# Overview/Using this guide

**Title:**

Wet Test Fixture Python Programming Guide

**Client:**

Profound Medical Inc, 2400 Skymark Ave #6, Mississauga, ON L4W 5K5, Canada

**Date:**

June 15th, 2022

**Contributions:**

|  |  |
| --- | --- |
| Program function & Design | Isaiah Baker |
| Programming & development | Isaiah Baker, Rameen Housseini |
| Report Preparation | Isaiah Baker, Rameen Housseini, Axel Montano |
| Corresponding Author | Rajiv Chopra (rajiv@8foldmfg.com) |

**Report Revisions:**

|  |  |
| --- | --- |
| Revision 1.0 | Initial Guide |

**Overview/Summary of utilizing this guide**

This guide is intended to provide a general understanding of how the parts of the Wet Test Fixture Python Interface codebase work and interact with each other. It will provide information on where to find the relevant classes and functions so future programmers can troubleshoot problems, maintain the code, and make improvements and additions in the future. It will also walk the programmer through the examples found in <repository folder>\Examples which provide simplified examples of key concepts to this application’s design.

More detailed information on how the code works can be found in the docstrings at the top of most functions and classes, as well as the comments found throughout the code. This document also contains a linked glossary to key terms, so it is recommended to click on those whenever available to fully understand the topics being discussed.

Below is an interactive table of contents that will hyperlink you to the relevant chapters and pages for the topic you select.

Table of Contents

[Overview/Using this guide 1](#_Toc106711414)

[Table of Contents 3](#_Toc106711415)

[Technologies and Dependencies 4](#_Toc106711416)

[General Programming 4](#_Toc106711417)

[Python 3.8 4](#_Toc106711418)

[GitHub 4](#_Toc106711419)

[Conda 4](#_Toc106711420)

[User Interface/Application Architecture/Multithreading 5](#_Toc106711421)

[PyQt5 5](#_Toc106711422)

[Hardware Interfacing Libraries 5](#_Toc106711423)

[Mini Circuits PWR-SEN-8FS Power Meter (Pythonnet, import clr) 5](#_Toc106711424)

[Keysight DSOX2002A Oscilloscope, 33500B Signal Generator (PyVISA) 5](#_Toc106711425)

[NI USB-6009 Digital IO board, NI USB-TC01 thermocouple (NI-DAQmx) 5](#_Toc106711426)

[Galil DMC 4143 Motor Controller (gclib) 6](#_Toc106711427)

[UA Interface (subprocess) 6](#_Toc106711428)

[Serial library (MT Balance, Parker Motor Controller, Power relay) 6](#_Toc106711429)

[Debugging 6](#_Toc106711430)

[Logging Library 6](#_Toc106711431)

[Termcolor library 7](#_Toc106711432)

[Data Analysis 7](#_Toc106711433)

[Numpy, Scipy, and Statistics Libraries 7](#_Toc106711434)

[Re library 7](#_Toc106711435)

[File input / output 7](#_Toc106711436)

[Shutil Library 7](#_Toc106711437)

[Yaml library 7](#_Toc106711438)

[ConfigParser 8](#_Toc106711439)

[Abstraction 8](#_Toc106711440)

[ABC (abstract base class) library 8](#_Toc106711441)

[High Level Overview 9](#_Toc106711442)

[Class Diagram 9](#_Toc106711443)

[Architecture Explanation 10](#_Toc106711444)

[Sequence of Events 11](#_Toc106711445)

[Startup 11](#_Toc106711446)

[Scripts 11](#_Toc106711447)

[Abort Immediately Button Clicked 12](#_Toc106711448)

[Abort After Step Button Clicked 12](#_Toc106711449)

[Summaries of Complex Script Actions 13](#_Toc106711450)

[Measure Element Efficiency RFB 13](#_Toc106711451)

[Find Element ‘n’ 13](#_Toc106711452)

[Organization of Files 14](#_Toc106711453)

[Top Level Files 14](#_Toc106711454)

[User Interface files and classes: 14](#_Toc106711455)

[Hardware Classes and Tests: 14](#_Toc106711456)

[Abstract Hardware Classes 15](#_Toc106711457)

[Simulated Hardware Classes 15](#_Toc106711458)

[Real Hardware Classes 16](#_Toc106711459)

[Other Files 16](#_Toc106711460)

[definitions.py: 16](#_Toc106711461)

[formulas.py: 16](#_Toc106711462)

[environment.yml: 17](#_Toc106711463)

[Systeminfo.ini: 17](#_Toc106711464)

[Utilities folder: 17](#_Toc106711465)

[Key Classes Explained 18](#_Toc106711466)

[Manager 18](#_Toc106711467)

[MainWindow 18](#_Toc106711468)

[Programming Guidelines and Best Practices 20](#_Toc106711469)

[Type Hinting 20](#_Toc106711470)

# Table of Contents

# Technologies and Dependencies

**Technologies and Dependencies**

This section includes a summary of the technologies used in the development of this application, as well as their use and justification. For instructions on installing these dependencies, refer to the readme.md file in the root of the repository.

## General Programming

**General Programming**

### Python 3.8

**Python 3.8**

Python 3.8 is the latest release of Python that can be used with Python.NET, which is necessary for the power meter’s hardware class. Python.NET is also compatible with Python 2.7, 3.5, 3.6, and 3.7, but we recommend using Python 3.8 since all our testing has been done with this version of Python.

### GitHub

**GitHub**

GitHub’s GIT VCS (version control system) allows for seamless branch creating and merging. As well as issue tracking and change blaming. The most important aspect of GitHub is the ability to review changes before a merge, before a commit, and to rollback to previous versions of the code in case something important was lost via a bad merge, or a computer crash, etc. GitHub also has a user-friendly GUI and is free to use.

### Conda

**Conda**

While not required, Conda makes installing required libraries and packages seamless and easy. It also allows you to write/export an environment file that others can use to automatically install all required packages on their machine for quick setup. The GUI aspect of Conda is called Anaconda Navigator which is also essential for working with Conda. PyCharm comes with integrated support for Conda, we are unsure about other Python IDE’s, so we recommend you research the IDE’s support for Conda before adopting it if you wish to use a different IDE. Conda also allows installing packages and libraries from different repositories from the default one. So far, the only repository we had to install from in addition to the default one is the “forge” repository. These can also be pre-defined in the environment file.

## User Interface/Application Architecture/Multithreading

**User Interface/Application Architecture/Multithreading**

### PyQt5

**PyQt5**

PyQt5 can be installed through Conda, or pip. This library must be imported by any class that uses a item type that has anything to do with the Qt GUI library. Even items that are not explicitly GUI related (such as ‘qWait,’ or ‘QEvent’) need the PyQt5 library imported. It may be theoretically possible to upgrade the project to PyQt6 but we do not recommend this unless the software team has a lot of time to dedicate fixing potential issues from arising, and there is no substantial difference between the two.

## Hardware Interfacing Libraries

**Hardware Interfacing Libraries**

### Mini Circuits PWR-SEN-8FS Power Meter (Pythonnet, import clr)

**Mini Circuits PWR-SEN-8FS Power Meter (Pythonnet, import clr)**

Pythonnet is a required library for interpreting DLL files and using methods built into them in the Python source code. As mentioned above, this was only needed for the power meter hardware class, and is only compatible with a few Python versions including Python 3.8. Refer to the power meter programming manual in the Hardware Programming Manuals folder for more information.

### Keysight DSOX2002A Oscilloscope, 33500B Signal Generator (PyVISA)

**Keysight DSOX2002A Oscilloscope, 33500B Signal Generator (PyVISA)**

The PyVISA library is required to communicate with the Keysight equipment using python. Its installer includes the NI-Visa interactive control tool for sending and receiving commands with these instruments directly in a terminal-like interface, which is useful for debugging and testing commands before using them in Python.

### NI USB-6009 Digital IO board, NI USB-TC01 thermocouple (NI-DAQmx)

**NI USB-6009 Digital IO board, NI USB-TC01 thermocouple (NI-DAQmx)**

The NI-DAQmx library is required for communicating with the digital IO board, as well as the thermocouple.

### Galil DMC 4143 Motor Controller (gclib)

**Galil DMC 4143 Motor Controller (gclib)**

The Galil motor controllers need to have gclib imported for Python to communicate with it. You may either download the API from the official galil website (<http://www.galil.com/downloads/api/>) or from conda directly into your working environment.

### UA Interface (subprocess)

**UA Interface (subprocess)**

Subprocess is a built-in library that allows python to run executables with arguments and direct their output. This is required to use the proprietary executable for reading and writing data using the UA interface. This executable can be found in the abstract\_ua\_interface folder.

### Serial library (MT Balance, Parker Motor Controller, Power relay)

**Serial library (MT Balance, Parker Motor Controller, Power relay)**

This is a built-in library used for sending and receiving data over a USB connection, usually as a stream of text or bytes. This is used for communicating for COM devices such as the MT Balance, Parker Motor Controller, Power relay. Each has a specific COM port, which can be found using device manager, as well as a specific Baud rate (check the hardware manuals if issues arise), and may require a newline or a carriage return character between commands.

We recommend using a simple serial terminal software for troubleshooting these devices (such as the one built into the Arduino IDE)

## Debugging

**Debugging**

### Logging Library

**Logging Library**

The logging library is what unifies all log statements in all classes to one handler and processor. The self.log(level:str, message:str) method is accessible to most of the frontend and backend classes. Hardware devices inherit it from the AbstractDevice class, whereas UI classes inherit it either from the MyQWidget or MyQDialog custom classes. These inherited classes handle the setup of these methods so it does not need to be repeated in every class. This includes the necessary imports, instantiating the root logger, et cetera. This allows all of the classes to collectively modify the WTF.log file in the logs folder. This method automatically adds the line number, class name, and thread name of where the log message was coming from.

### Termcolor library

**Termcolor library**

Allows for colored output, very useful for debugging via print statements.

Data Analysis

**Data Analysis**

Numpy, Scipy, and Statistics Libraries

**Numpy, Scipy, and Statistics Libraries**

Numpy and Scipy basically handle the heavy lifting of writing mundane to complex mathematical functions. We’ve used them to integrate a list of squared values using Simpson’s rule. Particularly when finding the voltage squared integral of the oscilloscope waveform. Numpy and Scipy are not built in but can be installed via Conda.

Re library

**Re library**

This python library allows us to search through strings via a regular expression, a very useful search syntax pattern that allows for seamless searches of complex and unconventional patterns.

File input / output

**File input / Output**

Shutil Library

**Shutil Library**

Used in the FileSaver class, gives python the ability to copy files amongst other high-level file operations.

Yaml library

**Yaml library**

A library that allows for the automatic and integrated interpretation of yaml files in Python. Can be installed through the package ‘pyyaml’ in Conda.

ConfigParser

**ConfigParser**

A built-in library for parsing ini files such as systeminfo.ini, used in the ui\_system\_info class.

Abstraction

**Abstraction**

ABC (abstract base class) library

**ABC (abstract base class) library**

This library is needed to write abstract classes and methods, as well as allowing multiple abstraction layers for a concrete class via importing ABCMeta from ABC (a good example of this is the abstract\_device.py file in the /Hardware/Abstract/ directory. This abstract class inherits QObject, which itself is an abstract class. In order to allow this, ABCMeta must be imported and used via: \_\_metaclass\_\_ = ABCMeta inside the class definition.

High Level Overview

**High Level Overview**

Class Diagram

**Class Diagram**

Diagram

Description automatically generated

Architecture Explanation

**Architecture Explanation**

At the core of this application’s design are two threads which run as long as the application is open. The frontend thread “ui thread”, controlled by the overarching MainWindow class, and the backend thread “manager thread” controlled by the overarching manager class. The manager class creates and commands hardware classes which utilize low level APIs from the device manufacturers to control the hardware. The MainWindow class instantiates ui classes which provide the layout and functionality of the user interface. The UI classes include widgets that are populated into the MainWindow’s tabs, and pop up dialogs, which are created by the MainWindow class in response to events.

Backend classes should never call frontend methods directly and vice versa, and in general methods from one thread should not call methods from another thread. Doing so can lead to a host of issues from race conditions to crashes. The solution is to use pyqtSignals and pyqtSlots. In a nutshell, emitting a signal is used as an indirect way of calling a function (the slot). Just like a function call signals contain zero or more parameters, which are passed to a special function called a slot. When one QObject emits a signal, and the signal has been connected (i.e. another QObject is “listening”), the slot function in the recipient class is called with the signal’s parameters.

Sequence of Events

**Sequence of Events**

Startup

**Startup**

1. Main.py begins running
2. Create QApplication
3. Create and show  MainWindow
4. Load system info and config files and populate UI with them
5. Launch a password prompt
6. Begin the manager thread and connect all devices

Scripts

**Scripts**

1. User clicks load script
2. (MainWindow.load\_script\_clicked) QFileDialog launches asking the user to choose a script file. The path is emitted as a signal.
3. (Manager.load\_script) The path is received by Manager, which aborts the current script if there is one and sets the step index to -1. It loads the script file and parses it into lists with length equal to the total number of actions to be completed, including repetitions but not including the begin loop or end loop tasks.
   1. task\_names: a list of strings containing the task names (and repetition number)
   2. task\_args: a list of dictionaries containing the parameters for each task, to be passed to the method for each script action in manager
   3. task\_execution\_order: a list containing the task number of each step in the script, and if it is a looped step the loop number and element number as well.
4. Manager emits a parsed list of dictionaries containing script info (containing loop items but not repeated steps loop iterations)
5. (MainWindow.visualize\_script) script is displayed in a QTreeWidget (script data is now accessible to the script editor as well)
6. User clicks “Run script”
7. (Mainwindow.run\_button\_clicked) buttons are disabled, serial number is read from the UA, and a dialog appears prompting the user for more metadata.
8. (PretestDialog class) a TestData object is created to package test metadata.
9. (PretestDialog.ok\_clicked) metadata is emitted as a signal containing the TestData object.
10. (Manager.test\_metadata\_slot) the Manager’s test\_data attribute is cleared and then populated with the metadata sent by the PretestDialog.
11. (Manager.run\_script) Flag variables like currently\_scripting are set so that the manager’s run loop knows to begin script execution in its core run loop.
12. (Manager.advance\_script) Step index is increased by 1 and run\_script\_step is called
13. (Manager.run\_script\_step) The step with the current script index is executed using task\_names[i] to identify which function to execute and task\_args[i] as the sole parameter, containing all necessary arguments for the function
14. Steps 12 and 13 are repeated until the end of the script is reached, the user clicks “abort after step” or “abort immediately”, the user closes the window, or an interrupt or pass/fail action aborts the script.

Abort Immediately Button Clicked

**Abort Immediately Button Clicked**

1. MainWindow.abort\_immediately\_button.clicked signal triggers manager.abort\_immediately slot
2. (manager.abort\_immediately)Script control variables like step\_index, and currently\_scripting are reset, step number of zero is emitted to the script visual in the UI.
3. (manager.abort\_immediately) abort\_immediately\_var is set to true.
4. (one of manager’s script methods) abort\_immediately\_var is checked by an if statement and the method returns false via its continue variable, meaning to stop the current script action and the higher level methods that called it (if any)
5. The higher level methods (if any) check the continue boolean returned by the lower level method and if it is false they return False as well
6. The script does not continue because currently\_scripting is false

Abort After Step Button Clicked

**Abort After Step Button Clicked**

1. MainWindow.abort\_after\_step\_button.clicked signal triggers manager.abort\_after\_step slot
2. (manager.abort\_immediately)Script control variables like step\_index, and currently\_scripting are reset, step number of zero is emitted to the script visual in the UI.
3. On the next iteration of the run loop, the script is not advanced because currently\_scripting is false.

Summaries of Complex Script Actions

**Summaries of Complex Script Actions**

Measure Element Efficiency RFB

**Measure Element Efficiency RFB**

1. Unpacks variables from a dictionary and typecasts them as necessary
2. Create an RFBData object to store the raw and analyzed data captured by this action
3. Configure the AWG to the desired frequency and amplitude in continuous mode.
4. Select the desired UA channel using the relay in the power module box
5. Create and start an RFBDataLogger thread to capture data from the forward and reflected power meters, as well as the RF Balance for the desired amount of time
6. In a loop with a specified number of iterations, turn the ultrasound output on and off at specified intervals (by turning the AWG output on and off), while continually retrieving data from the RFBDatalogger and emitting it as a signal to be visualized in the rfb tab in the UI
7. Stop the rfb logger and sensor threads
8. Perform data analysis on the completed data (see data analysis documentation)
9. If the data is invalid or has an abnormally high standard deviation, trigger an interrupt action which may prompt the user if they wish to re-run the test, abort automatically or retry automatically (depending on the “Interrupt action” parameter in the config file)

Find Element ‘n’

**Find Element ‘n’**

1. Unpacks variables from a dictionary and typecasts them as necessary
2. Configures the AWG to the desired frequency and amplitude in toneburst mode.
3. Configures the timebase of the oscilloscope
4. Rotates the UA to face the hydrophone
5. (if enabled) Loops through a given range of X coordinates
6. Captures a hydrophone waveform (or the RMS value only) at each coordinate
7. Moves the UA to the X coordinate with the highest voltage squared interval (or rms voltage)
8. (if enabled) Loops through a given range of R coordinates
9. Captures a hydrophone waveform (or the RMS value only) at each coordinate
10. Stores the element X and R coordinates in the element\_x\_coordinates and element\_r\_coordinates variables in manager so they can be referenced in later script actions and saved to disk

This method cycles the ultrasound output from the ultrasound applicator on and off for specified intervals while continuously capturing data from the forward power meter, reflected power meter, and radiation force balance. The readings are captured from all 3 sensors at the same time using 4 additional threads directed by using a RFBDataLogger object.

The refresh\_rfb\_tab helper method scrapes data from the RFBDataLogger and sends it as a signal to be displayed in the RFBTab of the UI. Finally, this method analyzes the data (stored in a RFBData object) and sends it to the FileSaver to be saved.

Organization of Files

**Organization of Files**

Top Level Files

**Top Level Files**

The three highest level python files are main.py, the script which directs the sequence of events upon application launch, MainWindow.py, the parent of all UI elements, and central frontend class, and manager.py, the central backend class which creates and directs hardware devices. All three of these classes are located in the root of the repository.

User Interface files and classes:

**User Interface files and classes:**

All the graphical elements of this program were designed in Qt designer, which outputted .ui files specifying various aspects, attributes, properties, behaviors, and contents of each individual window/prompt/dialog. We then subsequently converted the .ui files to python files with the command: “*python -m PyQt5.uic.pyuic -x <filename.ui> -o <filename.py>*”. All these UI and python files are in the **Widget\_Library** folder. This allowed us to interact with the individual objects in the UI element to various program files. Specifically, we imported the converted, now Python classes to new classes contained within the folder **ui\_elements**. While the Widget\_Library folder has no organized structure, the adapted GUI python classes in ui\_elements are organized nicely. Suppose persistent changes are needed towards any of the GUI elements. In that case, you must either modify its respective adapted python file in ui\_elements or open the UI file in Qt Designer, make and save the changes via Qt Designer, then run the *pyuic* command described above to have the change(s) appear during execution.

More specifically regarding the organized structure of the ui\_elements folder: within it contains four self-evidently named folders: “Dialogs,” “Images,” “script\_editor\_menus,” and “tabs.” The tab folder contains operation logic for the tabs you see in the main window. The images folder contains the PNG images used in the RFB tab’s graph legend. There are also three files in ui\_elements. “*my\_qwidget.py*” is a base class for all the task setting options in the script editor tab. It passes the main logger and enables all buttons to the class that inherits it. “*switch.py*” is a custom-made Qt Object that acts as a toggle switch, since the Qt designer suite didn’t come with one built-in. “*ui\_oscilloscope\_plot.py*” is, as the name suggests, the python class that controls the plots in the RFB and Scan tabs of the main window.

Hardware Classes and Tests:

**Hardware Classes and Tests:**

Hardware classes are contained within the “Hardware” folder from the project’s root directory. Each hardware class represents an interface for programmable equipment. There are both real and simulated hardware classes, so the code can still be run for testing purposes even if some devices are not connected. Both real and simulated hardware classes inherit from one or more abstract classes.

Abstract Hardware Classes

**Abstract Hardware Classes**

The abstract hardware classes are in the Abstract folder. They contain function declarations but not implementations. The base class of all hardware devices is AbstractDevice, which defines the methods and attributes every device should have, namely a connect method, a disconnect method, a get serial number method, and a wrap up method which calls the disconnect method. It also includes a method called log(message, level) which does not need to be overridden and gives all hardware devices the ability to log information to the wtf.log file. Note that this is a separate file from the scriptlog, and is intended mainly for debugging. See also: Log

More specific abstract classes inherit the functions and attributes of AbstractDevice and build upon them, for example AbstractSensor adds a get\_reading() method. Each real and simulated hardware class inherits from the abstract class of the most specific abstract class it realizes. For example MTBalance and SimulatedBalance are both realizations of AbstractBalance, and implement all the required abstract methods thereof.

The purpose of this hierarchy is to define the methods and attributes that external classes can count on a certain type of device to have, without worrying about the implementation details for a specific model of device, allowing for a new hardware class for one type of device to be developed according to the abstract template and swapped out easily, ideally changing only the line of code where the device is imported.

In Manager, notice that the type hints for devices are the name of the abstract class. For example, “Oscilloscope: AbstractOscilloscope” This allows the IDE to give type hints and code autofill suggestions for the classes’ abstract template.

Although the type hints say “abstract”, keep in mind that the instances of the device classes in Manager will be a subclass, either for a real device or a simulated device.

To see the documentation for the underlying abstract class, mouse over the method name in PyCharm. If a docstring for the abstract method exists, It will show up in the mouseover interface, unless it is overridden by the docstring of the child class

One drawback to this approach is that if you use ctrl-click to step into a hardware method from the manager class, it will bring you to the abstract class when you may instead want to see the actual hardware class. To get around this, just go to the desired class in the hardware folder and use ctrl-f to find the desired method.

Simulated Hardware Classes

**Simulated Hardware Classes**

The simulated hardware classes used for testing without real hardware present are in the Simulated folder. Unit tests to test these hardware classes, real or simulated, are in the unit\_test folder. These should be run to ensure no change in the code breaks any of the hardware code logic. The python classes for real hardware are in the root of the hardware folder with self-descriptive names. The hardware folder also contains the dynamic link libraries (.dll) used for the power meter class in the power\_meter\_dlls folder. The interface\_box\_executable folder contains the .exe file that the ultrasound actuator box uses to relay information back and forth between the program and itself. The power meter will require PythonNet to be installed, which makes the maximum Python version 3.8.

Inside the root project folder also lies “local.yaml” and “default.yaml,” both these files act as the “settings” or “preferences” of the application, allowing the user to set various variables and flags that the source code will refer to. The program will first look in local.yaml to see if the setting value is set in there, if not, it will then check the default.yaml file.

Inside the manager python class, all the hardware is initialized and signals that allow the manager thread to communicate to the main window thread are established. A logger is set, and the manager then waits for the user to load a script or issue a command via the main window. Some of the hardware classes are given a reference to the configuration files described earlier in the manager class. The main window class concurrently initializes all the tab python classes and signals to communicate to the manager thread. It uses signals from the manager to toggle various hardware monitoring indicators in real-time while the user is on the main screen.

Real Hardware Classes

**Real Hardware Classes**

Hardware classes for real programmable devices begin by inheriting from an abstract class and implementing all abstract methods. In general, they must also import a library made by the device’s manufacturer for interfacing with said device. Programming manuals for these devices can be found online or in the “\Hardware manuals” folder in the repository. For predictable performance between different devices, the behavior of any method that overrides an abstract class should obey the docstring of the underlying abstract method. In addition, they should use variables and class attributes from the underlying abstract classes whenever possible, and strive to use them in a sensible and self-explanatory way. This is to allow future programmers to program with hardware classes from external classes such as manager without needing to know all of the implementation details or syntax of the underlying hardware class. If a piece of hardware is replaced or upgraded, for example the transition from Galil motion controllers to Parker motion controllers, the two classes should realize the same abstract class and behave as similarly as possible. That way, once the new hardware class is implemented, the only thing that needs to be changed is the code in Manager.add\_devices() that imports and instantiates the object.

Other Files

**Other Files**

definitions.py:

**definitions.py:**

The definitions.py file initializes the root directory of the project, config paths, the power meter dll, and the system info ini path. All of these values are automatically generated, so you do not need to worry about changing them, since they are located in the project folder and python has the ability to find the directory it’s placed in. The directories are also platform agnostic (i.e. works with Windows, Mac, Linux, etc.), though the application has predominantly been tested with Windows 10 and Windows 11 only.

formulas.py:

**formulas.py:**

This file contains key scientific and physical formulas for this application so they can easily be viewed and changed if necessary. If any features that rely on scientific formulas are added, the formulas themselves should be implemented in this file.

environment.yml:

**environment.yml:**

This project uses conda to organize the required dependencies. The project will come with a file called environment.yaml which specifies the name of the environment, channels to get packages from, dependent packages lists from both conda channels and pip, and the directory to setup this environment. The README.md will describe how to use this file to set up the conda environment (method 2). Without doing this, if you run the project from an IDE and do not point the IDE to use the conda environment, the IDE will complain about several missing imports that it cannot identify and the project will fail to run.

Systeminfo.ini:

**Systeminfo.ini:**

This file comprehensively initializes hardware information that pertains to the wet test fixture. These values are then neatly displayed in the “System Info” tab of the main window.

Utilities folder:

**Utilities folder:**

This folder contains modularized aspects of the project that pertain to a particular objective, such as loading configuration data, saving information to a file, etc. These python classes are imported into several other classes that need them. You may see what classes are using these python files in PyCharm by selecting one of these files and pressing alt+F7.

Key Classes Explained

**Key Classes Explained**

Manager

**Manager**

The manager thread serves several central key functions as the executive class for the backend of the application.

First, it takes in a .WTF script file and parses its contents into various local data structures, such as a list of attribute-value pairs and a list of task names representing the sequence of all all actions to be performed in the script in the order they will be executed, as well as a list of dictionaries representing all of the parameter names and their values for each action. When a script loops over elements, these lists include entries for each loop iteration but do not include entries for the beginning and end of the loop itself. The script parser can even detect if the “Element” value within a loop is a static element number instead of the “Current” dynamic value usually used in looped tasks, and automatically correct it for you if desired.

Second, Manager manages the order of execution in the script process, including when tab switches should occur, when buttons/inputs should be enabled/disabled, and when dialog boxes occur.

Third, Manager contains the sequential code for each of the script actions themselves, each of which accepts a single dictionary containing all of the necessary parameters, unpacks them, and performs the step by step sequence associated with that script action.

The main window class mostly contains methods that the manager calls upon to change GUI elements. For example, if the manager wants to disable a certain button, a method written in the main window class (since it has direct access to those elements) will have a pyqtSlot decorator above it and the manager will use its pyqtSignal to toggle those elements on or off. The main window also initializes all the outer layer tabs. The outer layer tabs may have other tabs or multiple layouts that are managed within their own nested classes.

MainWindow

**MainWindow**

As the central class controlling the user interface, MainWindow is responsible for instantiating all of the tab widgets that appear within the application’s main window, as well as all of the dialog boxes that appear when certain events happen in the software’s backend classes. It is responsible for disabling the buttons in the user interface when the application is busy and re-enabling them when the process is complete (this involves calling the set\_buttons\_enabled method in all of the tab widgets)

MainWindow inherits its layout from the window\_wet\_test.py file in the Widget\_Library, which is generated programmatically based on the .ui file with the same name. This QTDesigner file contains promoted widgets for each of the tabs, which means that although the tabs will appear empty in the QTDesigner file preview, all of the tab widget classes will be instantiated and added to the MainWindow when it calls the self.setupUi method. Note that the tab widget classes that are added to the mainwindow are not the .py files in the widget\_library folder, rather they are the files from ui\_elements\tabs folder, which create a layer of functionality on top of the QTDesigner files (see QTDesigner section for more details).

MainWindow triggers actions in Manager, for example when load\_script is clicked it gets the file path using a QFileDialog, and then emits it as a signal to the Manager class which loads and parses it. The run button causes a similar chain of events, except instead of the MainWindow emitting a signal, MainWindow instantiates and shows a custom dialog, connects its signals to Manager, and when the dialog is filled out and dismissed it emits a signal to Manager, triggering the start of the script.

Buttons such as “Abort immediately”, “Abort after step”, “Insert UA”, and “Retract UA” are connected more or less directly to backend classes, so despite being in a backend class their code actually runs in the UI thread. This is permissible because the functions either run quickly or contain one or more self.app.processEvents() calls, ensuring that the UI remains responsive. See example 2 for more details.

MainWindow passes the Manager hardware classes to its child widgets and dialogs so that they can directly communicate using signals. It also connects signals from the Manager and hardware classes to UI elements to display feedback to the user.

Finally, MainWindow visualizes the current script, which it receives as a list of dictionaries emitted as a signal by Manager, by populating it into a QTreeWidget. This QTreeWidget is an object shared with the ScriptEditor widget, which has the ability to modify it. Note that when the script is modified it no longer matches the copy in Manager, so must be saved (and automatically reloaded) before it is run.

Programming Guidelines and Best Practices

**Programming Guidelines and Best Practices**

Type Hinting

**Type Hinting**

The flexibility of python as a dynamically typed language means that, unless otherwise specified, the input(s) of a function can be any type, and the output(s) can be anything, including nothing. On one hand this can be more convenient and allow the programmer to do things that could not be done in a statically typed language, but on the other hand this flexibility can mask eventual TypeErrors that can occur in hard-to-detect edge cases. For example if a function usually returns a float but occasionally returns a none if a rare error occurs, this can cause errors in other parts of the code that assume the function returns a float. The solution to this is to specify the intended types of a function’s inputs and outputs whenever possible.

Here is an example:



The capital names that appear yellow are imported from the typing library, which includes useful features like Union[...,...] to specify that a parameter can be one of multiple different types. If a function returns more than one value, use Tuple[...,...] to package the types of each returned variable.

Python will not stop you from running the code if these hints are violated, IDEs such as pycharm will flag the violations with strong warnings, which helps to prevent errors before they appear.