

Getting Started with Instrument Control Using Python 3 and PyVISA

Introduction

Are you tasked with automating the setup and control of instruments for a test sequence? Python 3 with VISA provides a powerful combination for test automation in a user-friendly way. VISA includes the necessary code interface between the test software on your PC and the hardware in the box. The packaged VISA libraries provide standards for configuring, programming and troubleshooting instrument systems that use a variety of communication interfaces. Python is a simple yet powerful language and is easy for new users to learn. Python gives a user the ability to write complex scripts that capitalize on VISA's libraries to automate your testing. The IDE, PyCharm, provides users error checking, code completion, and other tools to simplify the coding process in an easy-to-use GUI. The follow document includes everything a beginner user needs to start the automation process, including:

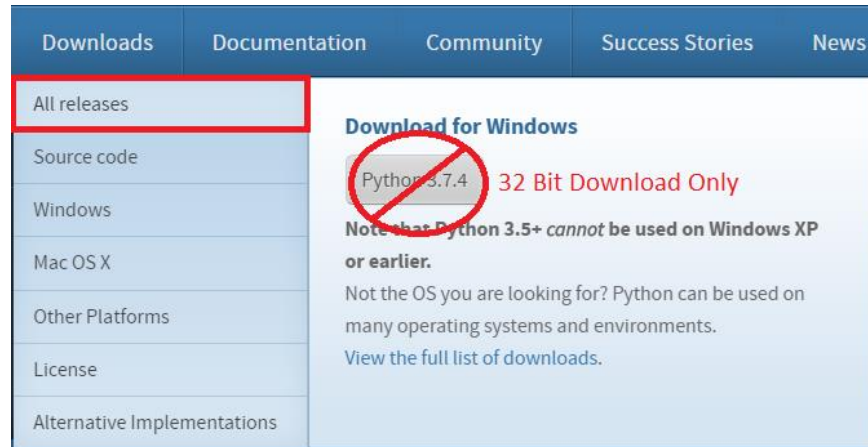
- Instructions for downloading and installing Python 3, NI VISA, PyVISA and PyCharm in a Windows environment.
- Steps for adding the VISA reference to your program.
- An overview on the general building blocks you will need for connecting to, sending commands to, and receiving data from your connected instrumentation.
- Building and running your simple instrument-controlling Python application.

This guide will not cover good coding practices and Python syntax in detail. For more information on coding in Python, please reference <https://www.python.org/>.

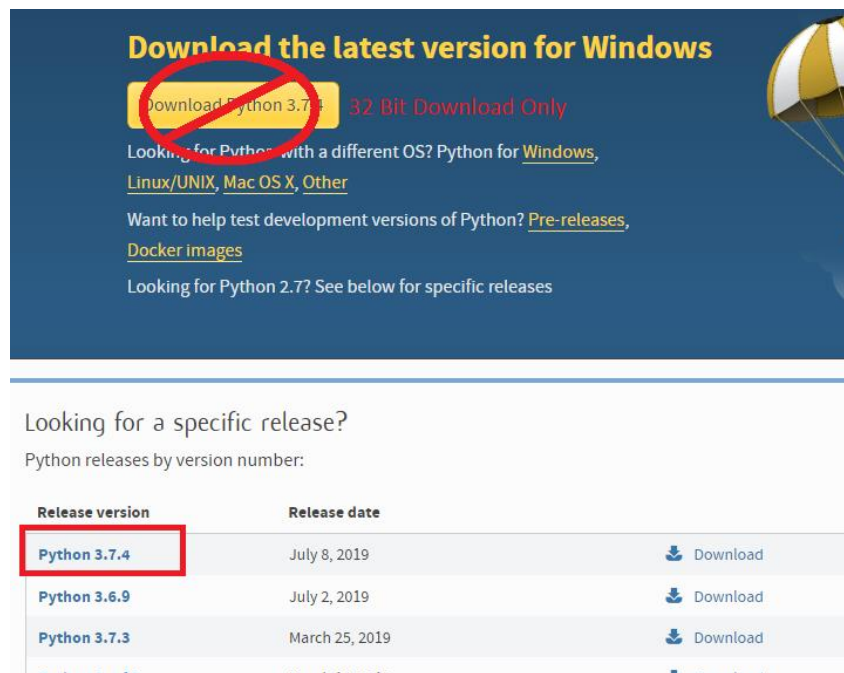
Installing Python 3

***NOTE:** These installation instructions assume that no version of Python is installed on the reader's computer. Having multiple versions of Python installed is possible but this work anticipates the user starting from scratch, and we have not investigated the benefits or problems that might be associated with supporting multiple versions. That is an advanced topic and will not be covered here. Please see <http://testerstories.com/2014/06/multiple-versions-of-python-on-windows/> for more information on installing and developing with multiple Python versions.*

1. Go to <https://www.python.org/> and click **Downloads**, then **All Releases**. Do not click the download button on the downloads dropdown menu, unless you are installing the 32-bit version.



2. Click on the link for **Python 3.7.4**. Do not click the yellow download button, unless you are installing the 32-bit version.

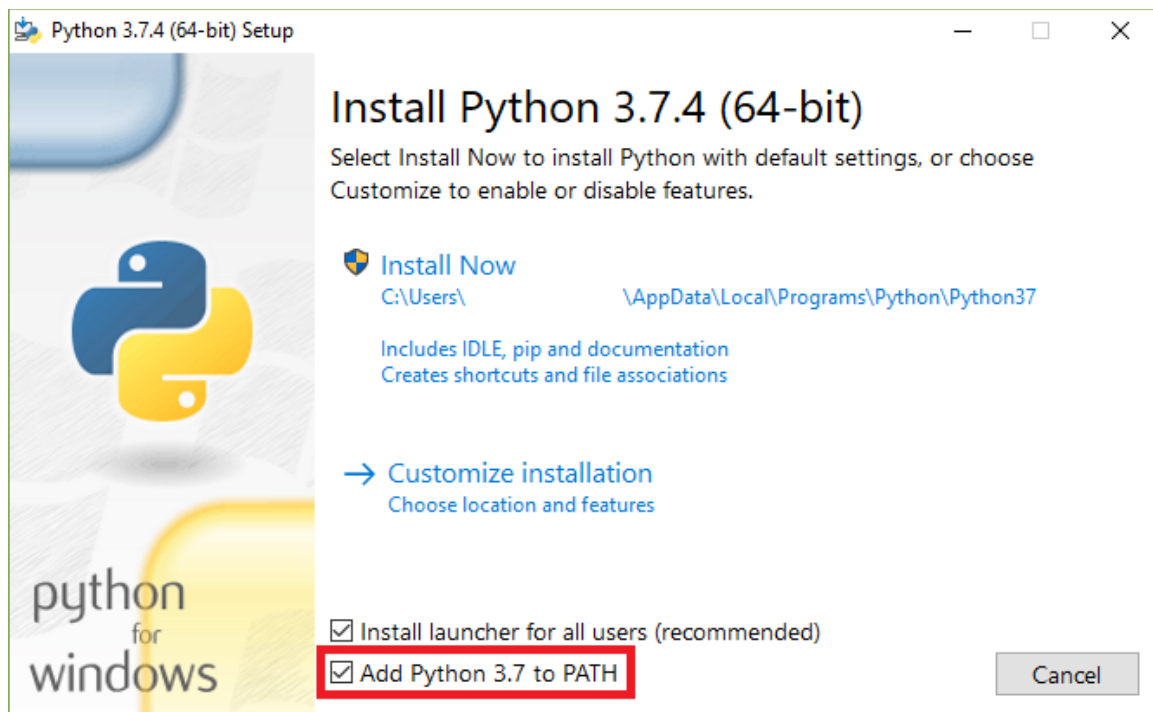


3. At the bottom of the page under Files, download either **Windows x86-64 executable installer** for a 64-bit system, or **Windows x86 executable installer** for a 32-bit system. The 64-bit version was selected for this example. If Windows asks you to Run or Save the program, select Run.

Files

Version	Operating System	Description
Gzipped source tarball	Source release	
XZ compressed source tarball	Source release	
macOS 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later
Windows help file	Windows	
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64
Windows x86-64 executable installer ★ 64	Windows	for AMD64/EM64T/x64
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64
Windows x86 embeddable zip file	Windows	
Windows x86 executable installer ★ 32	Windows	
Windows x86 web-based installer	Windows	

4. Run the downloaded executable and allow it to make changes to your PC. The Python Installer will open. Check the box at the bottom to **Add Python 3.7 to PATH**. Click **Install Now** to start the installation process.



5. Once the process is successful, close the installer.

Installing National Instruments VISA

1. Go to <https://www.ni.com/visa/> and click **Downloads** at the bottom under “Also See.”

Also see:

- [Product Information](#)
- [NI-VISA Licensing Information](#)
- [Downloads](#)
- [Related Areas](#)

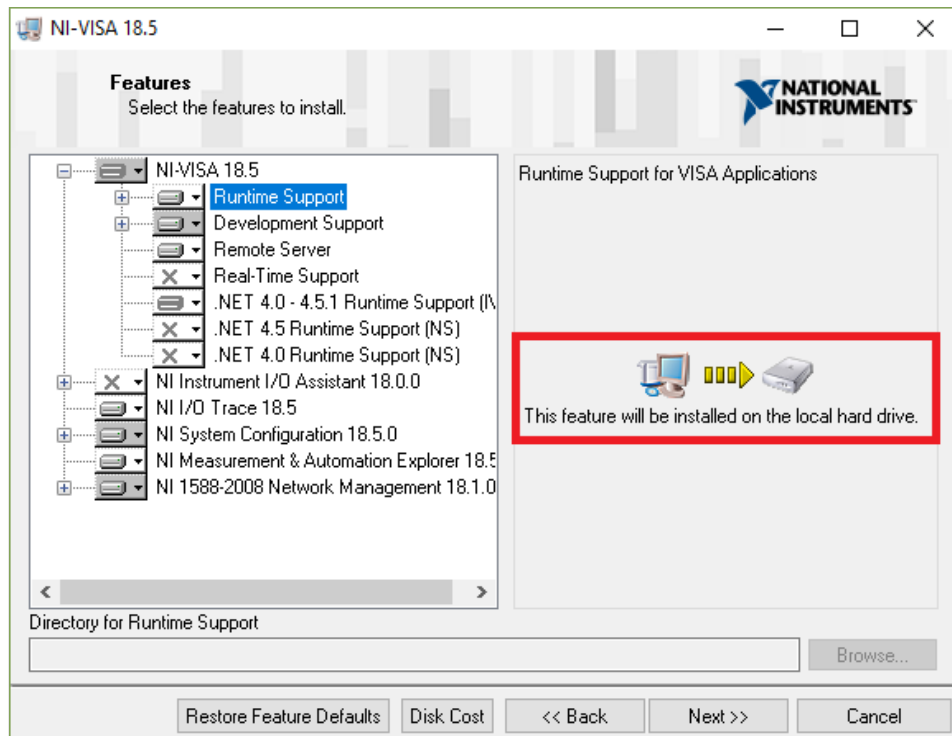


2. Select your operating system and version 18.5 (there are other versions available, but currently Keithley is recommending Version 18.5). Click **Download**.

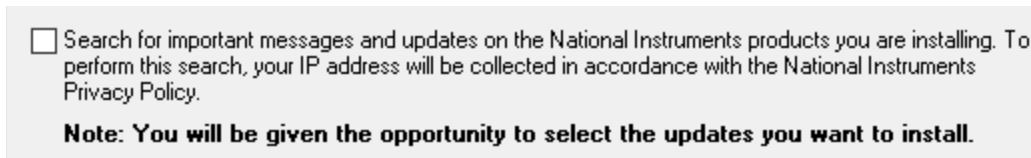
A screenshot of a web page titled "DOWNLOADS". It contains several selection options: "Supported OS" with a dropdown menu set to "Windows" and a "View Readme" link; "Version" with a dropdown menu set to "18.5" (this section is highlighted with a red box); "Application Bitness" with the text "32-bit & 64-bit"; "Included Editions" with a dropdown menu set to "Full"; and "Language" with the text "English".

DOWNLOADS		
Supported OS ⓘ	Windows ▼	View Readme
Version ⓘ	18.5 ▼	
Application Bitness ⓘ	32-bit & 64-bit	
Included Editions ⓘ	Full ▼	
Language ⓘ	English	

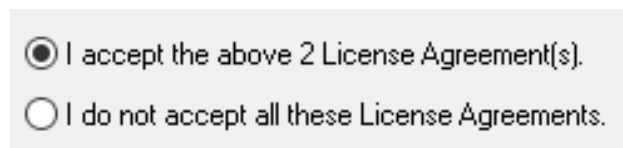
3. If Windows asks you to Open or Save the file, select Save or Save As. The file will download as a "zip" file. Find the file (NIVISA1850full.zip) in your downloads folder, or the location to which you saved it, and extract the files (right click >> Extract All...). In the extracted folder, double click the file **setup.exe** to run the installer.
4. Allow the application to make changes. Once the installer opens, hit **Next** to view the file path. Hit **Next** again to confirm the default file path.
5. In the next window, confirm that all features being installed will be installed on the local drive. The image to the right of the installation file tree will indicate the location of installation. Click **Next**.



6. Uncheck the box to receive notifications from the software. Click **Next**.



7. Accept all licenses. Click **Next** through both license windows.



8. Review software being installed. Click **Next** to begin the installation.

NOTE: The installation may take a while to finish.

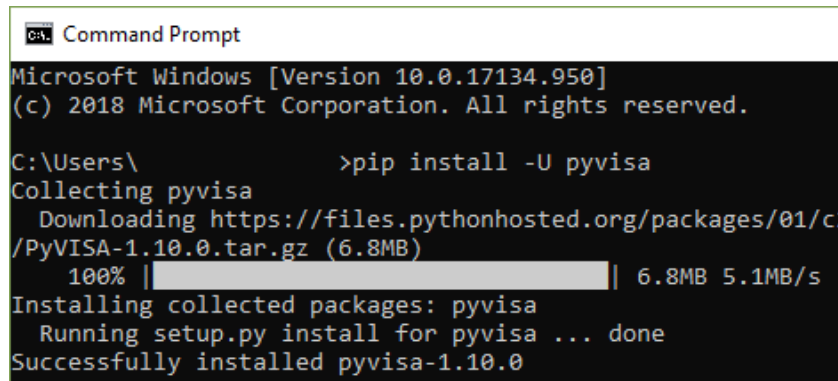
9. After the software is finished installing, click **Next** and then click to restart your computer now.

Installing PyVISA

1. Install PyVISA using PIP, a Python Package Manager. This manager is included with the Python install. To install PyVISA, open the Windows Command Prompt and type the following command.

```
pip install -U pyvisa
```

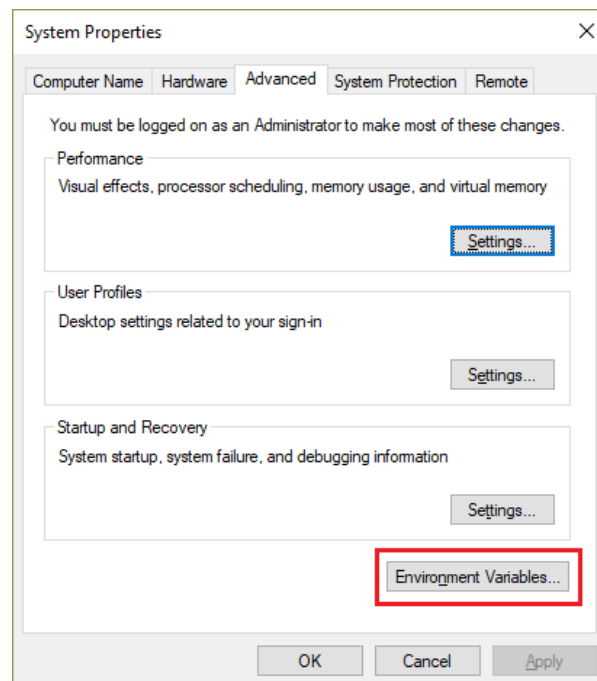
2. Hit Enter and allow the installer to run. Once the install is complete, you can close the command prompt.



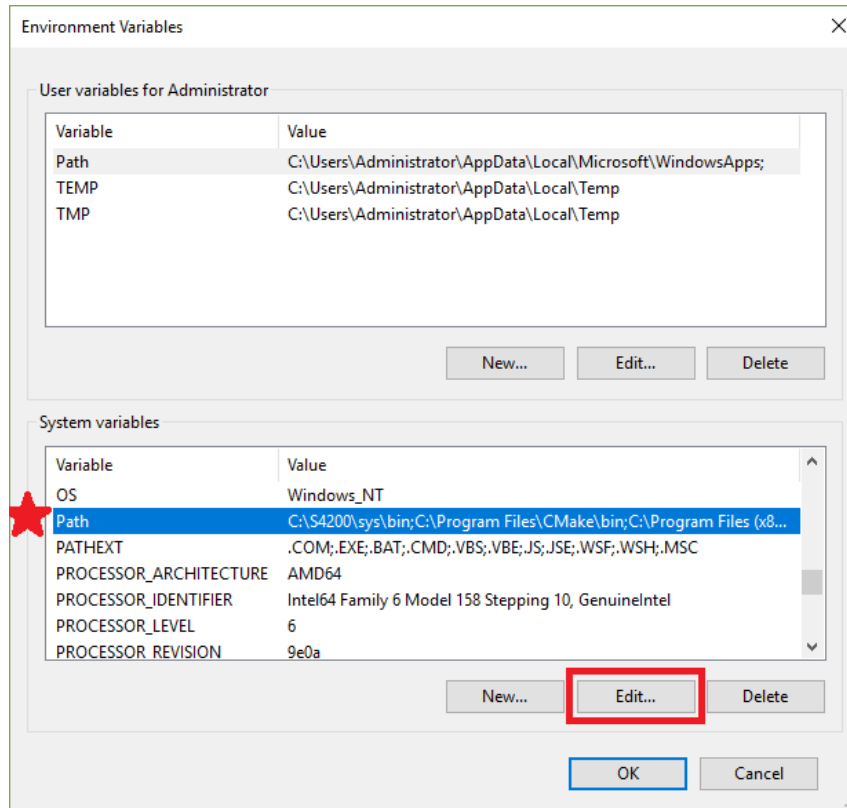
```
C:\Users\>pip install -U pyvisa
Collecting pyvisa
  Downloading https://files.pythonhosted.org/packages/01/c2/PyVISA-1.10.0.tar.gz (6.8MB)
    100% | 6.8MB 5.1MB/s
Installing collected packages: pyvisa
  Running setup.py install for pyvisa ... done
Successfully installed pyvisa-1.10.0
```

NOTE: If “pip” was not recognized by the PC, you may need to add PIP to the Path System Variables. There are multiple ways to get the Environmental Variables panel. One method is shown below:

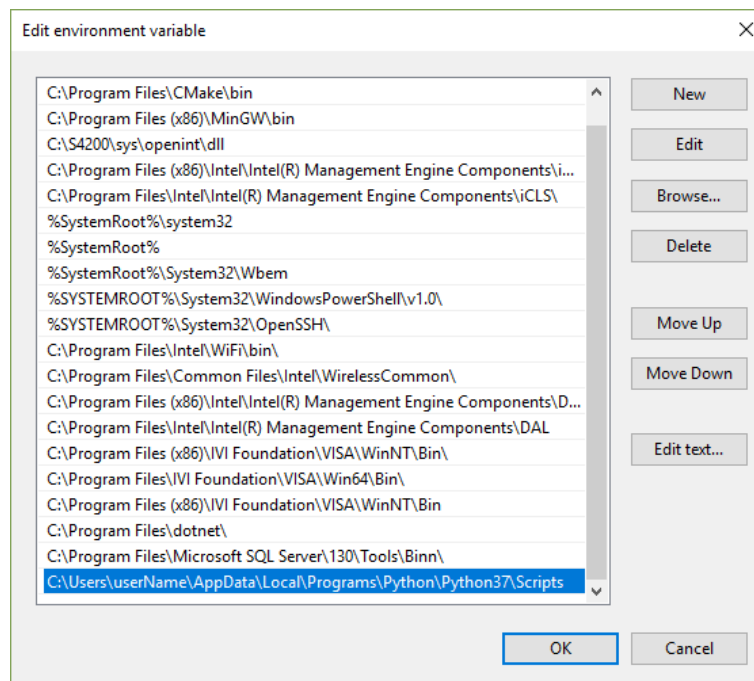
- 1) Open your control panel and go to System and Security >> System >> Advanced System Settings (on left side menu)
- 2) Click **Environment Variables...**



- 3) Find the variable **Path** in the System Variables list. Highlight that variable by clicking on it, and then click **Edit** below it.



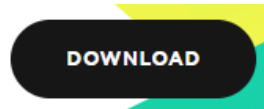
4) Click **New** in the window that opens and type in the path to the PIP.exe file. The PIP.exe file should be located in the Scripts Folder, which is in the folder where the Python executable was installed – in this example the Python37 folder.



5) Once this is entered, you can retry installing PyVISA.

Installing PyCharm

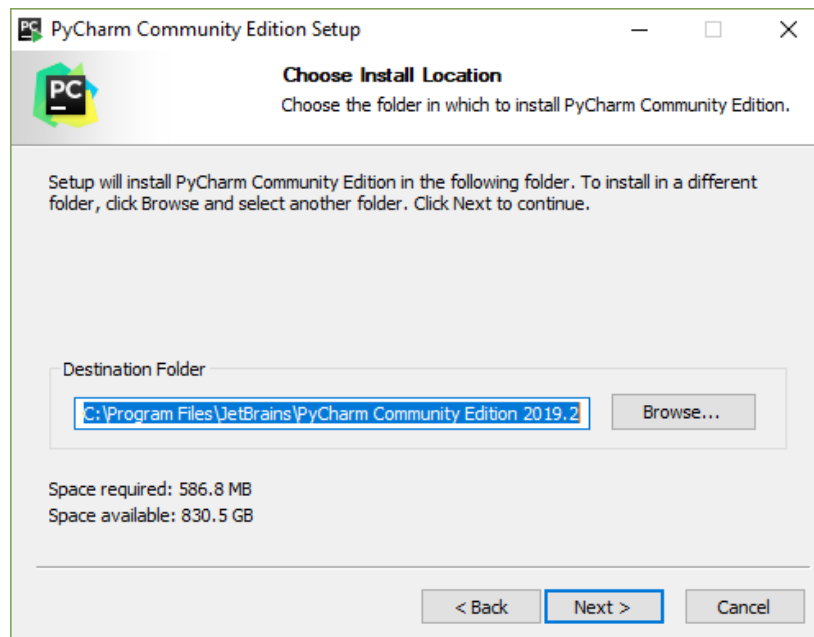
1. Go to <https://www.jetbrains.com/pycharm/>. Click **Download** in the center of the screen.



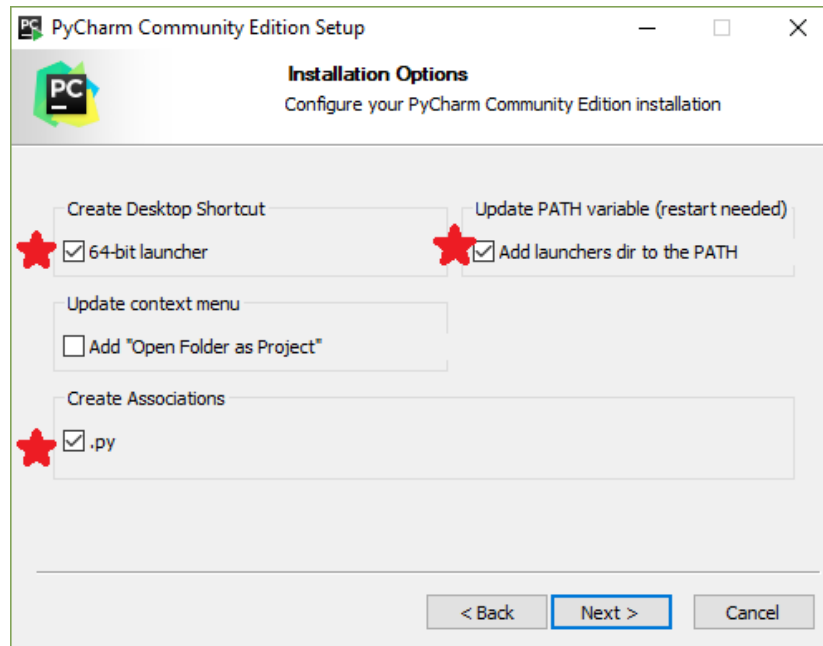
2. On the right, click the black button labeled **Download** to download the free PyCharm Community. If Windows asks you to Run or Save the program, select Run. Allow it to make changes to your device.



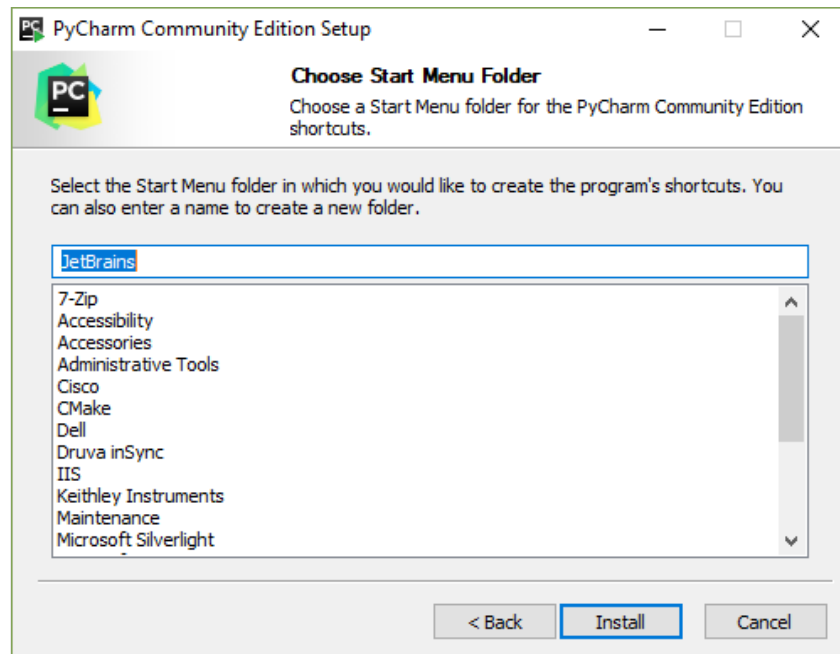
3. Once the Setup panel is open, click **Next** to continue the installation. Click **Next** again to install the files in the default Destination Folder.



4. Check the boxes to **Create Desktop Shortcut**, **Update PATH variable**, and **Create Associations**. Click **Next**.



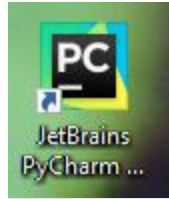
5. Click **Install** on the next window to start the installation. Use the default folder for the program's shortcuts.



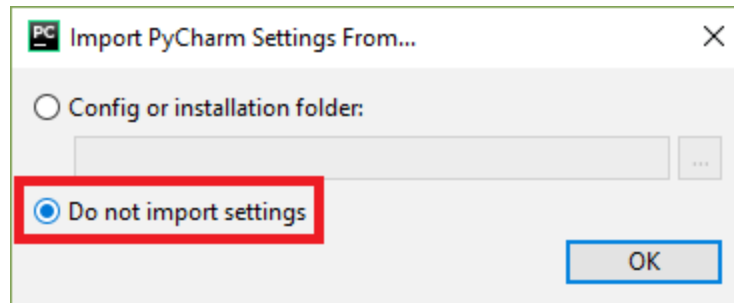
6. Restart your computer once the installation is complete.

Opening PyCharm for the First Time

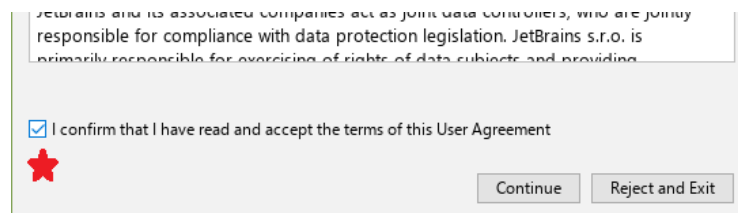
1. Open PyCharm by double clicking the desktop icon.



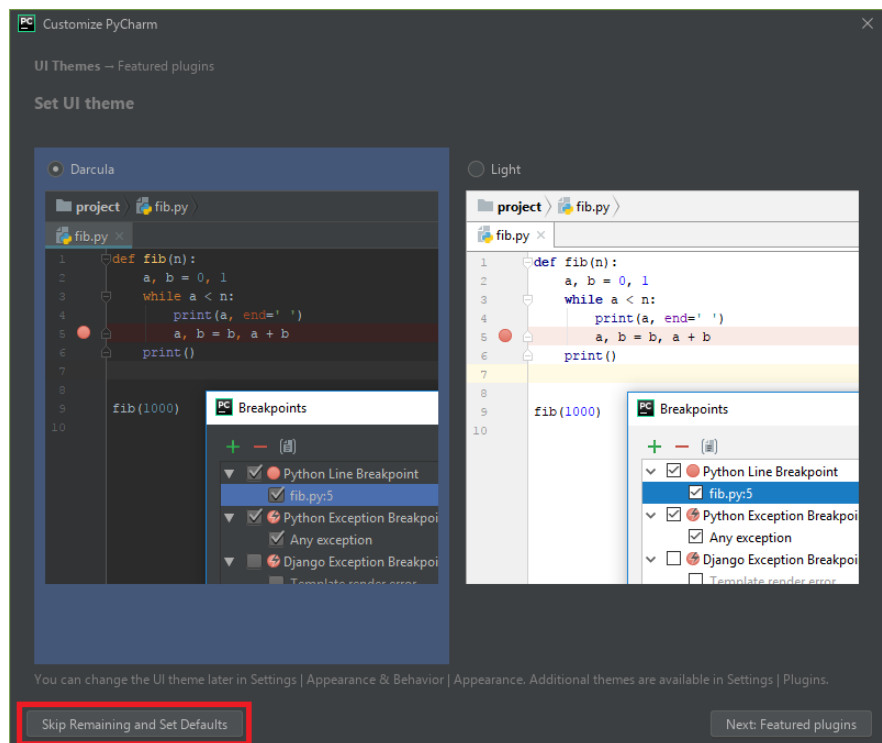
2. If this is your first time installing PyCharm, check **Do Not Import Settings**. Then click OK.



3. Agree to the license, then click Continue.

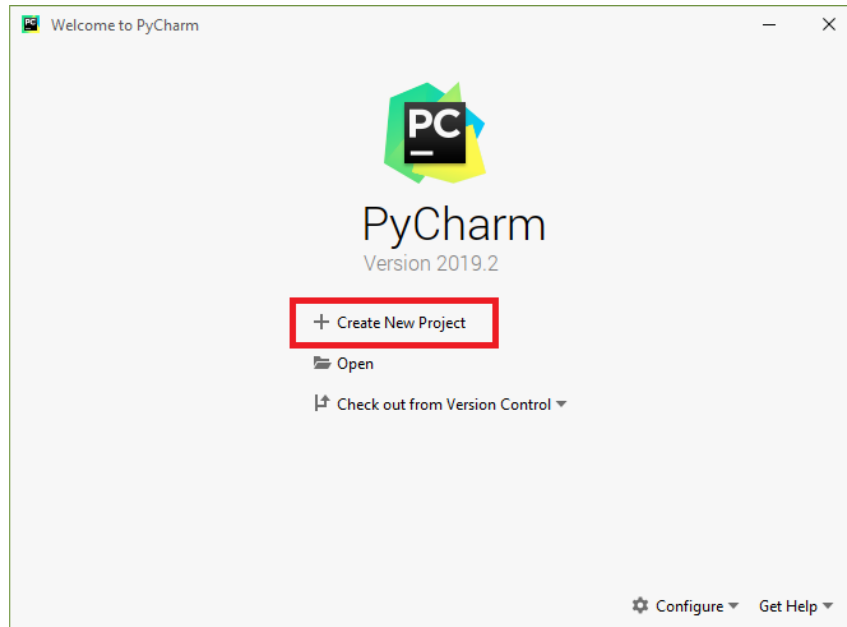


4. Select your color theme, then click on **Skip Remaining and Set Defaults**.

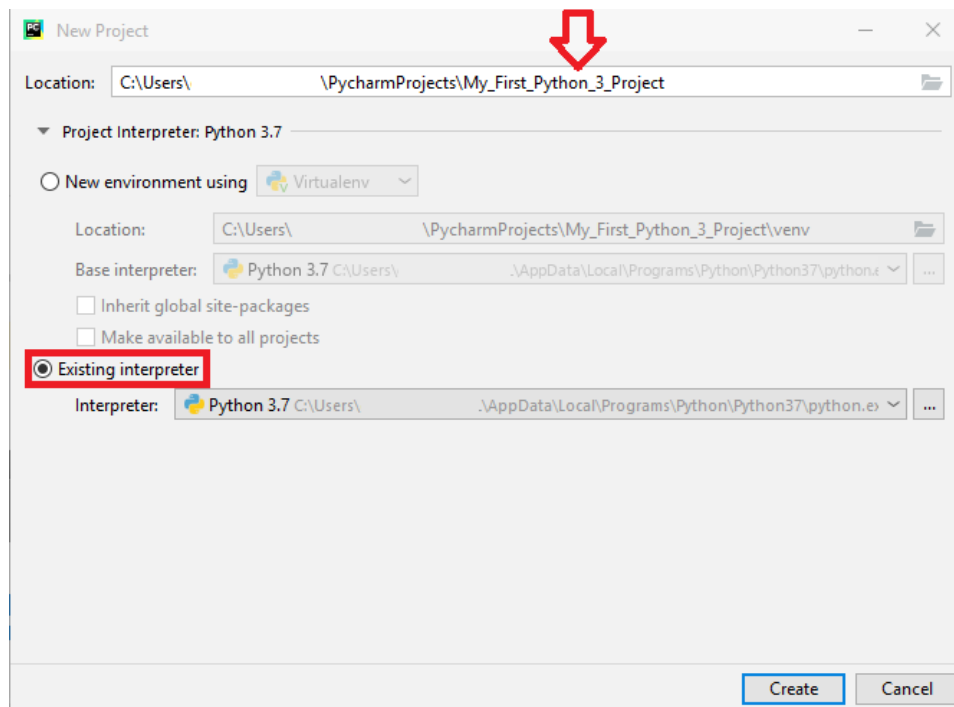


Create Your First Python Program

1. Click the PyCharm icon on your desktop to launch PyCharm. Click the **Create new project** button.

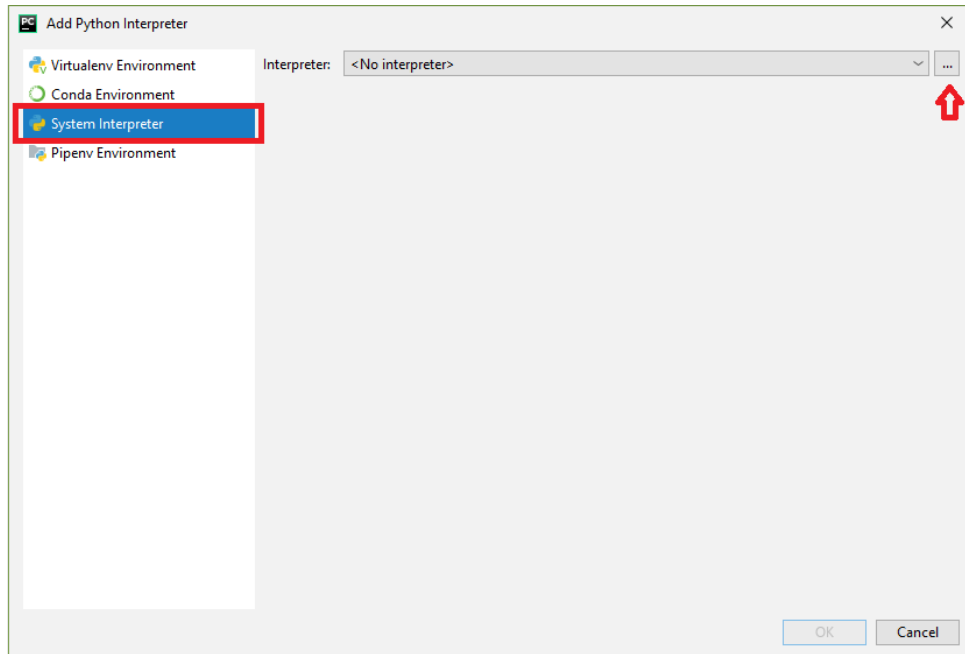


2. Specify the name of the project in the location file path. Expand **Project Interpreter** to see the options for the interpreter. Select **Existing interpreter**. The path for Python 3.7 should appear.

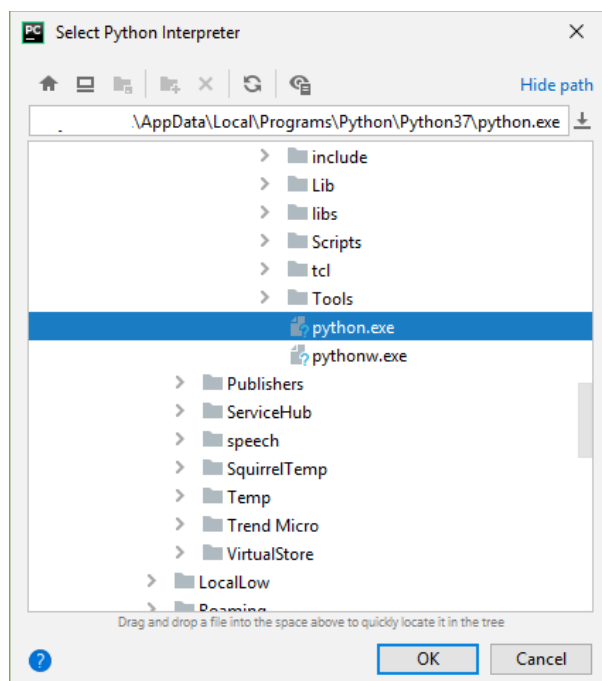


Note: If it does not find an interpreter, you can add one manually by clicking on the ellipsis button next to the path field.

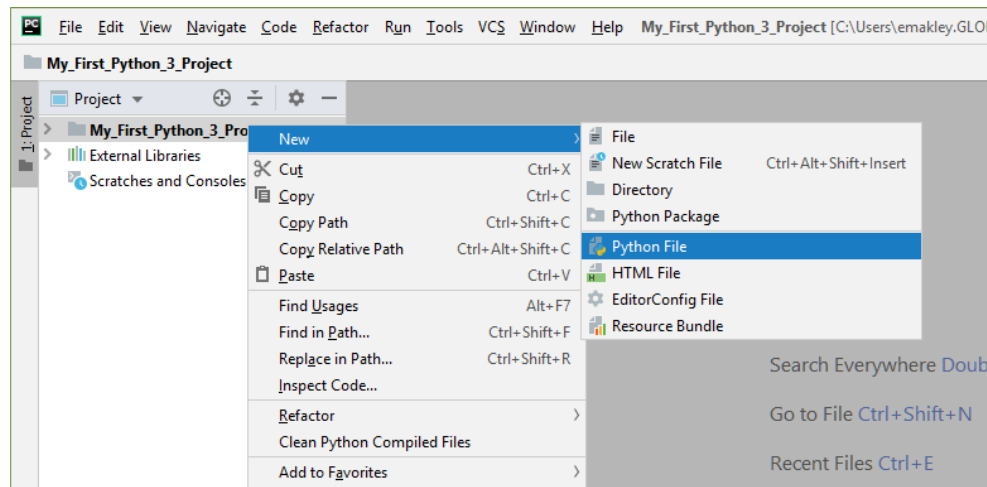
1) Select **System Interpreter** and click the ellipsis button next to the path area again.



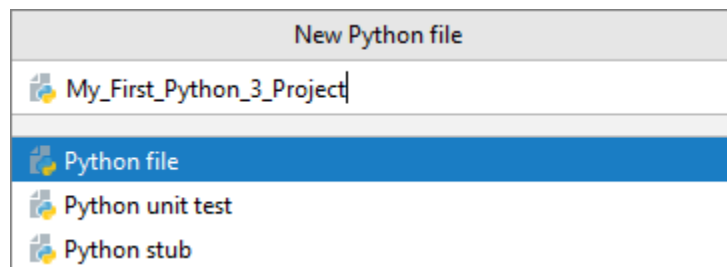
2) Input the path to python.exe. You may have to click on Show Hidden Files and Directories to see the AppData folder. Click OK, and then click OK again on the Add Python Interpreter panel. The path should now be added to your system interpreters.



- Once an interpreter is selected, click **Create** to create the project.
- Right Click on the project name in the project tree. Go to **New** and then to **Python File** to add a Python file to your project.



- Name the file and hit enter. The file will now open in the center of the screen. You are ready to start writing code!

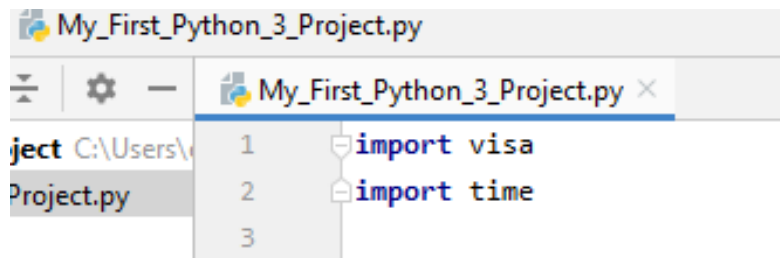


Writing a Program

NOTE: More information on using Python and VISA together is available at <https://pyvisa.readthedocs.io/en/latest/index.html>. Here you can find the API Documentation (functions, constants, parameters and return values) for all the library functions available to you. We only cover the basics in this guide.

The complete example program discussed in this section is given in Appendix A.

Begin by importing the libraries needed. Use the **import** keyword with the name of the library. In this example, we need the visa library to be able to control the instruments, and the time library to be able to use a timer in our program.



Let's first look at the main function, which is where we define a timer, resource manager and our instrument.

Start the main code by taking an initial time reading. We will keep track of the elapsed time by capturing the beginning and end times and taking the difference. To capture the time, use the `time()` function from the time library.

```
t1 = time.time() # Start the timer...
```

Next, define the resource string for our instrument. This string is the identification string of our instrument that VISA uses to connect to it. It defines the communication protocol; the instrument address and the type instrument being controlled. The instrument string can be found by launching the VISA Interactive Control Program.

```
instrument_resource_string = "USB0::0x05E6::0x7510::04345311::INSTR"
```

To connect to the instrument, we must instantiate the resource manager. The resource manager is the layer of VISA responsible for managing connections to instruments and identifying attributes about them in a relatively non-invasive manner. It can look at all resources connected to your computer without disrupting other programs which might be using them. It must be active before VISA can connect to your instrument.

```
resource_mgr = visa.ResourceManager() # Opens the resource manager
```

With the Resource Manager activated – or instantiated – we can now create an object that will be used to connect to your instrument and manage the transactions with it. We use the Resource Manager and this instrument object along with the instrument identification string to make the connection to the instrument.

We now need to define a variable name for our instrument. Since Python is not a strongly typed language, we can declare a simple name and assign it a value of `None`. Later, we'll give it a more useful value.

```
my_instr = None
```

We are ready to connect to the instrument. However, let's first pause for a minute and discuss the use of wrapper functions. Everything we are going to do in the rest of this example, could be done in the main program. However, we are going to use wrapper functions because they are more efficient to work with as test programs become more complex. Wrapper functions are functions that bundle other functions and variables. This abstraction can reduce complexity for a user, especially when the implementation of a library or other functions requires extensive or very complex coding. Wrapper

functions also work well for routines that are called repetitively or need to function across platforms. Calling a wrapper saves repetitive lines in the main function, reducing it to 1 function call that can be repeated as needed. And it can separate chunks of code that will be run together no matter what program they're in, saving time to rewrite functions for different programs.

In our example, we are wrapping functions with the intent of providing clarification and expansion for VISA tools that we are leveraging in our program. While some IDEs share brief pop-up information and suggestions as you type in your commands, the wrapper functions offer the ability to write explanatory comments in their definitions. In these comments, the user can share details of the function purpose, parameters, return value, and revision details that can assist other uses of all abilities.

Our first example of this is the `instrument_connect()` function. This function creates a resource manager, opens a resource to connect to the instrument and performs several other operations that the user can choose to be done upon start up. To begin the definition, we'll write the function name and the parameters after the keyword **def**.

```
def instrument_connect(resource_mgr, instrument_object, instrument_resource_string, timeout, do_id_query, do_reset,  
|                       do_clear):
```

All the parameters are explained in the comments above the definition. We pass the resource manager into and out of the function because we want to make sure than any changes to the resource manager carry through the rest of the program. It is good coding practice to include other information in the comments as well, so that other users are clear on the author's original intent and can use or edit the code as needed.

```

"""*****
Function: instrument_connect(resource_manager, instrument_resource_string, timeout,
                             do_id_query, do_reset, do_clear)

Purpose: Open an instance of an instrument object for remote communication.

Parameters:
    instrument_object (object) - Instance of an instrument object to be initialized
                                within this function.
    resource_manager (object) - Instance of a resource manager object.

    instrument_resource_string (string) - The VISA resource string associated with
                                           a specific instrument defining its connection
                                           characteristics (communications type, model,
                                           serial number, etc.)
    timeout (int) - Time in milliseconds to wait before the communication transaction
                    with the target instrument is considered failed (timed out)
    do_id_query (int) - A flag that determines whether or not to query and print the
                       instrument ID string.
    do_reset (int) - A flag that determines whether or not to issue a reset command to
                    the instrument during this connection.
    do_clear (int) - A flag that determines whether or not to issue a clear command to
                    the instrument during this connection.

Returns:
    None

Revisions:
    2019-08-07   JJB   Initial revision.
*****"""

```

Next is the code that executes the operations/actions the function is intended to perform, beginning with the connection to the instrument.

```
instrument_object = resource_mgr.open_resource(instrument_resource_string)
```

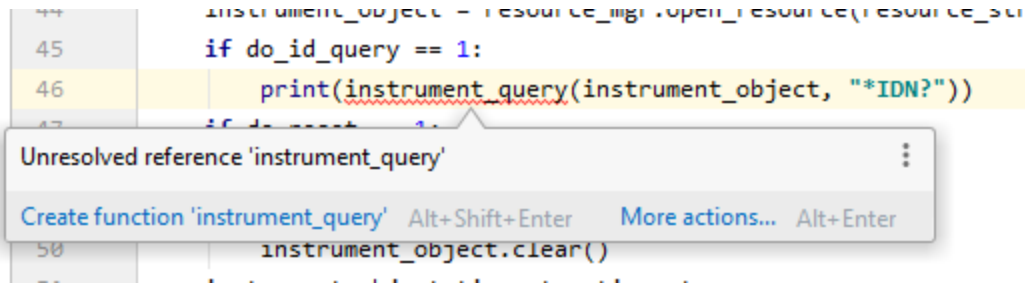
The next steps utilize the 3 flags that are set in the function parameters. Each of these flags represents an action. If the flag is set (1), then we do the action; if the flag is not set (0), then we move on.

```

if do_id_query == 1:
    print(instrument_query(instrument_object, "*IDN?"))
if do_reset == 1:
    instrument_write(instrument_object, "*RST")
if do_clear == 1:
    instrument_object.clear()

```

Note that PyCharm marks these other functions as errors. Hover over that line to see the error explanation from PyCharm.



The function errors because we haven't defined them yet and they are not an object method. We will define them later. For now, ignore the errors.

The next line assigns a timeout value for the instrument, which is essential when communicating with an instrument. This is the time that VISA will wait for the instrument to respond to a command. If the instrument stalls or otherwise does not respond in time, then VISA returns an error and moves the program along.

```
instrument_object.timeout = timeout
```

The final line is the return statement. We return the resource manager and the instrument object back to the main program so that they can be used elsewhere.

```
return resource_mgr, instrument_object
```

The `instrument_connect()` function is complete and we can call it from the main function. We set the call equal to the names which we have already defined so that our return values are assigned to our variable names.

```
resource_manager, my_instr = instrument_connect(resource_manager, my_instr, instrument_resource_string, 20000, 1, 1, 1)
```

Next, we'll define a few more wrapper functions that perform various operations. The `instrument_write()` function writes a SCPI command to the instrument. Note that in this function, we use a global variable to echo commands. A global variable is defined directly under the import statements and any function at any level can access its value. Echoing commands is a tool to debug the commands being set by printing all sent commands to the console. Defining it as a global variable allows us to turn it on and off for the whole program easily.

```

"""*****
Function: instrument_write(instrument_object, my_command)

Purpose: Issue controlling commands to the target instrument.

Parameters:
    instrument_object (object) - Instance of an instrument object.

    my_command (string) - The command issued to the instrument to make it
                           perform some action or service.

Returns:
    None

Revisions:
    2019-08-21    JJB    Initial revision.
*****"""

def instrument_write(instrument_object, my_command):
    if echo_commands == 1:
        print(my_command)
    instrument_object.write(my_command)
    return

```

The `instrument_read()` function reads some information from the target instrument and returns it to its caller.

```

"""*****
Function: instrument_read(instrument_object)

Purpose: Used to read commands from the instrument.

Parameters:
    instrument_object (object) - Instance of an instrument object.

Returns:
    <<<reply>>> (string) - The requested information returned from the
                           target instrument. Obtained by way of a caller
                           to instrument_read().

Revisions:
    2019-08-21    JJB    Initial revision.
*****"""

def instrument_read(instrument_object):
    return instrument_object.read()

```

The `instrument_query()` function writes a SCPI command to the instrument and returns information from the instrument directly after.

```

"""*****
Function: instrument_query(instrument_object, my_command)

Purpose: Used to send commands to the instrument and obtain an information string from the instrument.
Note that the information received will depend on the command sent and will be in string
format.

Parameters:
    instrument_object (object) - Instance of an instrument object.

    my_command (string) - The command issued to the instrument to make it
        perform some action or service.

Returns:
    <<<reply>>> (string) - The requested information returned from the
        target instrument. Obtained by way of a caller
        to instrument_read().

Revisions:
    2019-08-21    JJB    Initial revision.
*****"""

def instrument_query(instrument_object, my_command):
    if echo_commands == 1:
        print(my_command)
    return instrument_object.query(my_command)

```

The final function we need is a disconnect function. This function properly disconnects the instrument from the resource manager so that the PC is no longer linked to the instrument through VISA. You should always disconnect your instrument before ending a program.

```

"""*****
Function: instrument_disconnect(instrument_object)

Purpose: Break the VISA connection between the controlling computer
and the target instrument.

Parameters:
    instrument_object (object) - Instance of an instrument object.

Returns:
    None

Revisions:
    2019-08-21    JJB    Initial revision.
*****"""

def instrument_disconnect(instrument_object):
    instrument_object.close()
    return

```

We're almost finished with our program. All functions have been defined, and we can call them and use them in our main program. Here we'll write the command *RST which resets the instrument. Then we

loop a set of write, read and query the command `*IDN?` to receive identification information from the instrument. Any returned information is printed out to the console. After the loop, we disconnect the instrument, close the resource manager, record the completion time and then alert the user that the process is done after the elapsed time. An input prompt allows the user to terminate the program when they are ready.

```
instrument_write(my_instr, "*RST")
for j in range(1, 10):
    instrument_write(my_instr, "*IDN?")
    print(instrument_read(my_instr))
    print(instrument_query(my_instr, "*IDN?")) # query is the same as write + read

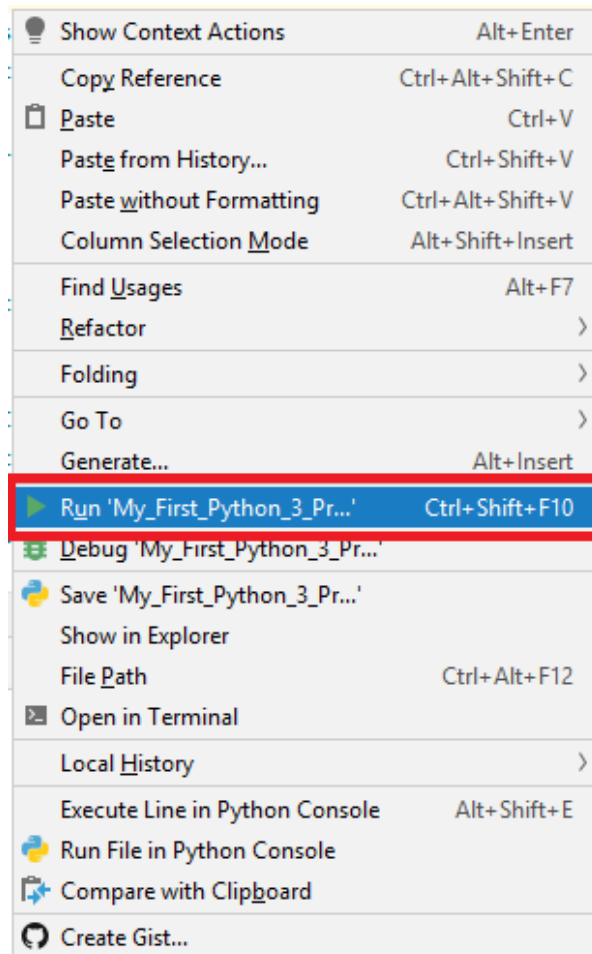
instrument_disconnect(my_instr)
resource_manager.close

t2 = time.time() # Stop the timer...

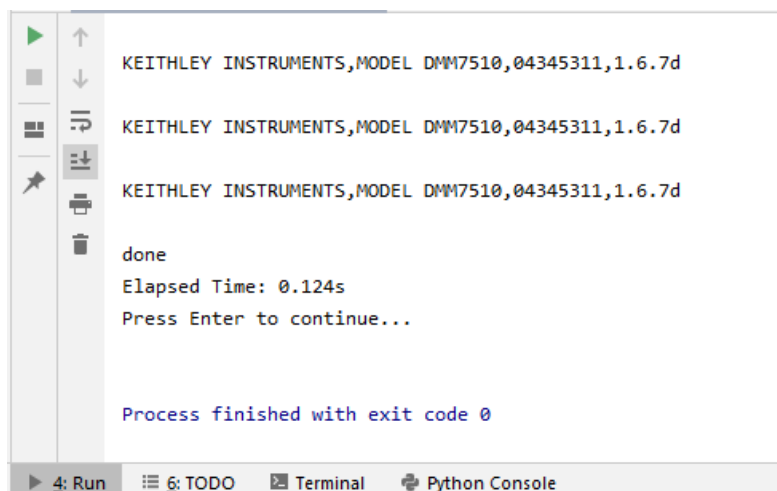
# Notify the user of completion and the data streaming rate achieved.
print("done")
print("Elapsed Time: {0:0.3f}s".format(t2 - t1))

input("Press Enter to continue...")
exit()
```

There are multiple ways to run the program including pressing Shift+F10, selecting “Run <Project_Name>” from the Run tab at the top or right clicking in the code editor window and selecting **Run <Project Name>**. This last method is shown below.



When the program runs, the output is sent to the run tab at the bottom of the screen.



Conclusion

Congrats, you controlled an external instrument using your PC! As noted earlier, there are many more features of the VISA library. See the API Documentation at

<https://pyvisa.readthedocs.io/en/latest/index.html> for a complete list of all available methods. PyCharm also has many debugging features that are beyond the scope of this informational document. We strongly encourage you to explore VISA, the Python language and PyCharm's feature to advance your automation skills. In the Appendix below, you will find the example we covered here today. You can copy and paste it into PyCharm to experiment with. This code (along with numerous other examples) can be found at <https://github.com/tektronix/keithley>.

Appendix A – Example Code

The following code is the full complete code example that this document walks through. This code (along with numerous other examples) can be found at <https://github.com/tektronix/keithley>.

```
import visa
import time

echo_commands = []

"""*****
Function: instrument_connect(resource_mgr, instrument_resource_string, timeout,
                             do_id_query, do_reset, do_clear)

Purpose: Open an instance of an instrument object for remote communication.

Parameters:
    resource_mgr (object) - Instance of a resource manager object.

    instrument_object (object) - Instance of an instrument object to be initialized
                                within this function.

    instrument_resource_string (string) - The VISA resource string associated with
                                         a specific instrument defining its connection
                                         characteristics (communications type, model,
                                         serial number, etc.)

    timeout (int) - Time in milliseconds to wait before the communication transaction
                    with the target instrument is considered failed (timed out)
    do_id_query (int) - A flag that determines whether or not to query and print the
                       instrument ID string.
    do_reset (int) - A flag that determines whether or not to issue a reset command to
                    the instrument during this connection.
    do_clear (int) - A flag that determines whether or not to issue a clear command to
                    the instrument during this connection.

Returns:
    None

Revisions:
    2019-08-07    JJB    Initial revision.
*****"""

def instrument_connect(resource_mgr, instrument_object, instrument_resource_string, timeout, do_id_query,
                      do_reset,
                      do_clear):
    instrument_object = resource_mgr.open_resource(instrument_resource_string)
    if do_id_query == 1:
        print(instrument_query(instrument_object, "*IDN?"))
    if do_reset == 1:
        instrument_write(instrument_object, "*RST")
    if do_clear == 1:
        instrument_object.clear()
    instrument_object.timeout = timeout
    return resource_mgr, instrument_object
```

```

"""*****
Function: instrument_write(instrument_object, my_command)

Purpose: Issue controlling commands to the target instrument.

Parameters:
    instrument_object (object) - Instance of an instrument object.

    my_command (string) - The command issued to the instrument to make it
                           perform some action or service.

Returns:
    None

Revisions:
    2019-08-21    JJB    Initial revision.
*****"""

```

```

def instrument_write(instrument_object, my_command):
    if echo_commands == 1:
        print(my_command)
    instrument_object.write(my_command)
    return

```

```

"""*****
Function: instrument_read(instrument_object)

Purpose: Used to read commands from the instrument.

Parameters:
    instrument_object (object) - Instance of an instrument object.

Returns:
    <<<reply>>> (string) - The requested information returned from the
                           target instrument. Obtained by way of a caller
                           to instrument_read().

Revisions:
    2019-08-21    JJB    Initial revision.
*****"""

```

```

def instrument_read(instrument_object):
    return instrument_object.read()

```

```

"""*****
Function: instrument_query(instrument_object, my_command)

Purpose: Used to send commands to the instrument and obtain an information string from the
         instrument. Note that the information received will depend on the command sent and
         will be in string format.

Parameters:
    instrument_object (object) - Instance of an instrument object.

    my_command (string) - The command issued to the instrument to make it
                           perform some action or service.

Returns:
    <<<reply>>> (string) - The requested information returned from the
                           target instrument. Obtained by way of a caller
                           to instrument_read().

```

```

Revisions:
    2019-08-21    JJB    Initial revision.
*****"

def instrument_query(instrument_object, my_command):
    if echo_commands == 1:
        print(my_command)
    return instrument_object.query(my_command)

"""*****
Function: instrument_disconnect(instrument_object)

Purpose: Break the VISA connection between the controlling computer
        and the target instrument.

Parameters:
    instrument_object (object) - Instance of an instrument object.

Returns:
    None

Revisions:
    2019-08-21    JJB    Initial revision.
*****"""

def instrument_disconnect(instrument_object):
    instrument_object.close()
    return

# =====
#
#   MAIN CODE STARTS HERE
#
# =====
t1 = time.time() # Start the timer...
instrument_resource_string = "USB0::0x05E6::0x7510::04345311::INSTR"
resource_manager = visa.ResourceManager() # Opens the resource manager
my_instr = None
resource_manager, my_instr = instrument_connect(resource_manager, my_instr, instrument_resource_string,
20000, 1, 1, 1)

instrument_write(my_instr, "*RST")
for j in range(1, 10):
    instrument_write(my_instr, "*IDN?")
    print(instrument_read(my_instr))
    print(instrument_query(my_instr, "*IDN?")) # query is the same as write + read

instrument_disconnect(my_instr)
resource_manager.close

t2 = time.time() # Stop the timer...

# Notify the user of completion and the data streaming rate achieved.
print("done")
print("Elapsed Time: {0:0.3f}s".format(t2 - t1))

input("Press Enter to continue...")
exit()

```


References

None

Revisions

Revision	Date	Authored by	Notes
1.0	2019-08-23	Josh Brown and Elizabeth Makley	