and test steps

At the “Numpy Difference” step, change the “Result Name” and “Column Name” to the respective “Get Waveform Data” steps’ step name and choose “Data” as the column name. Next, run the test plan and plot Data2-Data1. This effectively subtracts “Get Waveform Data (1)” containing the Channel 2 waveform data by “Get Waveform Data” containing the Channel 1 waveform data. If needed, you can also apply additional data manipulation using the built-in mathematical operations inside the X and Y axis text boxes as shown in Figure 5.



Input Test Step: Select the test step / sequence of test steps that contains the data

Choose ResultName and ColumnName that contains the data from the selected “Input Test Step”

Edit the X and Y boxes and select the respective data to plot. You can also add mathematical operations here if needed.

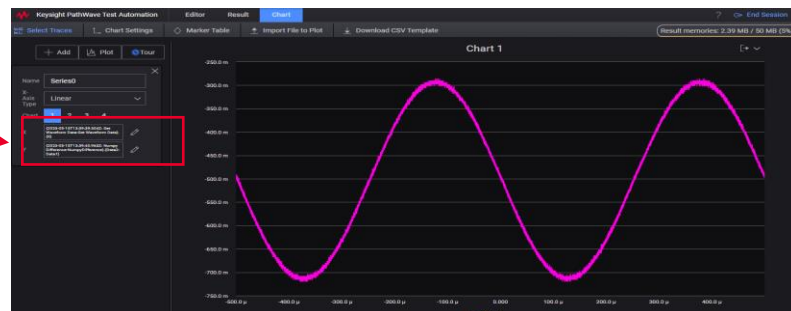


Figure 5. Viewing results

Conclusion

We have shown how Python scripts can be easily connected to DLS to create a powerful test automation on a web browser. This enables users to access instrument and measurement data anywhere, anywhere, with maximum flexibility and mobility. The DLS contains an industry-grade test sequencer based on OpenTAP test automation coupled with a versatile charting tool for users to visualize, compare, analyze, and share your test results in different format. Apart remote instrument controls and test automation, DLS also offers an easy access to thousands of on-demand industry-relevant Keysight learning resources. Also, the DLS enables multi-user remote access with built-in scheduler, optimized for hybrid collaborative learning. With built-in IMS LTI connection and SSO authentication, you can easily integrate DLS with your favorite Learning Management System or identity providers

For more information

Visit <https://www.keysight.com/us/en/product/SR101EDUA/digital-learning-platform.html> to learn more about DLS

For more insights on how to easily create test automation without programming, visit <https://www.keysight.com/us/en/assets/3123-1368/application-notes/Create-Test-Automation-without-Programming.pdf>

You can also take a quick tour of this software here <https://keysight.tourial.com/pages/0737d39c-cce0-4ac1-bfd3-e14d52d04425>



Keysight enables innovators to push the boundaries of engineering by quickly solving design, emulation, and test challenges to create the best product experiences. Start your innovation journey at www.keysight.com.

This information is subject to change without notice. © Keysight Technologies, 2023, Published in USA, June 6, 2023, 3123-1383.EN