Polo

Release 0.0.5

Ethan Holleman

Jul 23, 2020

CONTENTS:

Polo is a python GUI build using the Qt library for high throughput crysta.lization screening users.

CHAPTER

ONE

ABOUT

1.1 Background

One of the largest hurtles to obtaining X-ray diffraction data from biological samples is growing large, high quality crystals.

Currently, there is not way to reliably predict successful crystallization conditions based on protein sequence alone and so high-throughput approaches are very appealing. High-throughput crystallization screens test a large chemical space using hundreds of different crystallization cocktails at the nano-drop scale. Successful conditions can then be scaled up and optimized to grow larger crystals.

The high-throughput crystallization screening center at the Hauptman-Woodward Medical Research Institute provides this high-throughput screening service to users; offering 1536 condition screens for both soluble and membrane protein samples. Each plate is imaged over a period of two months in using both visible light microscopy and UV-TPEF photography.

This high-throughput produces a large volume of images that must be sorted through in order to pick out the best condition; a task that can be very tedious and repetitious.

In 2019 Bruno *et al* published Classification of crystallization outcomes using deep convolutional neural networks which included a CNN model that could accurately classify crystallization screening images, opening the door to automating this process. The MARCO model has been used in large scale projects such as the xtution database but has not been utilized in a average-user oriented graphical program.

Polo is therefore designed to incorporate the benefits of the MARCO model and integrate the functionality of established crystallization image labeling software such as MacroscopeJ to create a GUI targeted for HWI and other high-throughput crystallization screening users that incorporates all the tools needed to go from raw crystallization images to designing optimization screens without the need to install any dependencies.

1.2 Features

1.2.1 Automatic Image Classification

Using the MARCO model Polo can cut down the time you spend looking through your crystallization images by identifying wells likely to contain a protein crystal. This can reduce the total number of images that need to be considered from thousands to hundreds.

1.2.2 Multiple Image View Modes

Polo allows you to view your crystallization images in a variety of ways that make it easier to identify true crystal hits. Images can be viewed individually or in grids of up to 96. If a sample has been imaged at multiple points in time it is easy to create timeline views that allow you to assess the effectiveness of a screening condition over time. Additionally, if your samples have been imaged with photographic technologies outside of visible light microscopy it is easy to compare these images to verify the presence of protein crystals.

1.2.3 Integrated FTP Browser

Polo includes an simple FTP browser that allows you to download image files from a remote server directly into Polo without then need to install other software such as FileZilla. Polo is also packaged with unrar for Windows and Mac.

1.2.4 Data Management

Your image classifications are easily saved and managed via the xtal file format. Xtal files are similar to mso files created by MacroscopeJ and encode your image classifications, MARCO classifications, cocktail formulation and other metadata. In addition, xtal files increase portability by encoding your screening images directly into the file along side your metadata. This allows your classifications to be easily shared with one file to anyone else with Polo on their computer.

Polo also has options to export your runs to csv files without encoding your screening images or to HTML reports for a more visual way to share your results with those who do not use Polo.

1.2.5 Open Source Code Base

Polo is written in Python and is licensed under the GNU 3.X license. This allows for modification and use of any of the Polo source code. If you wish to change modify or extend any of Polo's functionality you are free to do so. Additionally, documentation is available at THIS LINK.

CHAPTER

TWO

INSTALLATION GUIDE

Please visit the GitHub Release page and follow the install instructions for the latest release for your operating system.

More details coming soon.

CHAPTER

THREE

FAQS

A place to include frequently asked questions

CHAPTER

FOUR

USER'S GUIDE

If you have yet to install Polo on your machine, head to the installation guide here.

4.1 Importing and Opening Image Data

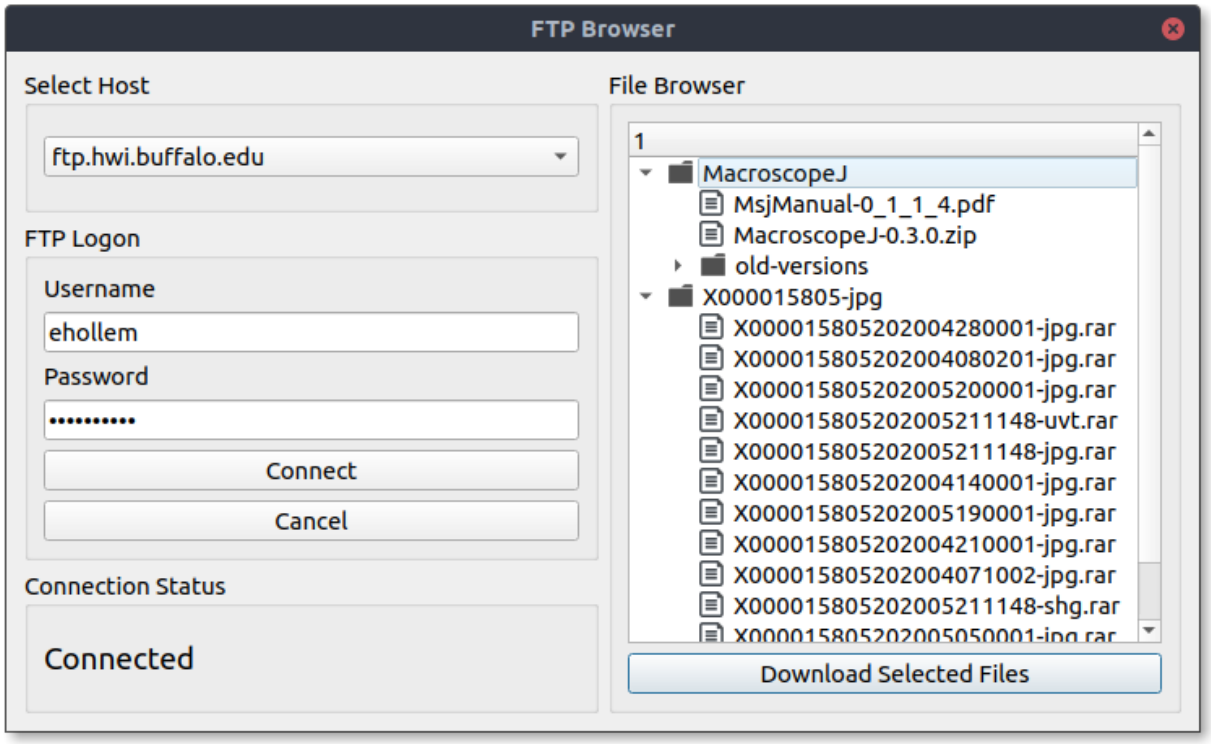
The first step to using Polo is adding your own data. Polo organizes data into "runs", which consist of a set of related screening images. Polo organizes runs into three different categories which are described below.

- HWI Screening Runs
- Non-HWI Screening Runs
- Raw Image Collections

4.1.1 Getting Your Data (via FTP)

If you do not already have your data downloaded to your local machine, Polo includes an FTP file browser which utilizes Python's ftplib package. For HWI users this allows you to download your screening images from the HWI server without leaving the application.

To open the FTP file browser, on the menubar navigate to Import -> Images -> From FTP. Enter your credentials in the new window. Once you are connected to a server files available to download will be listed in the browser menu.

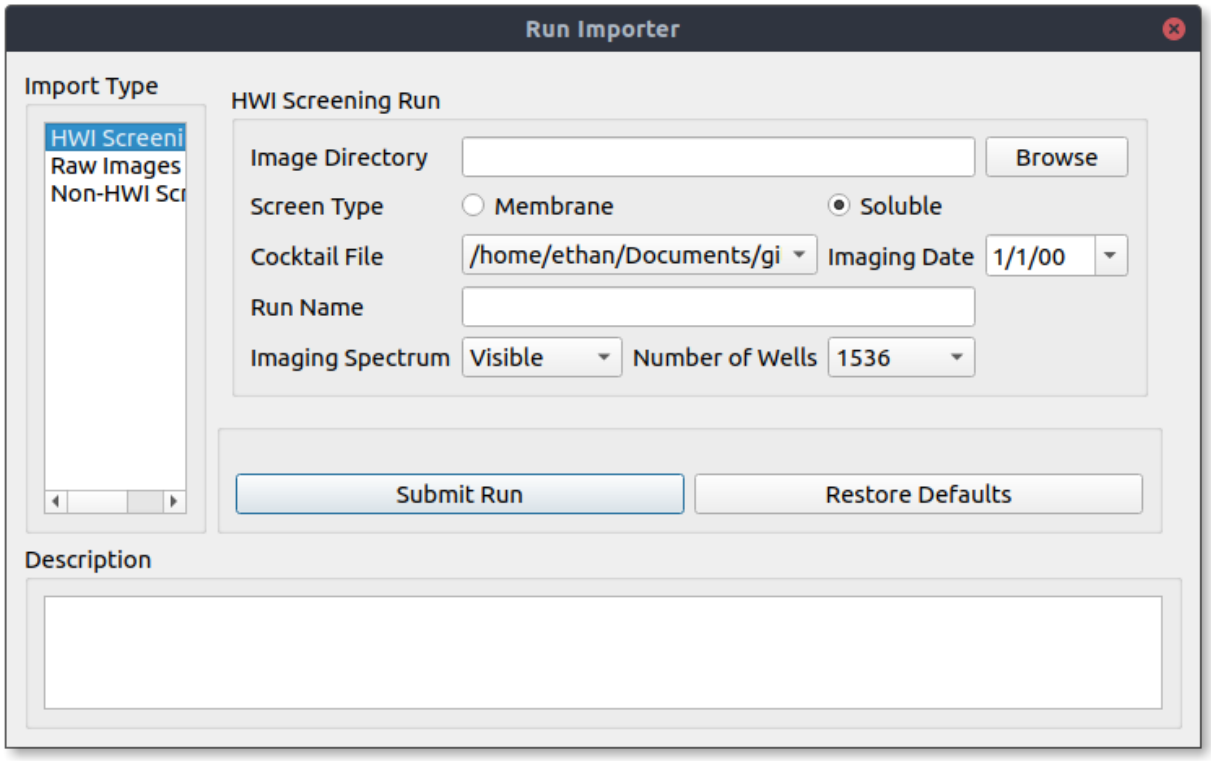


Select the checkboxes next to the files you wish to download, or select all files in a directory at once. Pressing "Download Selected Files" will start the download.

Since these are large files, Polo will download them in the background so you can continue to the program normally. Once all files are downloaded Polo will notify you that your download is complete. Closing Polo before this will result in an incomplete download.

4.1.2 Importing Images from a Directory

Once you have your images on your local machine you can import them into Polo by using the Run Importer tool. It can be opened by navigating to the menu bar and selecting Import -> Images -> From Directory. A window like the one below will then open.



If you are importing HWI screening images select **HWI Screening Run** from the **Import Type** window (Beta testers do this). Select **Browse** to open a file browser and select the directory containing the images you want to import.

Other import types are included for compatibility with other high-throughput protocols but metadata for these import modes is out of the scope of the program. This means runs imported as these types will have fewer available features.

However, since HWI has standard naming conventions Polo will suggest settings for your run including what cocktail file to use, the date of imaging, the spectrum and the number of wells. However, if you want to change any of these settings you are free to do so.

Once you are happy with your import click **Submit Run** to load your images into Polo.

4.1.3 Importing a Saved Run

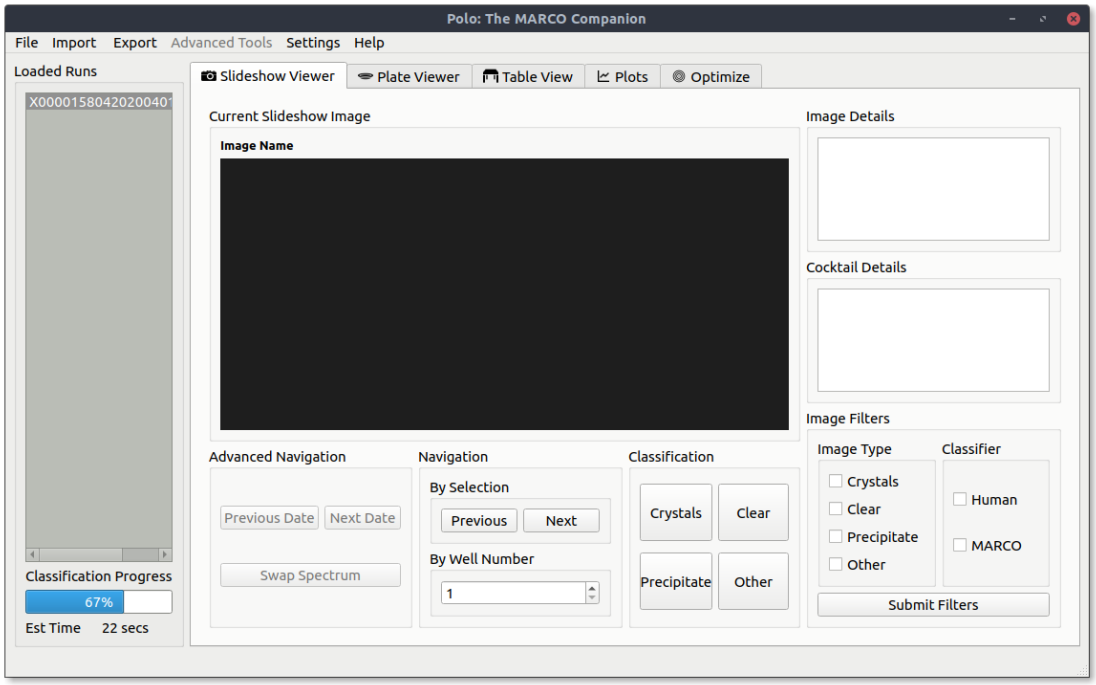
One of the advantages of Polo is the introduction of the .xtal file format. Xtal is a json like file that can store all the data relating to an individual screening run in a single file with no other dependencies. Xtal files store images, classifications, cocktail data and other annotations made while using the Polo program. For more information you can visit Xtal File Format.

If you have xtal file ready for import you can load it into Polo by navigating to the menu bar and selecting **Import** -> **Images** -> **From Saved Run**. This will open a file browser and allow you to select the xtal file you wish to import.

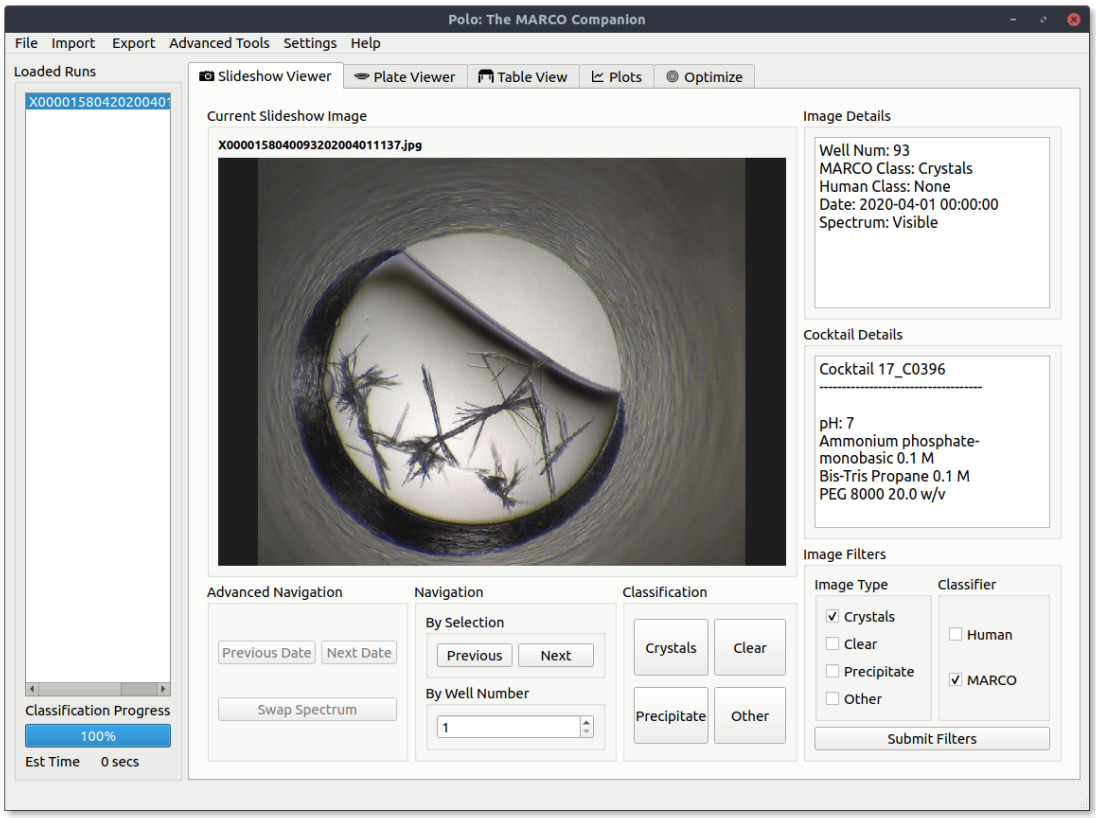
4.1.4 Opening a Run

Once a run has been successfully imported, the run name will appear in the **Loaded Runs** list, like in the image below.

If it is a new run double clicking on the run name will run the MARCO model on the run's images. You can check on MARCO's progress by using the **Classification Progress** bar located just below the Loaded Runs list. Polo will also attempt to estimate the time remaining in your classification job.

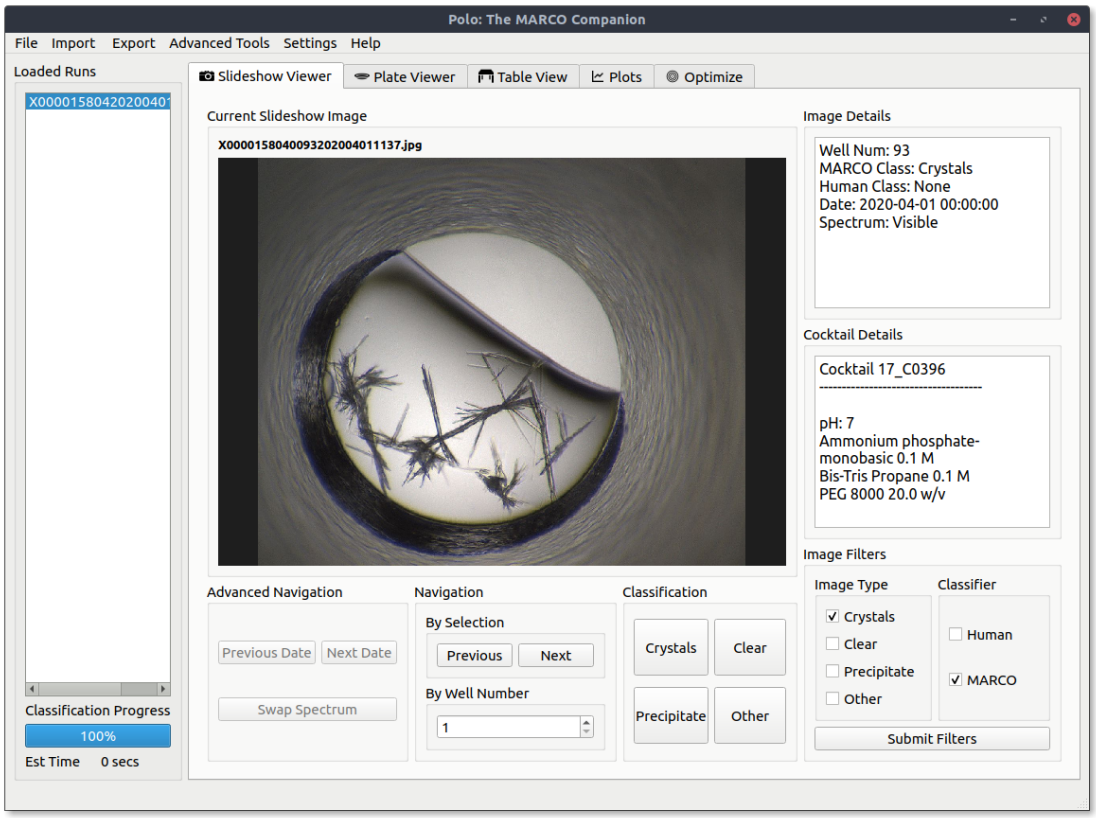


Once a run has been classified you can load it into the current view by double clicking on it again. This will set the selected run as your **Current Run** and all actions will be taken in reference to this run. You can change your current run by double clicking another loaded run in the **Loaded Runs** tab.



4.2 Using the Slideshow View

The slideshow view is the main Polo user interface. It allows you to view your screening images, label them and filter them by MARCO classifications, your own classifications or both.



4.2.1 Basic Navigation

Once you have images loaded in you can cycle through them by pressing the **Next** or **Previous** image buttons under the navigation panel. You can also use the right or left arrowkeys respectively. If you are viewing an HWI screening run you can also navigate directly to a specific well number by entering the well number into the **By Well Number** box.

4.2.2 Classification

Arguably the most important Polo feature is the ability to easily label your screening images. To label the currently displayed image press the button in the **Classification** panel with your desired label. You can classify images as Crystals, Precipitate, Clear or Other. To increase your speed you can classify images using keyboard shortcuts which are listed below.

- 1: Crystal
- 2: Precipitate
- 3: Clear
- 4: Other

Classifying an image will automatically move you to the next image in the slideshow.

4.2.3 Filtering

Using the checkboxes under the **Image Filters** panel in the lower right corner of the window will allow you to filter the kinds of images in your current slideshow. For example if you only wanted to see images that MARCO has classified as Crystal you could check the Crystal box under **Image Types** and MARCO under the **Classifier** panel. If you had checked Human instead only images that you have classified as Crystal would be shown.

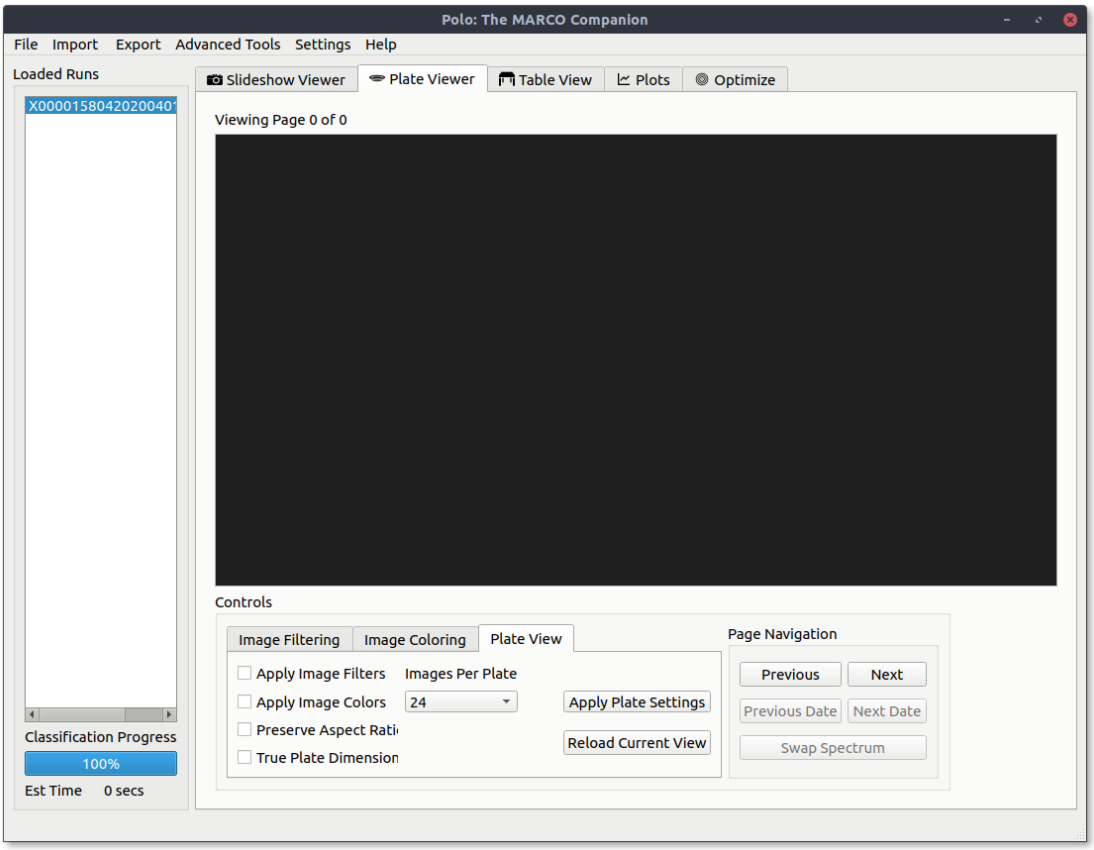
You can reset the slideshow to include all images by selecting all checkboxes or no boxes and pressing submit filters.

4.2.4 Image Metadata

Image metadata will be displayed in the **Image Details** and **Cocktail Details** windows when it is available. Image details will give you basic information about the image currently being displayed, such as well number, imaging technology imaging date and current classifications. If you are viewing an HWI run the chemical conditions the current image was plated in will be displayed in the **Cocktail Details** window.

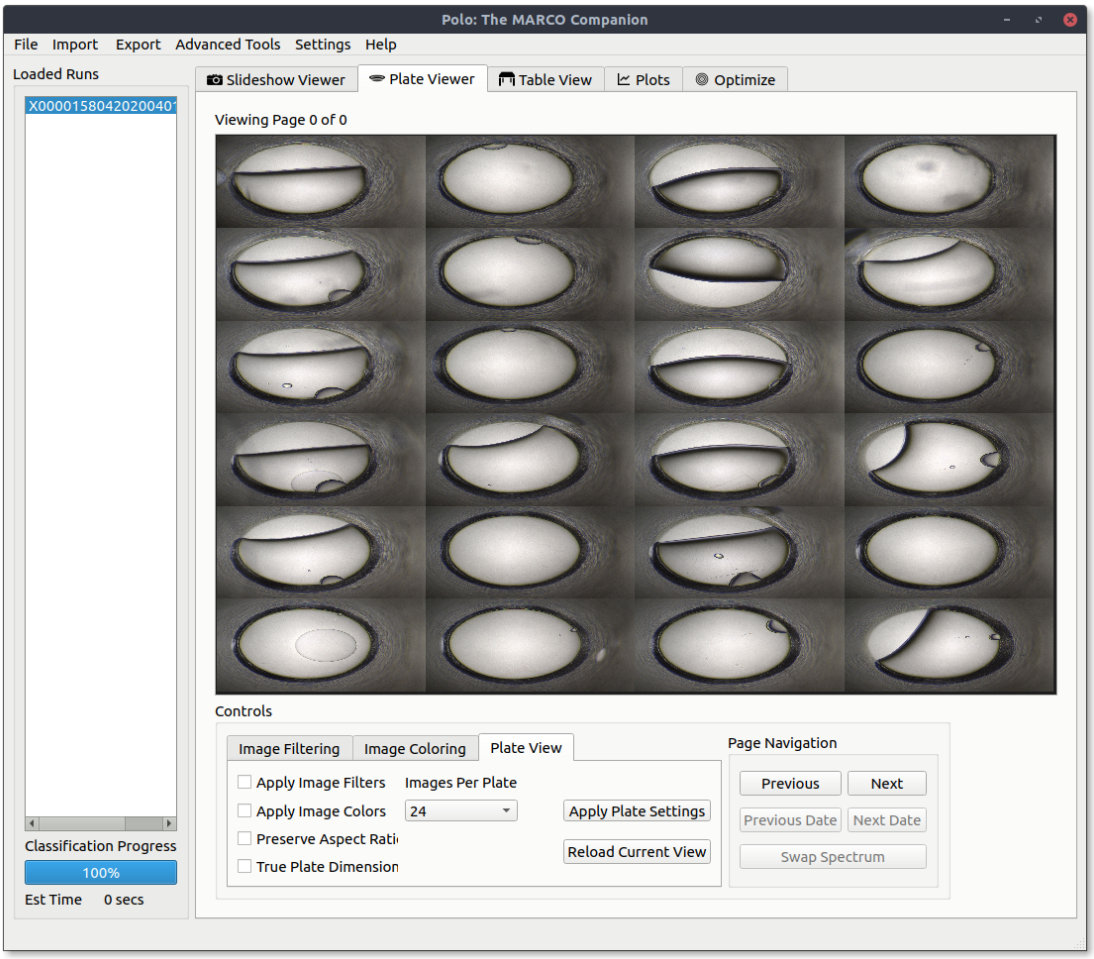
4.3 Using the Plate Viewer

To view multiple images in a grid you can utilize the **Plate Viewer**, which can be found under the **Plate Viewer** tab. When you first open it up, it will look something like the image below.

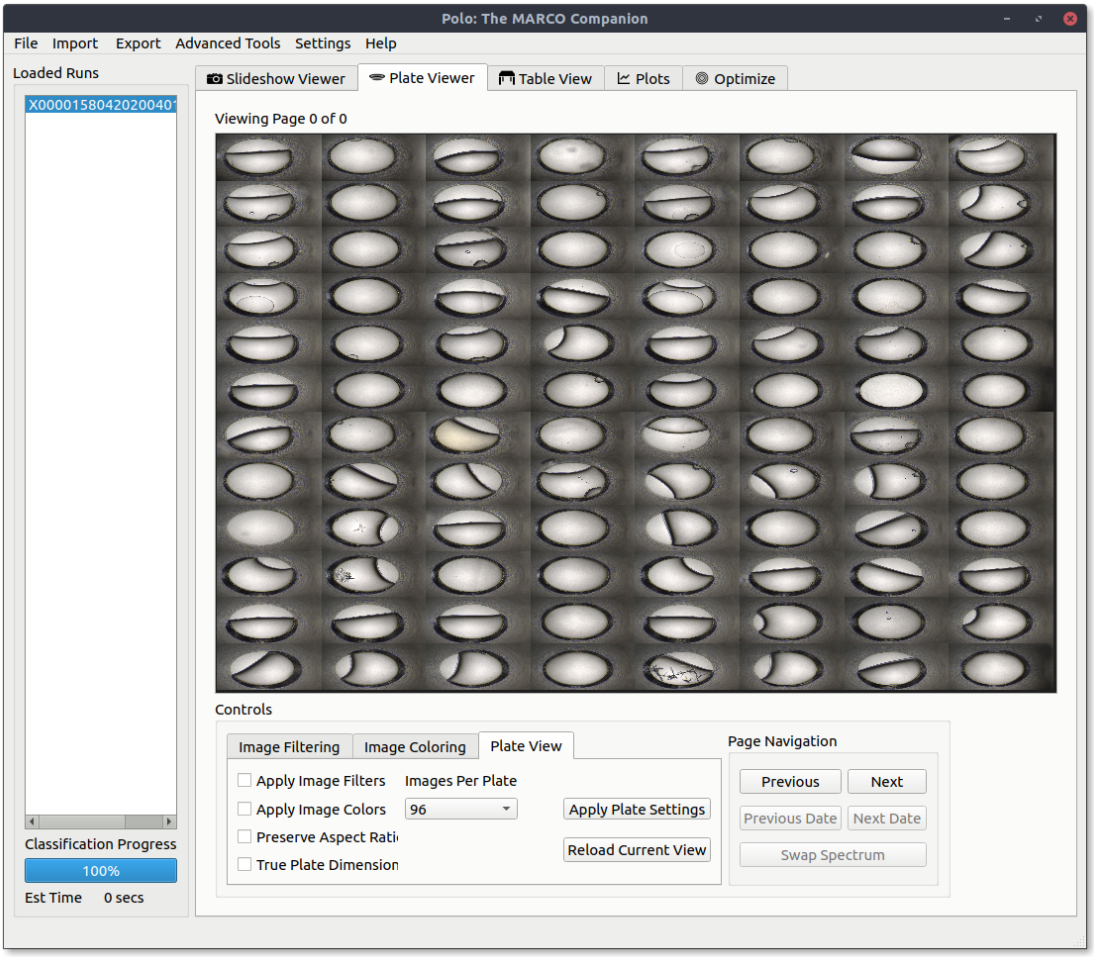


4.3.1 Basic Controls

Assuming you have a run loaded and selected press **Reload Current View** button to load in some images with the current settings.



You can adjust the number of images shown in the grid by using the **Images Per Plate** combo box and pressing **Reload Current View**.



You can navigate to the next or previous view by using the **Next** or **Previous** buttons respectively.

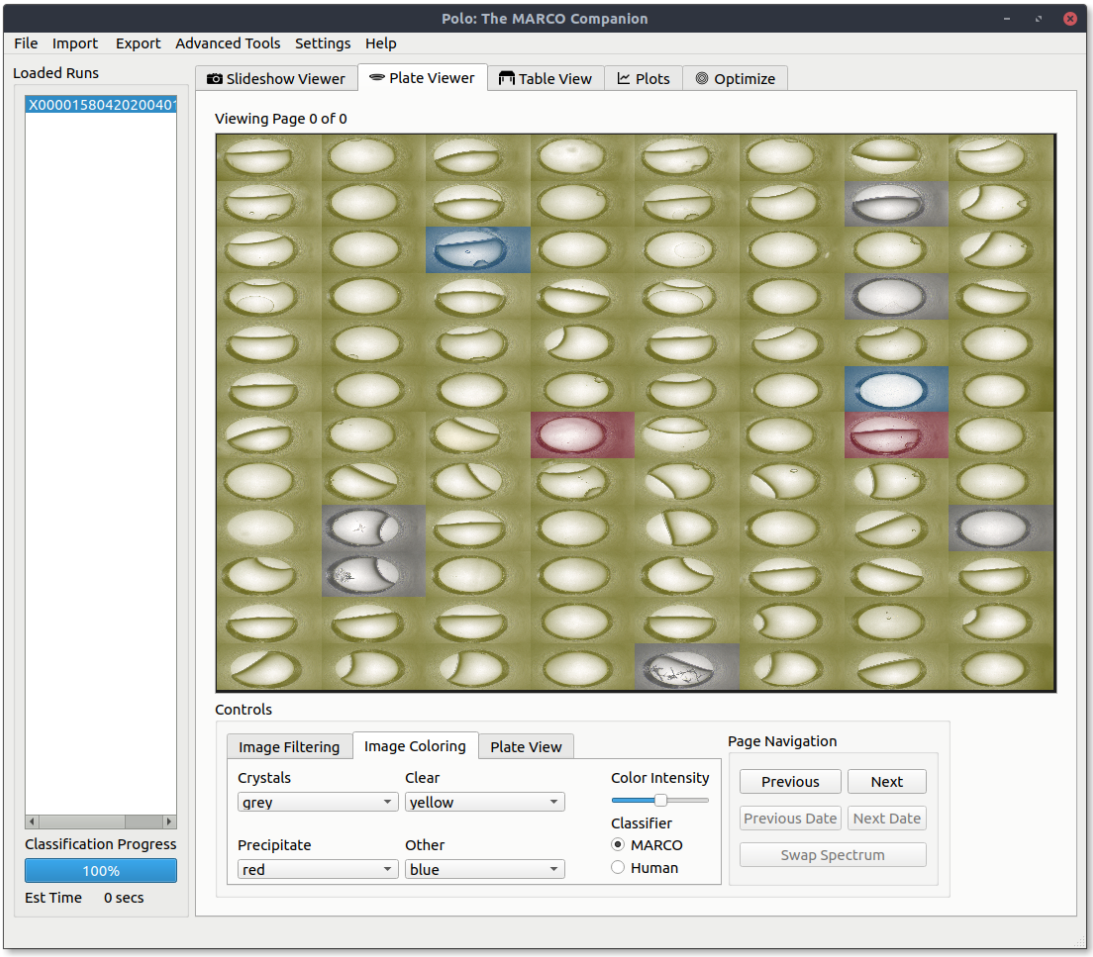
4.3.2 Image Filtering and Coloring

The Plateviewer window allows you to highlight images by either their MARCO classification or the one you have given them. This allows you to find true hits faster. First we will look at how to color images by their classification.

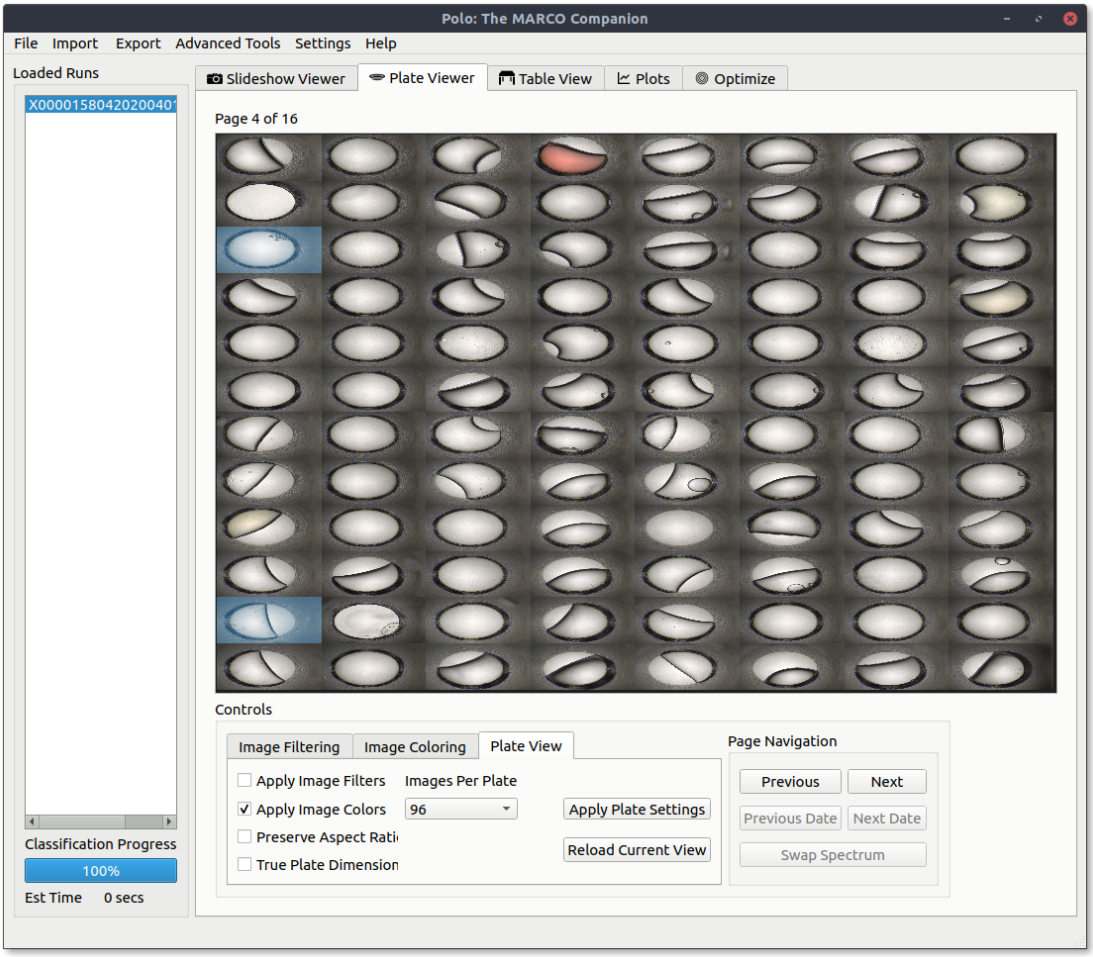
Open the **Image Coloring** tab on the bottom of the Plateviewer window. Use the combo boxes to assign a color to each classification type. The **MARCO** and **Human** radiobuttons tell Polo what classifier to use. After you have picked out a color scheme switch back to the **Plate View** tab select the **Apply Image Colors** checkbox and press the **Apply Plate Settings** button to color your images.

Note: The **Apply Image Colors** checkbox must be selected for colorings to be applied.

A 96 well view with colors applied to all MARCO classifications.

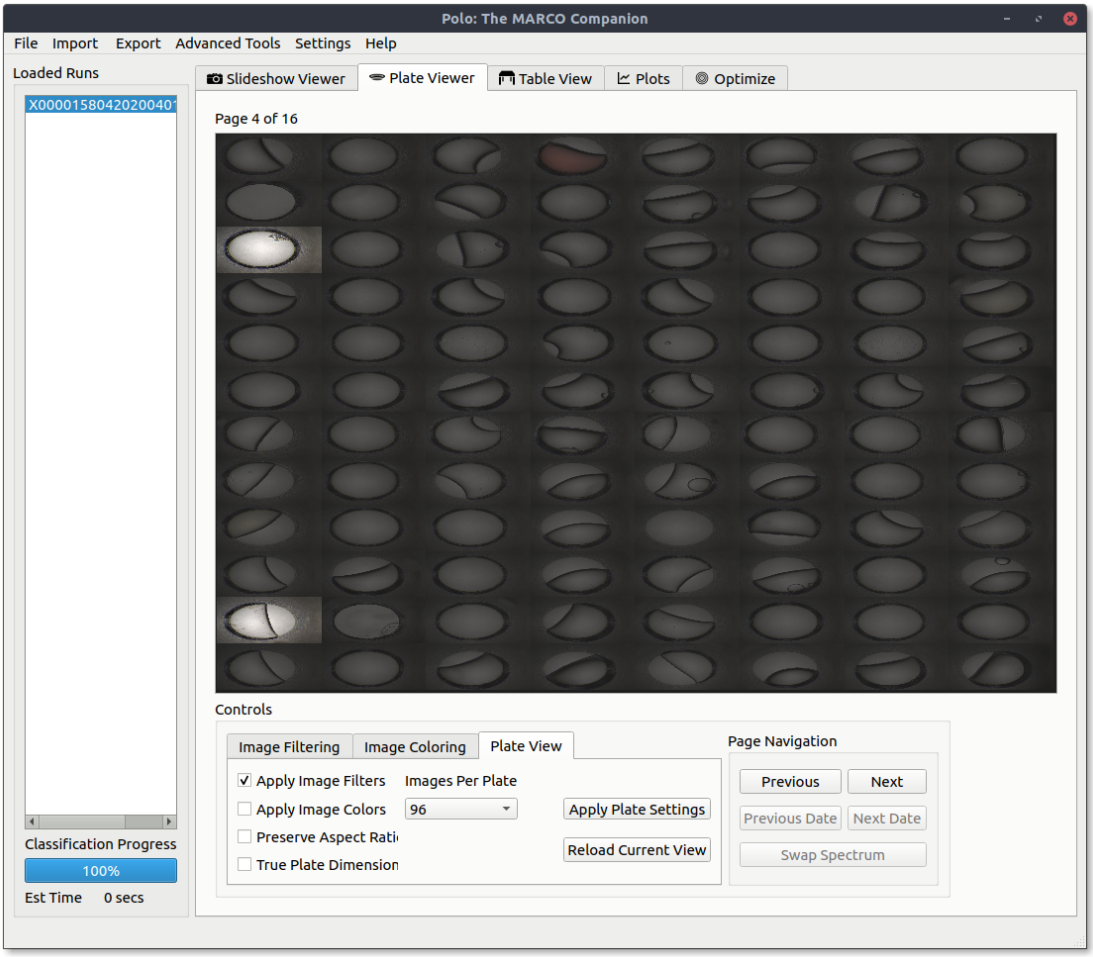


A disappointing 96 well view with only the Crystal classified images colored blue.



Similarly, images can also be emphasized and deemphasized by either their human or MARCO classification. Use the checkboxes under the **Image Filtering** tab to select which images to emphasize. Then switch back to the **Plate View** tab and check the **Apply Image Filters** box and hit **Apply Plate Settings**.

The same view as above but only selecting for MARCO classified crystal images.



4.3.3 Detail View

If you see an image that looks interesting you can select it and it will be opened in a popout window like the one below. Here you can view details about the image and assign it a classification using the buttons in the **Human Classification** panel.

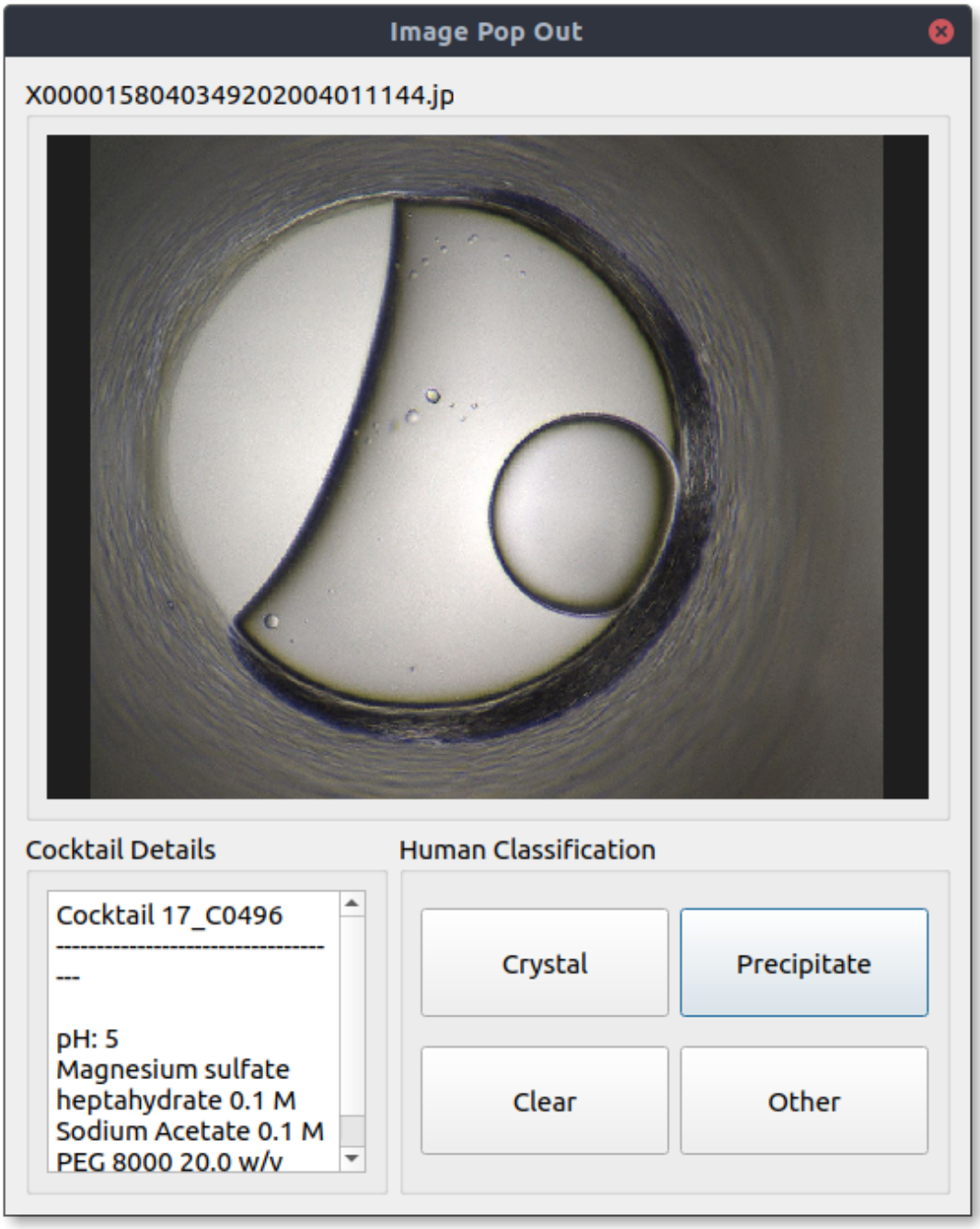
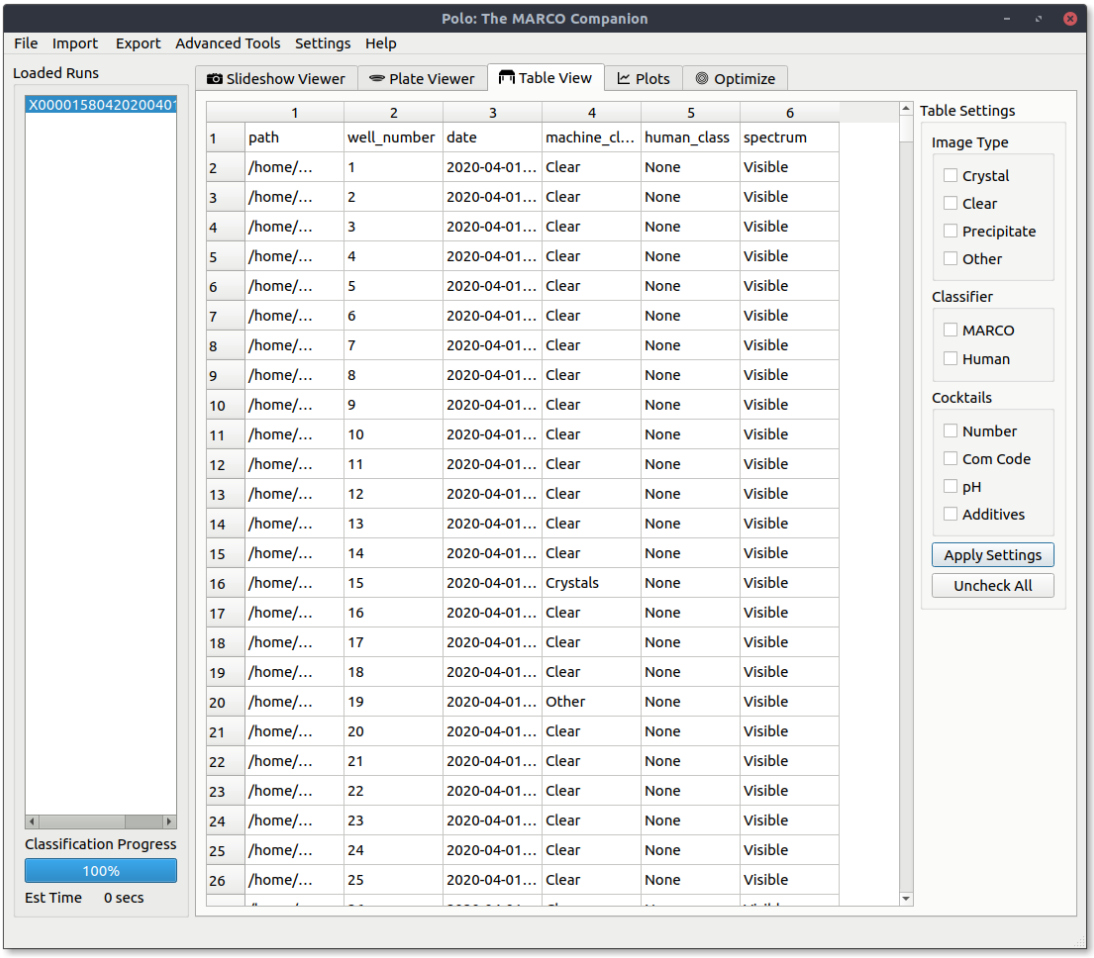


Image Pop Out
X0000158040349202004011144.jp
Cocktail Details
Cocktail 17_C0496
---
pH: 5
Magnesium sulfate heptahydrate 0.1 M
Sodium Acetate 0.1 M
PEG 8000 20.0 w/v
Human Classification
Crystal
Precipitate
Clear
Other

4.4 Using the Table View

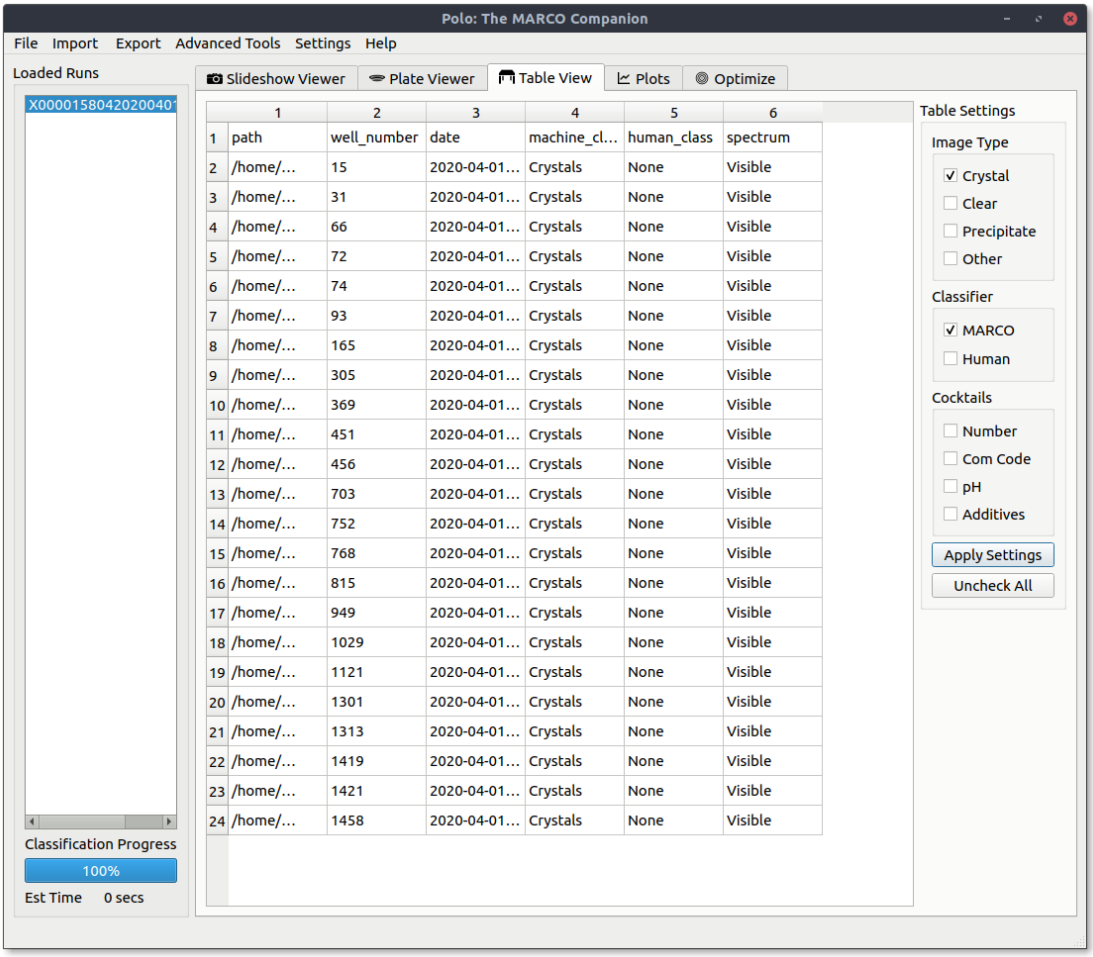
The table view allows you to view a run in a spreadsheet like view and does not display images.



	1	2	3	4	5	6
1	path	well_number	date	machine_cl...	human_class	spectrum
2	/home/...	1	2020-04-01...	Clear	None	Visible
3	/home/...	2	2020-04-01...	Clear	None	Visible
4	/home/...	3	2020-04-01...	Clear	None	Visible
5	/home/...	4	2020-04-01...	Clear	None	Visible
6	/home/...	5	2020-04-01...	Clear	None	Visible
7	/home/...	6	2020-04-01...	Clear	None	Visible
8	/home/...	7	2020-04-01...	Clear	None	Visible
9	/home/...	8	2020-04-01...	Clear	None	Visible
10	/home/...	9	2020-04-01...	Clear	None	Visible
11	/home/...	10	2020-04-01...	Clear	None	Visible
12	/home/...	11	2020-04-01...	Clear	None	Visible
13	/home/...	12	2020-04-01...	Clear	None	Visible
14	/home/...	13	2020-04-01...	Clear	None	Visible
15	/home/...	14	2020-04-01...	Clear	None	Visible
16	/home/...	15	2020-04-01...	Crystals	None	Visible
17	/home/...	16	2020-04-01...	Clear	None	Visible
18	/home/...	17	2020-04-01...	Clear	None	Visible
19	/home/...	18	2020-04-01...	Clear	None	Visible
20	/home/...	19	2020-04-01...	Other	None	Visible
21	/home/...	20	2020-04-01...	Clear	None	Visible
22	/home/...	21	2020-04-01...	Clear	None	Visible
23	/home/...	22	2020-04-01...	Clear	None	Visible
24	/home/...	23	2020-04-01...	Clear	None	Visible
25	/home/...	24	2020-04-01...	Clear	None	Visible
26	/home/...	25	2020-04-01...	Clear	None	Visible

4.4.1 Filtering

Just like the **Slideshow Viewer** and **Plate Viewer** tabs you can filter what data is shown to you in the table view. For those familiar with SQL this is like a SELECT / WHERE statement. Press the **Apply Settings** button to apply your currently selected filters.



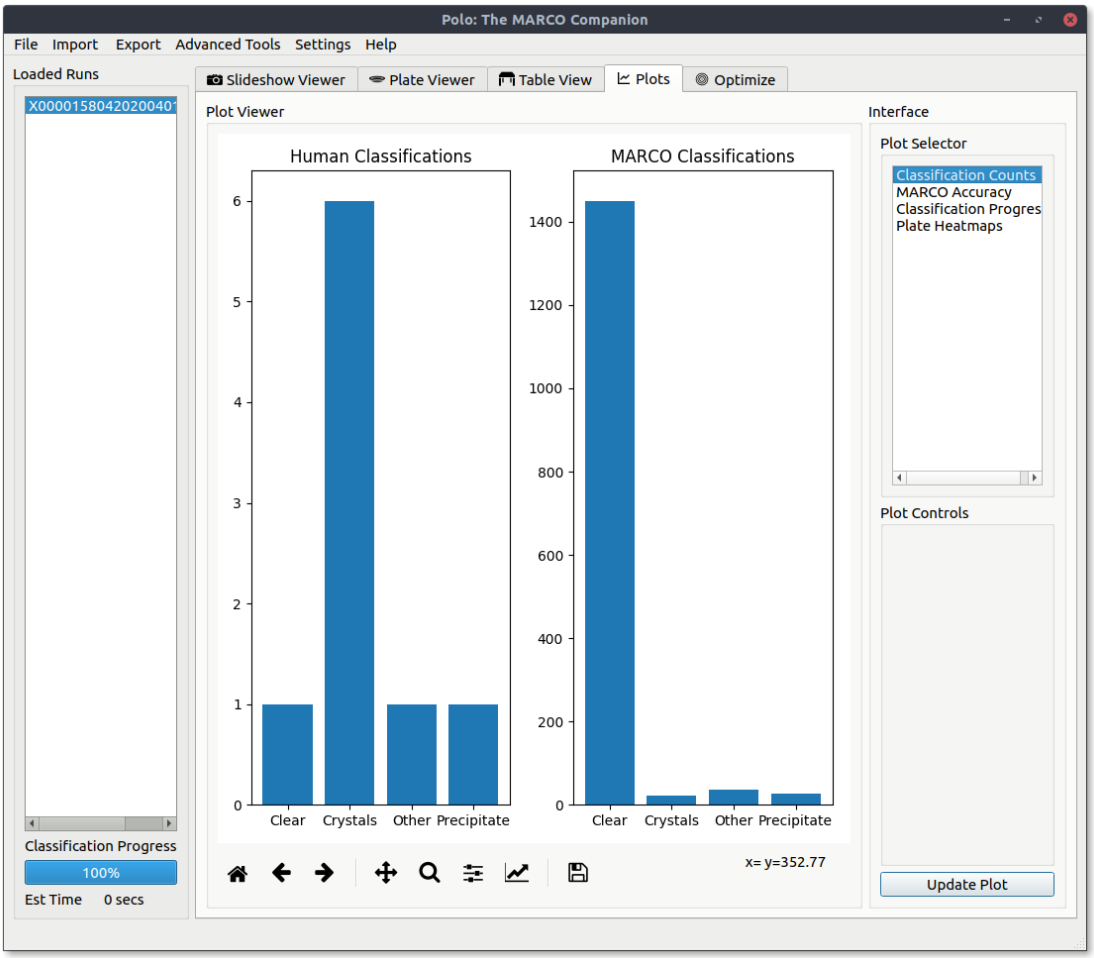
	1	2	3	4	5	6
1	path	well_number	date	machine_cl...	human_class	spectrum
2	/home/...	15	2020-04-01...	Crystals	None	Visible
3	/home/...	31	2020-04-01...	Crystals	None	Visible
4	/home/...	66	2020-04-01...	Crystals	None	Visible
5	/home/...	72	2020-04-01...	Crystals	None	Visible
6	/home/...	74	2020-04-01...	Crystals	None	Visible
7	/home/...	93	2020-04-01...	Crystals	None	Visible
8	/home/...	165	2020-04-01...	Crystals	None	Visible
9	/home/...	305	2020-04-01...	Crystals	None	Visible
10	/home/...	369	2020-04-01...	Crystals	None	Visible
11	/home/...	451	2020-04-01...	Crystals	None	Visible
12	/home/...	456	2020-04-01...	Crystals	None	Visible
13	/home/...	703	2020-04-01...	Crystals	None	Visible
14	/home/...	752	2020-04-01...	Crystals	None	Visible
15	/home/...	768	2020-04-01...	Crystals	None	Visible
16	/home/...	815	2020-04-01...	Crystals	None	Visible
17	/home/...	949	2020-04-01...	Crystals	None	Visible
18	/home/...	1029	2020-04-01...	Crystals	None	Visible
19	/home/...	1121	2020-04-01...	Crystals	None	Visible
20	/home/...	1301	2020-04-01...	Crystals	None	Visible
21	/home/...	1313	2020-04-01...	Crystals	None	Visible
22	/home/...	1419	2020-04-01...	Crystals	None	Visible
23	/home/...	1421	2020-04-01...	Crystals	None	Visible
24	/home/...	1458	2020-04-01...	Crystals	None	Visible

4.5 Using Plot Functions

Polo utilizes the Matplotlib python library to provide a few diagnostic plots to give you more information on your screening run. Once a run is loaded in plots can be viewed by selecting the **Plots** tab.

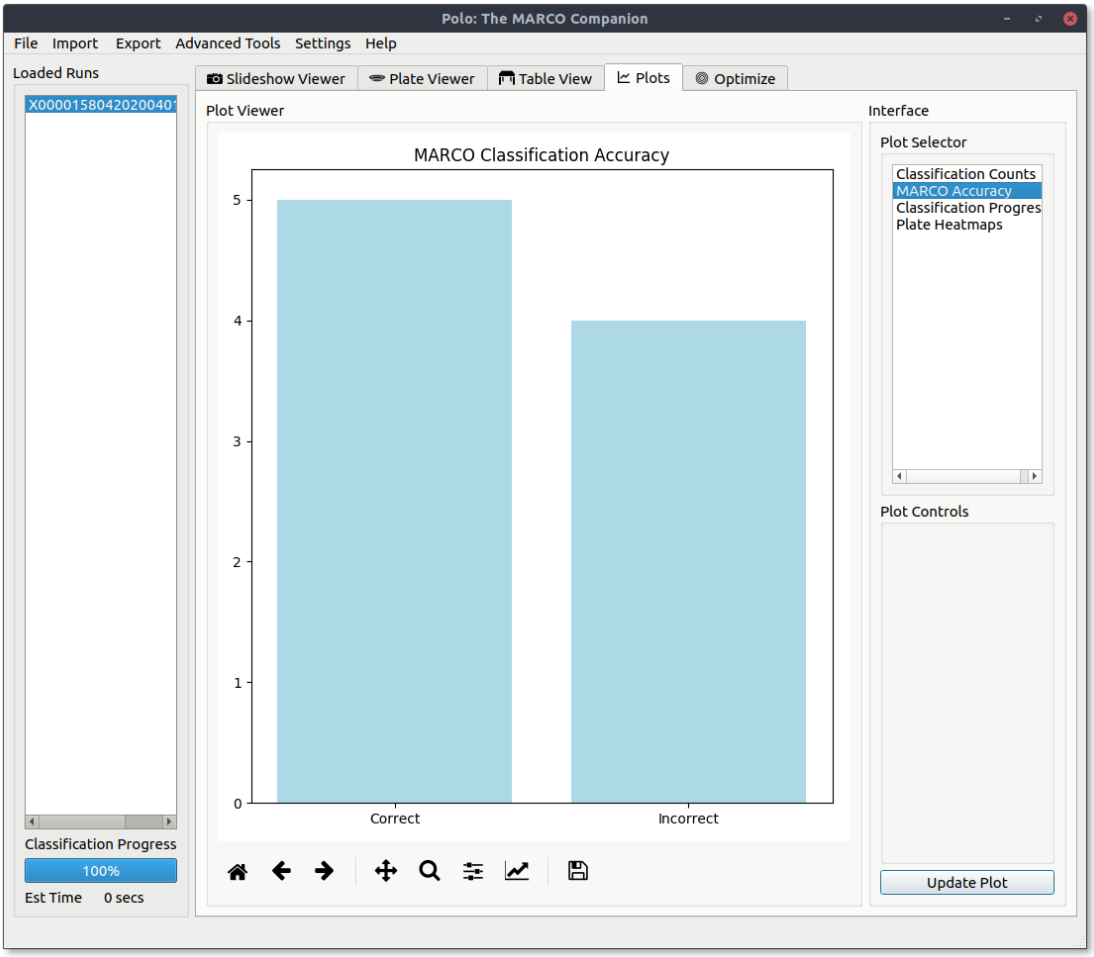
4.5.1 Classification Counts

Visualize human and MARCO classifications by image type.



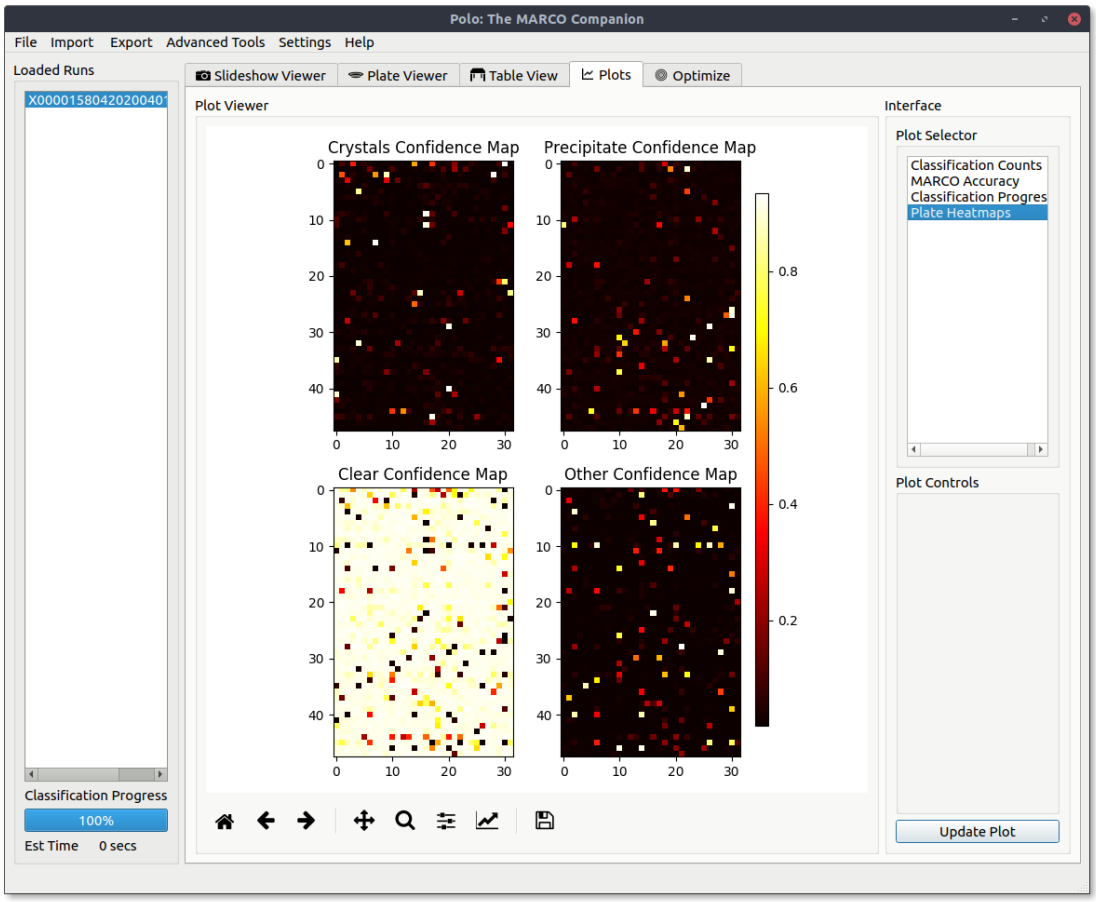
4.5.2 MARCO Accuracy

View a basic bar graph of MARCO classification accuracy. Correct classifications are those where the model applied the same classification as the human.



4.5.3 Plate Heatmaps

View the MARCO model confidence for each image classification across all images in your screening run.



4.5.4 Saving a Plot

If you wish, you can save a plot as an image to your machine by using the plot control icons at the bottom of the **Plots** tab. Select the floppy disc to save the current plot.

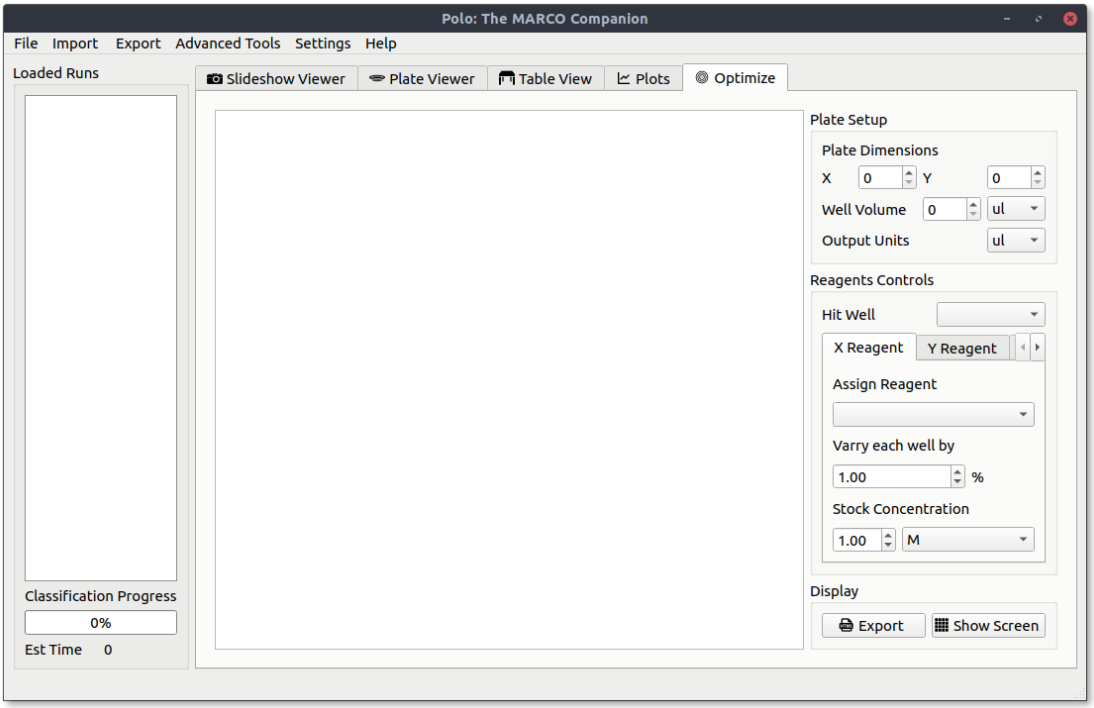
4.6 Using the Optimize Tool

Once you have identified crystal containing wells, you can easily design optimization screens using the optimize tool. The optimize tool automatically creates screens around the cocktail conditions a crystal hit grown in.

Currently all conditions come directly from HWI screening conditions and changing the conditions manually is not currently supported.

For more information on optimization protocols you can read this document provided by HWI,

Reproducing HWI HTCSC crystallization screening hits



4.6.1 Setting Up Your Plate

The first thing to do is to tell Polo what the plate you will be using for your optimization screen. Set the number of wells in your plate by adjusting the plate dimensions.

For example a standard 24 well plate could have 6 wells on X axis and 4 on the Y or vice versa. Which value you assign to which axis is arbitrary as long as you assign the reagent you want to screen for on that same axis.

You should also adjust the well volume by using the **Well Volume** and associated units combo boxes. This volume will be used as the maximum volume for each well.

4.6.2 Selecting Reagents

Once your plate is set up you are ready to select reagents to screen for. Wells you have classified as crystal containing are listed in the **Hit Well** combo box under the **Reagent Controls** panel. Selecting a well will display its containing reagents under the **Assign Reagents** combo boxes of the **X Reagent** and **Y Reagent** tabs.

Note: A reagent assigned under the **X Reagent tab** will be varied on the X-axis of the plate and the reagent assigned under the **Y Reagent** tab will be varied on the Y-axis of the plate.

Since any given plate only has two axis only two reagents can be screened for, but many crystallization cocktail contain three or more reagents. Reagents not assigned to either the x or y reagent will be considered constant and will be present in your final screen but always at the same concentration.

Once you have assigned your reagents set a stock concentration for each reagent and select a percentage to the x and y reagents by. This is always in reference to the hit concentration for the particular reagent.

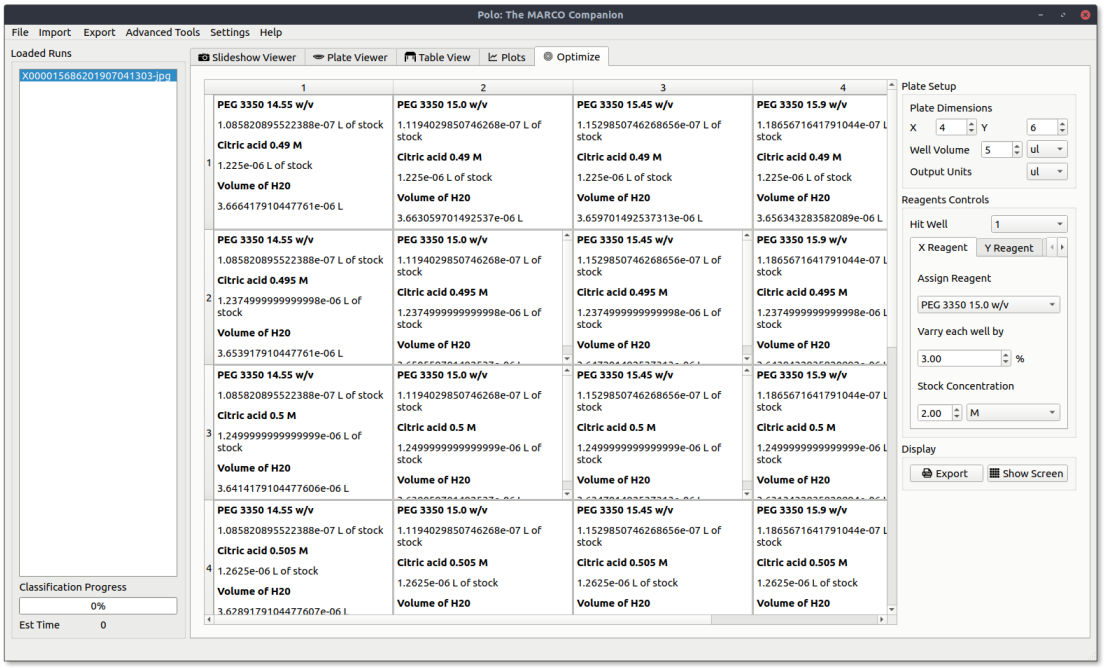
For example if you selected citric acid as your x reagent and it was present in the crystallization cocktail at a concentration of 0.5 M the concentration of citric acid in your optimization screen will be varied in reference to 0.5 M by the

percentage selected in the **Vary each well by** spin box.

Additionally, the hit concentration is always the center most well of the axis. Meaning if there are 6 wells on your plate's x-axis you can expect the concentration of citric acis in the 3rd well to be 0.5 M (using the example above).

4.6.3 Viewing Your Optimization Screen

After everything is set up, press **Show Screen** to display your current optimization screen. It will look something like the image below.



If it does not fill the screen slightly adjust the size of the Polo window and it should snap into place.

4.6.4 Exporting A Screen

After setting up your screen you can export it to an html file to print and / or share with your friends. To do so just hit the **Export** button. This will open a file browser and you can select where you would like to save the file. You can then open the file with any browser to view it or save it as a pdf.

4.7 Advanced Tools

Advanced tools go a bit beyond image classification and viewing and allow you to take advantage of the fact a single screening run will likely be imaged at multiple points in time and with multiple imaging technologies. All tools described below can be found by navigating to the menu bar and selecting **Advanced Tools**.

4.7.1 Time Resolved Runs

Usually, the sample plate will be imaged at multiple points in time. Polo allows you to link these runs together using the **Add Time Resolved** tool under **Advanced Tools**. Polo will automatically determine the order of the runs based on their date and then allow you to navigate between them in both the Slideshow Viewer tab and Plate Viewer tab using the **Next Date** and **Previous Date** buttons.

4.7.2 Adding an Alternative Spectrum

Oftentimes, the same plate will be imaged with a photographic technology besides microscopy in order to verify the presence of crystals in a sample. If you have images taken with such a technology you can identify them as such in the **Run Importer** window (link here).

Then navigate to **Advanced Tools** and select **Add Spectrum** which will open a dialog like the one shown below.

Valid runs will automatically be assigned to an image category based on the spectrum they were assigned at import. Select one run from each category that you would like to link and when finished press **Submit Assignment**.

You will then be able to swap between these images in both the Slideshow Viewer and Plate Viewers. This can be very useful for verifying a crystal is in fact a protein crystal and not salt.

4.8 Saving a Run

Once you have invested time into picking out the best screening conditions you are going to want to save your work and share it with others. The best way to do this in Polo is by saving your current run as a .xtal file.

4.8.1 .xtal File Format

.xtal is a json based format for saving screening runs after they have been loaded into and processed using Polo. It will maintain all of your image classifications and metadata in a single file. Additionally, the actual screening images are encoded directly into the xtal file which makes sharing your data to another Polo user very easy as it only requires sharing the single .xtal file.

4.8.2 Actually Saving Your Run

To save your Run from Polo, navigate to the menu bar and and select **File -> Save** or **Save As**. If you have not saved your current run to a xtal file before a file browser will automatically be opened and you can specify a location and name for your save file.

Note: Do not close Polo while a run is being saved!

After the save is complete Polo will notify you with a popup telling you it is safe to close the program and your save as completed. You can now share your xtal file with anyone else using Polo!

4.9 Exporting a Run

In addition to saving screening runs as xtal files, Polo supports exporting to formats which can be viewed outside of the Polo application. Currently available export options are described below.

4.9.1 HTML Reports

4.9.2 CSV Exports

CHAPTER

FIVE

BETA TESTERS GUIDE

5.1 Thank You!

Thank you for taking some time out of your day to help me test Polo. It means a lot. On this page you will find info on how to get started, where to find test data, what to do with it and how to report any bugs that you find.

5.2 Getting Polo Beta Version

Visit the *Installation Guide* to learn more about getting the latest Polo release for your system. For test releases test data will be packaged along with the installer to help you start using Polo. You also might want to skim over the *Background* section of the About page to get a better idea of what Polo's use cases are. The *User's Guide* goes into more detail about the specific functions of each tool included in Polo.

5.3 What Now?

Now that you have got some data you can start working with it. Here is a list of things to do with your data and links to sections of the documentation on how to do it.

- Open the xtal file: *Importing Images from a Directory*
- Open the folder of images and classify them: *Importing a Saved Run*
- Classify images using the Slideshow Viewer: *Using the Slideshow View*
- Classify images using the Plate Viewer: *Using the Plate Viewer*
- Create an optimization screen: *Using the Optimize Tool*
- Do something unexpected in an attempt to break the program (**Highly Recommended!**)

For more ideas and help getting started check out the *User's Guide*

5.4 I Found a Bug

Thats great! If you have some GitHub knowhow you can report it on the issues on the issues page of the Polo repository or fill out this google form Thanks for your help!

CHAPTER

SIX

POLO PACKAGE

6.1 Subpackages

6.1.1 polo.crystallography package

Submodules

polo.crystallography.cocktail module

class polo.crystallography.cocktail.**Cocktail**(*number=None*, *well_assignment=None*, *commercial_code=None*, *pH=None*, *reagents=[]*)

Bases: `object`

Cocktail instances are used to hold a collection of *Reagent* instances that form one chemical cocktail. Cocktails also hold other metadata including their commercial code, the cocktail pH and the well they are assigned to. Currently, cocktails are only supported for HWIRuns.

Parameters

- **number** (*str, optional*) – The cocktail number, defaults to None
- **well_assignment** (*int, optional*) – Well number in the screening plate this cocktail belongs to, defaults to None
- **commercial_code** (*str, optional*) – Commercial code of cocktail, defaults to None
- **pH** (*float, optional*) – pH of the cocktail, defaults to None
- **reagents** (*Reagent, optional*) – list of reagent instances that make up the contents of the cocktail, defaults to None

add_reagent(*new_reagent*)

Adds a reagent to the existing list of reagents referenced by the :attr:`~polo.crystallography.cocktail.Cocktail.reagents attribute.

Parameters **new_reagent** (*Reagent*) – Reagent to add to this cocktail

property cocktail_index

Attempt to pull out the cocktail number as an integer from the `number` attribute. This property is dependent on consistent formating between cocktail menus that has not checked at this time.

Returns Cocktail number

Return type int

property well_assignment
Return the current well assignment for this Cocktail.

Returns well assignment

Return type int

class polo.crystallography.cocktail.**Reagent**(*chemical_additive=None*, *concentration=None*, *chemical_formula=None*, *stock_con=None*)

Bases: object

Reagent instances represent one specific kind of chemical compound at a specific concentration. Multiple Reagents make up a cocktail. Reagents are generally created from the contents of HWI cocktail csv files which describe all 1536 cocktails and the reagents that compose them in one file. The cocktail csv files can be found in the *data* directory.

Parameters

- **chemical_additive** (*str, optional*) – Name of the chemical reagent, defaults to None
- **concentration** (UnitValue, *optional*) – Concentration of the reagent, defaults to None
- **chemical_formula** (*Formula, optional*) – Chemical formula for this reagent, defaults to None
- **stock_con** (UnitValue, *optional*) – Concentration of this reagent's stock solution, defaults to None

property chemical_formula
The chemical formula for of this Reagent. Not all Reagents have available chemical formulas as cocktail csv files do not include formulas for all reagents. See the setter method for more details on how chemical formulas are converted from strings to *Formula* objects.

Returns Chemical formula

Return type Formula

property concentration
The current concentration of this Reagent. Concentration ultimately refers back to a condition in a specific screening well.

Returns Chemical concentration

Return type *UnitValue*

property molar_mass
Attempt to calculate the molar mass of this reagent. Closely related to the molarity property. The molar mass of the reagent cannot be calculated for all HWI reagents.

Returns Molar mass of the Reagent if it is calculable, False otherwise.

Return type *UnitValue* or bool

property molarity
Attempt to calculate the molarity of this reagent at its current concentration. This calculation is not certain to return a value as HWI cocktail menu files use a variety of units to describe chemical concentrations, including %w/v or %v/v.

%w/v is defined as grams of colute per 100 ml of solution * 100. This can be converted to molarity when the molar mass of the reagent is known.

%v/v is defined as the volume of solute over the total volume of solution * 100. The density of the reagent is required to convert %w/v to molarity which is not included in HWI cocktail menu files. This makes conversion from %w/v out of reach for now.

If the reagent concentration cannot be converted to molarity then this function will return False.

Returns molarity or False

Return type *UnitValue* or Bool

peg_parser(*peg_string*)

Attempts to pull out a molar mass from a PEG species since the molar mass is often included in the name of PEG species. A string is considered to be a potential PEG species if it contains 'PEG' or 'Polyethylene glycol' in it.

Parameters **peg_string** (*str*) – String to look for PEG species in

Returns molar mass if found to be valid PEG species, False otherwise.

Return type float or Bool

stock_volume(*target_volume*)

Attempt to calculate the required amount of stock solution to produce the Reagent's set concentration in the given *target_volume* argument. Stock concentration is taken from the `stock_con` attribute. If `stock_con` is not set or the molarity of the reagent can not be calculated this method will return False.

Parameters **target_volume** (`UnitValue`) – Volume in which stock will be diluted into

Returns Volume of stock or False

Return type *UnitValue* or False

units = ['M', '(w/v)', '(v/v)']

class polo.crystallography.cocktail.**UnitValue**(*value=None*, *units=None*)

Bases: `object`

UnitValues are used to help handle numbers with units. They are not the most robust but help to keep things more organized. UnitValues can be created by either passing values to the *values* and *units* args explicitly or by calling the classmethod *make_from_string()* which will use regex to pull out supported units and values.

classmethod make_from_string(*string*)

Create a *UnitValue* from a string containing a value and a unit. Utilizes the `polo.unit_regex` expression to pull out the units.

```
unit_string = '10.0 M'  # concentration of 10 molar
sv = UnitValue.make_from_string(unit_string)
# sv.value = 10 sv.units = 'M'
```

Parameters **string** (*str*) – The string to extract the UnitValue from

Returns UnitValue instance

Return type *UnitValue*

round(*digits*)

saved_scalers = {'c': 0.01, 'm': 0.001, 'u': 1e-06}

scale(*scale_key*)

Scale the *value* using a key character that exists in the *saved_scalers* dictionary. First converts the value to its base unit and then divides by the *scale_key* argument value. The *scale_key* can be thought of as a SI prefix for a base unit.

```
# self.saved_scalers = {'u': 1e-6, 'm': 1e-3, 'c': 1e-2}
v_one = UnitValue(10, 'L')  # value of 10 liters
v_one = v_one.scale('u')  # get v_one in microliters
```

Parameters **scale_key** (*str*) – Character in *saved_scalers* to convert value to.

Returns UnitValue converted to scale_key unit prefix

Return type *UnitValue*

set_from_string(*string*)

to_base()

Converts the *value* to the base unit, if it is not already in the base unit.

Returns UnitValue converted to base unit

Return type *UnitValue*

property value

polo.crystallography.image module

class polo.crystallography.image.**Image**(*path=None*, *bites=None*, *well_number=None*, *human_class=None*, *machine_class=None*, *prediction_dict=None*, *plate_id=None*, *date=None*, *cocktail=None*, *spectrum=None*, *previous_image=None*, *next_image=None*, *alt_image=None*, *favorite=False*, *parent=None*, ***kwargs*)

Bases: PyQt5.QtGui.QPixmap

Image objects hold the data relating to one image in a particular screening well of a paricular run. Images encode the actual image file as a file path to the image if it available on the local machine, as as a base64 encoded image held in memory or both.

Parameters

- **path** (*str or Path, optional*) – Path to the actual image file, defaults to None
- **bites** (*str or bytes, optional*) – Image encoded as base64, defaults to None
- **well_number** (*int, optional*) – Well number of the image, defaults to None
- **human_class** (*str, optional*) – Human classification of this image, defaults to None
- **machine_class** (*str, optional*) – MARCO classification of this image, defaults to None
- **prediction_dict** (*dict, optional*) – Dictionary containing MARCO model confidence for all image classifications, defaults to None
- **plate_id** (*str, optional*) – HWI given unique ID for plate this image belongs to , defaults to None
- **date** (*Datetime, optional*) – Date this image was taken on, defaults to None
- **cocktail** (Cocktail, *optional*) – Cocktail assigned to the well this image is of, defaults to None
- **spectrum** (*str, optional*) – Keyword describing the imaging tech used to take the image , defaults to None

- **previous_image** (*Image, optional*) – Image of the same well and sample but taken on a previous date , defaults to None
- **next_image** (*Image, optional*) – Image of the same well and sample but taken on a future date, defaults to None
- **alt_image** (*Image, optional*) – Image of the same well and sample but taken with a different imaging tech, defaults to None

property bites

classify_image()
Classify the *Image* using the MARCO CNN model. Sets the `machine class` and `prediction_dict` attributes based on the model results.

static clean_base64_string(*string*)
Image instances may contain byte strings that store their actual crystallization image encoded as base64. Previously, these byte strings were written directly into the json file as strings causing the *b* byte string identifier to be written along with the actual base64 data. This method removes those artifacts if they are present and returns a clean byte string with only the actual base64 data.

Parameters **string** (*str*) – A string to interrogate for base64 compliance

Returns byte string with non-data artifacts removed

Return type bytes

property date
The date associated with this *Image*. Presumably should be the date the image was taken.

Returns Datetime object representation of the *Image*'s imaging date

Return type datetime

delete_all_pixmap_data()
Deletes the pixmap data for the *Image* instance this method is called on and for any other `Image` is linked to. This includes images referenced by the `alt_image` , `next_image` and `previous_image` attributes.

delete_pixmap_data()
Replaces the *Image*'s pixmap data with a null pixmap which effectively deletes the existing pixmap data. Used to free up memory after a pixmap is no longer needed.

encode_base64()

encode_bytes()
If the *path* attribute exists and is an image file then encodes that file as a base64 string and returns the encoded image data.

Returns base64 encoded image

Return type str

property formated_date
Get the image's `marco_date` attribute formated in the month/date/year format. If the *Image* has no *date* returns an empty string.

Returns Date

Return type str

get_linked_images_by_date()
Get all `Image` instance by date. Image linking by date is accomplished by creating a bi-directional

linked list between *Image* instances, where each *Image* acts as a node and the `next_image` and `previous_images` act as the forwards and backwards pointers respectively.

Returns All `Image` by date

Return type list

get_linked_images_by_spectrum()
Get all `Image` instance by spectrum. Linking images by spectrum is accomplished by creating a monodirectional circular linked list where *Image* instances serve as nodes and their `alt_image` attribute acts as the pointer to the next node.

Returns List of all *Images* linked to this *Image* by spectrum

Return type list

get_tool_tip()
Create a string to use as the tooltip for this *Image*.

Returns Tooltip string

Return type str

height()
Get the height of the *Image*'s pixmap. The pixmap must be set for this function to return an actual size.

Returns Height of the *Image*'s pixmap

Return type int

property human_class
Return the *human_class* attribute which specifies the current human classification of the *Image*.

Returns Current human classification of the *Image*

Return type str

property machine_class
MARCO classification of the *Image*.

Returns Current MARCO classification of this image

Return type str

classmethod no_image()
Return an *Image* instance using the image data referenced by the `polo.DEFAULT_IMAGE_PATH` constant. The default image is used to fill in for missing data and when filters cannot find any matching results.

Returns Default *Image*

Return type *Image*

property path
Filepath for the image. Note that if this path is loaded from an xtal file, this path may not exists because the xtal file may have been created on a different machine.

Returns Path to image file

Return type str

setPixmap(*scaling=None*)
Loads the *Image*'s pixmap into memory which then allows for displaying the *Image* to the user. `Image` pixmap when the *Image* actually needs to be shown to the user as it is expensive to hold in memory.

Parameters **scaling** (*float, optional*) – Scaler for the pixmap; between 0 and 1, defaults to None

standard_filter(*image_types*, *human*, *marco*, *favorite*)
Method that determines if this *Image* should be included in a set of filtered Image meets the filtering requirements specified by the method's arguments, otherwise returns False.

Parameters

- **image_types** (*list or set*) – Collection of image classifications. The *Image*'s classification must in included in this collection for the method to return True.
- **human** (*bool*) – If True, use the *Image*'s human classification as the overall image classification.
- **marco** (*bool*) – If True, use the *Image*'s MARCO classification as the overall image classification.

Returns True if the *Image* meets the filter requirements, False otherwise

Return type bool

classmethod to_graphics_scene(*image*)
Convert an Image object to a *QGraphicsScene* with the Image added as a pixmap to the *QGraphicsScene*.

Parameters **image** (Image) – Image instance

Returns QGraphicsScene

Return type QGraphicsScene

width()
Get the height of the *Image*'s pixmap. The pixmap must be set for this function to return an actual size.

Returns Width of the *Image*'s pixmap

Return type int

polo.crystallography.run module

class polo.crystallography.run.**HWIRun**(*cocktail_menu*, *plate_id=None*, *num_wells=1536*, *alt_spectrum=None*, *next_run=None*, *previous_run=None*, ***kwargs*)
Bases: *polo.crystallography.run.Run*

AllOWED_PLOTS = ['Classification Counts', 'MARCO Accuracy', 'Classification Progress',

add_images_from_dir()
Populates the images attribute with a list of *Images* instances read from the image_dir attribute filepath. Currently is dependent on having a cocktail Menu available.

get_linked_alt_runs()
Return all HWIRun is linked to by spectrum. See link_to_alt_spectrum().

Returns List of runs linked to this run by spectrum

Return type list

get_tooltip()
The same as get_tooltip().

insert_into_alt_spec_chain()
When runs are first loaded into Polo they are automatically linked together. Normally, *HWIRuns* that are

linked by date should contain only visible spectrum images and `alt_spectrum` attribute. This means that one could navigate from a visible spectrum *HWIRun* to all alt spectrum `link_to_alt_spectrum`.

link_to_alt_spectrum(*other_run*)
Similar to `link_to_next_date()` except instead of creating a linked list through the `next_run` and `previous_run` attributes this method does so through the `alt_spectrum` attribute. The linked list created is mono-directional so if a series of *HWIRun*'s are being linked the last run should be linked to the first run to circularize the linked list.

Parameters **other_run** (`HWIRun`) – *HWIRun* to link to this *HWIRun* by spectrum

link_to_next_date(*other_run*)
Link this *HWIRun* to another *HWIRun* instance that is of the same sample but photographed at a later date. This creates a bi-directional linked list structure between the two `HWIRun` instance will point to the *other_run* through the `next_run` attribute and *other_run* will point back to this *HWIRun* through its `previous_run` attribute. This method does not attempt to recognize which run was imaged first so this should be determined before calling, likely by sorting a list of *HWIRun instances by their :attr:`~polo.crystallography.run.Run.date* attribute.

Example:

```
# Starting with a collection of Run objects in a list
runs = [run_b, run_a, run_d, run_c]
# sort them by date
runs = sorted(runs, lambda r: r.date)
# link them together by date
[r[i].link_to_next_date(r[i+1]) for i in range(len(runs)-2)]
```

This would create a linked list with a structure link the representation below.

```
run_a <-> run_b <-> run_c <-> run_d
```

Parameters **other_run** (`HWIRun`) – *HWIRun* instance representing the next imaging run

class polo.crystallography.run.**Run**(*image_dir*, *run_name*, *image_spectrum=None*, *date=None*, *images=[]*, ***kwargs*)

Bases: `object`

AllOWED_PLOTS = ['Classification Counts', 'MARCO Accuracy', 'Classification Progress']

add_images_from_dir()
Adds the contents of a directory to `images` attribute.

encode_images_to_base64()
Helper method that encodes all images in the *class`~polo.crystallography.run.Run* to base64.

get_current_hits()

get_images_by_classification(*human=True*)
Create a dictionary of image classifications. Keys are each type of classification and values are lists of *Image* s with classification equal to the key value. The *human* boolean determines what classifier should be used to determine the image type. Human = True sets the human as the classifier and False sets MARCO as the classifier.

get_tooltip()

image_filter_query(*image_types*, *human*, *marco*, *favorite*)
General use method for returning :class:`~polo.crystallography.image.Image`s based on a set of filters. Used whereever a user is allowed to narrow the set of :class:`~polo.crystallography.image.Image`s available for view.

Parameters

- **image_types** (*list or set*) – Returned images must have a classification that is included in this variable
- **human** (*bool*) – Qualify the classification type with a human classifier.
- **marco** (*bool*) – Qualify the classification type with a MARCO classifier.
- **favorite** (*bool*) – Returned images must be marked as *favorite* if set to True

unload_all_pixmaps(*start=None*, *end=None*, *a=False*)

Delete the pixmap data of all *Image* instances referenced by the `images` attribute. This method should be used to free up memory after the :class:`~polo.crystallography.run.Run`is no longer being viewed by the user.

Parameters

- **start** (*int, optional*) – Start index for range of images to unload, defaults to None
- **end** (*int, optional*) – End index for range of images to unload, defaults to None
- **a** – Flag to unload pixmap data for all *Images* this *Image* is linked to

Module contents

6.1.2 polo.marco package

Submodules

polo.marco.run_marco module

polo.marco.run_marco.**classify_image**(*tf_predictor*, *image_path*)

Given a tensorflow predictor (the MARCO model) and the path to an image, runs the model on that image. Returns a tuple where the first item is the classification with greatest confidence and the second is a dictionary where keys are image classification types and values are model confidence for that classification. The image classifications of this dictionary are used universally throughout the program and are accessible through the `IMAGE_CLASSIFICATIONS` constant.

Parameters

- **tf_predictor** (*tensorflow model*) – Loaded MARCO model
- **image_path** (*str*) – Path to the image to be classified by the model

Returns tuple

Return type tuple

Module contents

6.1.3 polo.threads package

Submodules

polo.threads.thread module

class polo.threads.thread.**ClassificationThread**(*run_object*)

Bases: *polo.threads.thread.thread*

Thread that is specifically for classifying images using the MARCO model. This is a very CPU intensive process so it cannot be run on the GUI thread.

Parameters **run_object** (*Run or HWIRun*) – Run who's images are to be classified

change_value

estimated_time

run()
Method that actually does the classification work. Emits the the *change_value* signal everytime an image is classified. This is primary to update the progress bar widget in the *RunOrganizer* widget to notify the user how many images have been classified. Additionally, every five images classified the *estimated_time* signal is emitted which includes a tuple that contains as the first item the time in seconds it took to classify the last five images and the number of images that remain to be classified as the second item. This allows for making an estimate on about how much time remains in until the thread finishes.

class polo.threads.thread.**FTPDownloadThread**(*ftp_connection*, *file_paths*, *save_dir_path*)
Bases: *polo.threads.thread.thread*

Thread specific for downloading files from a remote FTP server.

Parameters

- **ftp_connection** (*FTP*) – FTP connection object to download files from
- **file_paths** (*list*) – List absolute filepaths on the FTP server to download
- **save_dir_path** (*str or Path*) – Path on the local machine to store all downloaded files in

download_path

file_downloaded

run(*self*)

class polo.threads.thread.**QuickThread**(*job_func*, *parent=None*, ***kwargs*)
Bases: *polo.threads.thread.thread*

QuickThreads are very similar to thread objects except instead of you writing code that would be executed by the *run* method directly, the function that the *QuickThread* will execute is passed as an argument to the *__init__*. Any arguments that the passed function requires are passed as key word arguments. Once the thread finished any values returned by the passed function are stored in the *QuickThread*'s polo.threads.thread.QuickThread.results attribute.

```
my_func = lambda x, y: x + y
x, y = 40, 60
my_thread = QuickThread(job_func=my_func, x=x, y=y)
# set up the thread with my_func and the args we want to pass
my_thread.start()
# my_thread.result will = 100 (x + y)
```

Parameters **job_func** (*func*) – Function to execute on the thread

run(*self*)

class polo.threads.thread.**thread**(*parent=None*)
Bases: PyQt5.QtCore.QThread

Very basic wrapper class around `QThread` class. Should be inherited by a more specific class and then the *run* method can be overwritten to provide functionality. Whatever code is in the *`run()`* method will be executed when `start()` is called. The *`run()`* method should not be called explicitly.

Parameters **`parent`** (*`QWidget, optional`*) – parent widget, defaults to None

`run`(*self*)

Module contents

6.1.4 polo.utils package

Submodules

polo.utils.dialog_utils module

`polo.utils.dialog_utils.`**`make_message_box`**(*message*, *parent=None*, *icon=1*, *buttons=1024*, *connected_function=None*)

General helper function to create popup message box dialogs to convey situational information to the user.

Parameters

- **`message`** (*`str`*) – The message to display to the user.
- **`parent`** (*`QDialog, optional`*) – Parent dialog, defaults to None
- **`icon`** (*`int, optional`*) – QMessageBox icon to display along with the message, defaults to QtWidgets.QMessageBox.Information
- **`buttons`** (*`set, optional`*) – Buttons to include in the message box, defaults to QtWidgets.QMessageBox.Ok
- **`connected_function`** (*`func, optional`*) – Function to connect to buttonClicked event, defaults to None

Returns The message box.

Return type QMessageBox

polo.utils.exceptions module

exception `polo.utils.exceptions.`**`EmptyDirectoryError`**
Bases: `Exception`

Raised when attempting to load images from an empty directory

exception `polo.utils.exceptions.`**`EmptyRunNameError`**
Bases: `Exception`

Raised when reading in an HWI directory but it does not contain number of images corresponding to number of wells.

exception `polo.utils.exceptions.`**`ForbiddenImageTypeError`**
Bases: `Exception`

Raised when user attempts to load in an image that is not in the allowed image types.

exception `polo.utils.exceptions.`**`IncompletePlateError`**
Bases: `Exception`

Raised when reading in an HWI directory but it does not contain number of images corresponding to number of wells.

exception polo.utils.exceptions.**InvalidCocktailFile**
Bases: Exception

Raised when user attempts to load in a file containing well cocktail information that does not confrom to existing formating standards.

exception polo.utils.exceptions.**NotASolutionError**
Bases: Exception

Raised when user attempts to load in an image that is not in the allowed image types.

exception polo.utils.exceptions.**NotHWIDirectoryError**
Bases: Exception

Raised when user attempts to read in a directory as HWI but it does not look like one.

TODO: Add utils function to determine when to raise this exception.

polo.utils.ftp_utils module

polo.utils.ftp_utils.**catch_ftp_errors**(*funct*)
General decorator function for catching any errors thrown by other ftp_utils functions.

polo.utils.ftp_utils.**logon**(**args*, ***kwargs*)

polo.utils.io_utils module

class polo.utils.io_utils.**BarTender**(*cocktail_dir*, *cocktail_meta*)
Bases: object

Class for organizing and accessing *Menu* data.

Parameters

- **cocktail_dir** (*str or Path*) – Directory containing cocktail menu csv filepaths
- **cocktail_meta** (*Path or str*) – Path to cocktail metadata file which describes the contents of each cocktail menu csv file

Cocktail metadata file should be a csv file with the following headers ordered from top to bottom. Each header name is followed by a description.

```
File Name: Name of cocktail menu file
Dates Used: Range of dates the cocktail menu was used (m/d/y-m/d/y)
Plate Number
Screen Type: 'm' for membrane screens, 's' for soluble screens
```

add_menus_from_metadata()
Adds *Menu* objects to the :attr:polo.utils.io_utils.BarTender.menus` attribute by reading the cocktail csv files included in the COCKTAIL_DATA_PATH directory.

static date_range_parser(*date_range_string*)
Utility function for converting the date ranges in the cocktail metadata csv file to datetime objects using the *datetime_converter()* method.

Date ranges should have the format

```
'start date - end date'

If the date range is for the most recent cocktail menu then there
will not be an end date and the format will be

'start date - '
```

Parameters **date_range_string** (*str*) – string to pull dates out of

Returns tuple of datetime objects, start date and end date

Return type tuple

static datetime_converter(*date_string*)
General utility function for converting strings to datetime objects. Attempts to convert the string by trying a couple of datetime formats that are common in cocktail menu files and other locations in the HWI file universe Polo runs across.

Parameters **date_string** (*str*) – string to convert to datetime

Returns datetime object

Return type datetime

get_menu_by_basename(*basename*)
Uses the basename of a *Menu* file path to return a *Menu* object. Useful for retrieving menus based on the text of comboBoxes since when menus are displayed to the user only the basename is used.

Parameters **basename** (*str*) – Basename of a *Menu* file path

Returns Menu instance who's basename matches the *basename* argument, returns None is no menu is found

Return type *Menu* or None

get_menu_by_date(*date*, *type_='s'*)
Get a *Menu* instance who's usage dates include the date passed through the *date* argument and matches the screen type passed through the *type_* argument.

Screen types can either be 's' for 'soluble' screens or 'm' for membrane screens.

Parameters

- **date** (*datetime*) – Date to search menus with
- **type** (*str*) – Type of screen to return (soluble or membrane)

Returns menu matching the given date and type

Return type *Menu*

get_menu_by_path(*path*)
Returns a *Menu* instance by its file path, which is used as the key for accessing the menus attribute normally.

Parameters **path** (*str*) – file path of a menu csv file

Returns Menu instance that is mapped to given path

Return type *Menu*

get_menus_by_type(*type_='s'*)
Returns all *Menu* instances of a given screen type.

's' for soluble screens and 'm' for membrane screens. No other characters should be passed to *type_*.

Parameters **type** (*str (max length 1)*) – Key for type of screen to return

Returns list of menus of that screen type

Return type list

class polo.utils.io_utils.**CocktailMenuReader**(*menu_file_path*, *delim=','*)

Bases: object

CocktailMenuReader instances should be used to read a csv file containing a collection of cocktail screens. The csv file should contain cocktail related formulations and assign each cocktail to a specific well in the screening plate. CocktailMenuReader is essentially a wrapper around the `csv.DictReader` class. However it returns a `Cocktail` instance instead of returning a dictionary via when it's __iter__ method is called.

```
cocktail_menu = 'path/to/my/csv'
my_reader = CocktailMenuReader(cocktail_menu)
csv_cocktails = my_reader.read_menu_file()
# csv_cocktails now holds list of Cocktail objects read from
# cocktail_menu
```

Parameters

- **menu_file** (*str or Path*) – Path to cocktail menu file to read. Should be csv formated
- **delim** (*str, optional*) – Seperator for menu_file; really should not need to be changed, defaults to ','

cocktail_map = {0: 'well_assignment', 1: 'number', 2: 'commercial_code', 8: 'pH'}

formula_pos = 4

read_menu_file()

Read the contents of cocktail menu csv file. The *Menu* file path is read from the `Menu.menu_file_path` attribute. The first **two** lines of all the cocktail menu files included in Polo are header lines and so the reader will skip the first two lines before actually reading in any data. Each row is converted to a *Cocktail* object and then added to a dictionary based on the *Cocktail* instance's well assignment.

Returns Dictionary of Cocktail instances. Key value is the Cocktail's well assignment in the screening plate (base 1).

Return type dict

classmethod set_cocktail_map(*map*)

Classmethod to edit the *cocktail_map*. The *cocktail_map* describes where the `Cocktail` level information is stored in a given cocktail row in the csv file. It is a dictionary that maps specific indices in a row to `Cocktail` attributes.

The default cocktail_map dictionary is below.

```
>>> cocktail_map = {
0: 'well_assignment',
1: 'number',
8: 'pH',
2: 'commercial_code'
}
```

This tells instances of CocktailMenuReader to look at index 0 of a row for the well_assignment attribute of the Cocktail class, index 1 for the number attribute of the Cocktail class, etc.

Parameters **map** (*dict*) – Dictionary mapping csv row indicies to Cocktail object attributes

classmethod set_formula_pos(*pos*)
Classmethod to change the *CocktailMenuReader.formula_pos* attribute. The *formula_pos* describes the location (base 0) of the chemical formula in a row of a cocktail menu file csv. For some reason, HWI cocktail menu files will only have one chemical formula per row (cocktail) no matter the number of reagents that composite that cocktail. This is why its location is represented using an int instead of a dict.

Generally, *CocktailMenuReader.formula_pos* should not be changed without a very good reason as the position of the chemical formula is consistent across all HWI cocktail menu files.

Parameters **pos** (*int*) – Index where chemical formula can be found

class polo.utils.io_utils.**HtmlWriter**(*run*, ***kwargs*)
Bases: *polo.utils.io_utils.RunSerializer*

static make_template(*template_path*)
Given a path to an html file to serve as a jinja2 template, read the file and create a new template object.

Parameters **template_path** – Path to the jinja2 template file.

write_complete_run(*output_path*, *encode_images=True*)
Create an HTML report from a Run or HWIRun instance.

Parameters

- **output_path** (*str or Path*) – Path to write html file to.
- **encode_images** (*bool, optional*) – Write images as base64 directly to the html file, defaults to True. Greatly increases the file size but means that report will still contain images even if the originals are deleted or removed.

Returns Path to html report if write succeeds, Exception otherwise.

Return type str or Exception

write_grid_screen(*output_path*, *plate_list*, *well_number*, *x_reagent*, *y_reagent*, *well_volume*, *run_name=None*)
Write the contents of optimization grid screen to an html file.

Parameters

- **output_path** (*str*) – Path to html file
- **plate_list** (*list*) – list containing grid screen data
- **well_number** (*int or str*) – well number of hit screen is created from
- **x_reagent** (Reagent) – reagent varied in x direction
- **y_reagent** (Reagent) – reagent varied in y direction
- **well_volume** (*int or str*) – Volume of well used in screen
- **run_name** (*str, optional*) – name of run, defaults to None

class polo.utils.io_utils.**JsonWriter**(*run*, *output_path*)
Bases: *polo.utils.io_utils.RunSerializer*

Small class that can be used to serialize a run to a json formated file.

Parameters

- **run** (Run *or* HWIRun) – Run to write as a json file
- **output_path** (*str or Path*) – Path to write json file to

static json_encoder(*obj*)
General purpose json encoder for encoding python objects. Very similar to the encoder function *json_encoder()* except does not include class and module information in the returned dictionary. If the object cannot be converted to a dictionary it is returned as a string.

Parameters **obj** (*obj*) – Object to convert to dictionary

Returns dict or str

Return type dict or str

write_json()
Write the Run instance referenced by the run attribute to a json file at the location specified by the output_path attribute. If the file is written successfully returns True otherwise returns an Exception.

Returns True or Exception

Return type bool, Exception

class polo.utils.io_utils.**Menu**(*path*, *start_date*, *end_date*, *type_*, *cocktails={}*)
Bases: object

property cocktails

class polo.utils.io_utils.**MsoReader**(*mso_path*)
Bases: object

The MsoReader class is used to parse the content of mso formated files and apply the image classifications stored in these files to runs in Polo.

classify_images_from_mso_file(*images*)
Applies the image classifications stored in an mso file to a collection of *Image* objects. Allows for some degree of compatability with MarcoscopeJ and for users who have stored their image classifications in the mso format. Additionally, human classification backups are saved as mso files when Polo is closed as they take up much less space than xtal files.

Parameters **images** (*list*) – List of images to apply classifications to

Returns List of images with mso classifications applied, or Exception if this fails

Return type list or Exception

static read_mso_classification(*mso_classification*)
Helper method to read and convert image classifications in a mso file to MARCO image classifications. The exact conversion scheme is layed out in the REV_MSO_DICT constant. Additionally, MarcoscopeJ will allow images to have multiple classifications by assigning multiple image codes seperated by "-". If this is the case Polo takes the classification corresponding to the mso code with the highest value.

Parameters **mso_classification** (*str*) – Mso image classification code

Returns MARCO classification if code can be decoded, None otherwise

Return type str or None

class polo.utils.io_utils.**MsoWriter**(*run*, *output_path*)
Bases: *polo.utils.io_utils.RunSerializer*

property first_line
Create the first line of the mso file.

Returns List to write as the first line of the mso file.

Return type list

get_cocktail_csv_data()
Reads and returns the cocktail csv data assigned to the `cocktail_menu` attribute of the MsoWriter's `run` attribute.

Returns List of lists containing cocktail csv data.

Return type list

mso_version = 'msoversion2'

static row_formater(*cocktail_row*)
Format a cocktail row as read from a cocktail csv file to an mso file row. Main change is appending empty strings to the cocktail row so list ends up always having a length of 17. This is important because the image classification code always occurs at the 18th item in an mso file row.

Parameters **cocktail_row** (*list*) – Cocktail row as read from cocktail csv file.

Returns Cocktail row reformated for mso writing.

Return type list

write_mso_file(*use_marco_classifications=False*)
Writes an mso formated file for use with MarcoScopeJ based on the images and classifications of the `Run` instance referenced by the MsoWriter's `run` attribute.

Parameters **use_marco_classifications** (*bool, optional*) – Include the MARCO classification in the mso file instead of human classifications, defaults to False

Returns True if file is written successfully, False otherwise.

Return type bool

class polo.utils.io_utils.**PptxWriter**(*output_path*, *image_types=None*, *human=False*, *marco=False*, *favorite=False*)
Bases: `object`

Use for creating pptx presentation slides from `Run` or `HWIRun` instances.

Parameters

- **output_path** (*str or Path*) – Path to write pptx file to.
- **included_attributes** (*dict, optional*) – [description], defaults to {}
- **image_types** (*set or list, optional*) – Images included in the presentation must have a classification in this set, defaults to None
- **human** (*bool, optional*) – Use human classification as the image classification, defaults to False
- **marco** (*bool, optional*) – Use the MARCO classification as the image classification, defaults to False
- **favorite** (*bool, optional*) – Only include images marked as favorite, defaults to False

add_classification_slide(*well_number*, *rep_image*)
Add a slide containing details about an images MARCO and human classification in a table.

Parameters

- **well_number** (*int*) – Well number (index) of image to use in the title of the slide
- **rep_image** (`Image`) – Image object to make slide from

add_cocktail_slide(*well*, *cocktail*)
Add slide with details on `Cocktail` information.

Parameters

- **well** (*int*) – Well number to use in slide title
- **cocktail** (Cocktail) – Cocktail to write as a slide

add_image_to_slide(*image*, *slide*, *left*, *top*, *height*)
Helper method for adding images to a slide. If the Image does not have a file written on the local machine as can be the case with saved runs who's image data only exists in their xtal files this method will write a temporary image file to the Polo TEMP_DIR which then should be deleted after the presentaton file is written.

Parameters

- **image** (Image) – Image to add to the slide
- **slide** (*slide*) – Slide to add the image to
- **left** (*float*) – Left cordinate location of the image in inches
- **top** (*float*) – Top cordinate location of the image in inches
- **height** (*float*) – Height of the image in inches

Returns []

Return type [type]

add_multi_image_slide(*slide*, *images*, *labeler*)
General helper method for adding a slide that will have multiple images.

Parameters

- **slide** (*slide*) – Slide to add the images to
- **images** (*list*) – Images to add to the slide
- **labeler** (*func*) – Function to use to label the individual images

Returns slide with images added

Return type slide

add_multi_spectrum_slide(*images*, *well_number*)
Create a slide to show a all spectrums a well has been imaged in.

Parameters

- **images** (*list*) – Images to include on the slide
- **well_number** (*int*) – Well number to use in the slide title

Returns New slide

Return type slide

add_new_slide(*template=5*)

add_single_image_slide(*image*, *title*, *metadata=None*, *img_scaler=0.65*)
General helper method for adding a slide with a single image to a presentation.

Parameters

- **image** (Image) – Image to add to the slide
- **title** (*str*) – Title to use for the slide
- **metadata** (*str, optional*) – Additional information to write to the slide, defaults to None

- **img_scaler** (*float, optional*) – Scaler to apply to size of the image, defaults to 0.65 ,should be between 0 and 1. 1 is full sized image.

Returns The new slide with Image added

Return type slide

add_table_to_slide(*slide*, *data*, *left*, *top*)
General helper method for adding a table to a slide.

Parameters

- **slide** (*slide*) – Slide to add the table to
- **data** (*list*) – List of lists that has the data to write to the table
- **left** (*float*) – Left offset in inches to place to table
- **top** (*float*) – Top cordinate for placing the table

Returns Slide with table added

Return type slide

add_text_to_slide(*slide*, *text*, *left*, *top*, *width*, *height*, *rotation=0*, *font_size=14*)
Helper method to add text to a slide

Parameters

- **slide** (*slide*) – Slide to add text to
- **text** (*str*) – Text to add to the slide
- **left** (*float*) – Left cordinate location of the text in inches
- **top** (*float*) – Top cordinate location of the text in inches
- **width** (*float*) – Width of the text in inches
- **height** (*float*) – Height of the text in inches
- **rotation** (*int, optional*) – Rotation to apply to the text in degrees, defaults to 0
- **font_size** (*int, optional*) – Font size of text, defaults to 14

Returns Slide with text added

Return type slide

add_timeline_slide(*images*, *well_number*)
Create a timeline (time resolved) slide that show the progression of a sample in a particular well.

Parameters

- **images** (*list*) – List of images to include in the slide
- **well_number** (*int*) – Well number to use in the title of the slide

Returns New slide

Return type slide

delete_presentation()

delete_temp_images()
Delete an temporary images used to create the pptx presentation.

Returns True, if images are removed successfully, Exception otherwise.

Return type bool or Exception

make_single_run_presentation(*run*, *title*, *subtitle=None*, *cocktail_data=True*, *all_specs=False*, *all_dates=False*)

sort_runs_by_spectrum(*runs*)

Divids runs into two lists, one containing visible spectrum runs and another containing all non-visible runs.

Parameters **runs** (*list*) – List or runs

Returns tuple, first item is visible runs second is non-visible runs

Return type tuple

class polo.utils.io_utils.**RunCsvWriter**(*run*, *output_path=None*, ***kwargs*)

Bases: *polo.utils.io_utils.RunSerializer*

property fieldnames

Get the current fieldnames based on the data stored in the `run` attribute. Currently is somewhat expensive to call since it requires parsing all records in `run` in order to determine all the fieldnames that should be included in order to definitely avoid keyerrors later down the line.

Returns List of fieldnames (headers) for the csv data

Return type list

get_csv_data()

Convert the `run` attribute to csv style data. Returns a tuple of headers and a list of dictionaries with each dictionary representing one row of csv data.

Returns Tuple, list of headers and list of dicts

Return type tuple

classmethod image_to_row(*image*)

Given an `Image` object, convert it into a list that could be easily written to a csv file.

Parameters **image** (`Image`) – `Image` object to convert to list

Returns List

Return type list

property output_path

Get the hidden attribute *_output_path*.

Returns Output path

Return type str

write_csv()

Write the `Run` object referenced by the `run` attribute as a csv file to the location specified by the *output_path* attribute.

Returns True, if csv file content was written successfully, return error thrown otherwise.

Return type Bool or Exception

class polo.utils.io_utils.**RunDeserializer**(*xtal_path*)

Bases: `object`

static clean_base64_string(*string*, *out_fmt=<class 'bytes'>*)

Image instances may contain byte strings that store their actual crystallization image encoded as base64. Previously, these byte strings were written directly into the json file as strings causing the b' byte string identifier to be written along with the actual base64 data. This method removes those artifacts if they are present and returns a clean byte string with only the actual base64 data.

Parameters **string** (*str*) – a string to interrogate

Returns byte string with non-data artifacts removed

Return type bytes

static dict_to_obj(*d*)
Opposite of the obj_to_dict method in XtalWriter class, this method takes a dictionary instance that has been previously serialized and attempts to convert it back into an object instance. Used as the *object_hook* argument when calling *json.loads* to read xtal files.

Parameters **d** (*dict*) – dictionary to convert back to object

Returns an object

Return type object

make_read_xtal_thread()

xtal_header_reader(*xtal_file_io*)
Reads the header section of an open xtal file. Should always be called before reading the json content of an xtal file. Note than xtal files must always have a line of equal signs before the json content even if there is no header content otherwise this method will read one line into the json content causing the json reader to throw an error.

Parameters **xtal_file_io** (*TextIoWrapper*) – xtal file currently being read

Returns xtal header contents

Return type list

xtal_to_run(***kwargs*)
Attempt to convert the file specified by the path stored in the `xtal_path` attribute to a `Run` object.

Returns Run object encoded by an xtal file

Return type *Run*

xtal_to_run_on_thread()
Wrapper method around *xtal_to_run* method. Does the exact same thing except creates a *QuickThread* instance and runs *xtal_to_run* on the thread. When finished adds the newly created run to the main window's loaded_run dictionary to signify that the run has been loaded and is ready for further operations.

class polo.utils.io_utils.**RunImporter**
Bases: `object`

Class to hold general use methods for importing runs into Polo.

static crack_open_a_rar_one(*rar_path*)
Method to open a compressed rar archive.

Parameters **rar_path** (*str or Path*) – Path to rar archive.

Returns Path to uncompressed archive if successful

Return type Path

static directory_validator(*dir_path*)
Check if a directory should proceed further down the import pipeline. Includes checks to make sure the directory exists, is a directory and that the directory contains images of filetypes that can be imported.

Parameters **dir_path** (*str or Path*) – Path to a directory

Returns True, if directory can be imported, an Exception otherwise

Return type bool or Exception

static import_from_xtal_thread(*xtal_path*)
Given the path to an xtal file returns a `QuickThread` which can be run to load the run serialized in the xtal file.

Parameters **xtal_path** (*str*) – File path to xtal file.

Returns QuickThread for deserializing the given xtal file.

Return type *QuickThread*

static import_general_run(*data_dir*, ***kwargs*)
Attempt to import a `Run` from a directory of images.

Parameters **data_dir** (*[type]*) – [description]

Returns [description]

Return type [type]

static import_hwi_run(*data_dir*, ***kwargs*)
Attempt to create a `HWIRun` from a directory of images.

Parameters **data_dir** (*str or Path*) – Directory to import from.

Returns HWIRun if import is successful, False otherwise

Return type *HWIRun*, bool

static import_run_from_directory(*data_dir*, ***kwargs*)
Imports a run from a local directory. First attempts to import the run as an `HWIRun` and if this fails attempts an import as a general `Run` object.

Parameters **data_dir** (*str or Path*) – Directory to build the Run from

Returns Run, HWIRun or False depending on directory content and if import succeeds.

Return type *Run*, *HWIRun*, bool

static make_xtal_file_dialog(*parent=None*)
Create a file dialog specifically for browsing for xtal files.

Parameters **parent** (*QDialog, optional*) – Parent for the file dialog, defaults to None

Returns QFileDialog

Return type QFileDialog

static parse_hwi_dir_metadata(*dir_name*)
Parse the directory name of an `HWIRun` directory to pull out metadata. If the directory name conforms to HWI naming conventions the function should be able to return the image spectrum, the plate id, the date and the run name. If the directory does not conform to HWI naming standards this will cause an exception which is caught and the function will return None.

If the parse is successful will return a dictionary with the following format.

```
{'image_spectrum': String describing the image spectrum,
'plate_id': String representing the plate id,
'date': Datetime instance of imaging date,
'run_name': String holding the run name
}
```

Parameters **dir_name** (*str or Path*) – Name of directory to check for metadata

Returns Dictionary of extracted metadata or None if parse fails

Return type dict or None

static unpack_rar_archive_thread(*archive_path*)
Create a QuickThread that is setup to de-compress a rar archive file.

Parameters **archive_path** (*str or Path*) – Path to rar archive.

Returns QuickThread that can be run to de-compress the given archive path.

Return type *QuickThread*

class polo.utils.io_utils.**RunLinker**
Bases: object

Class to hold methods relating to linking runs either by date or by spectrum.

static link_runs_by_date(*runs*)

static link_runs_by_spectrum(*runs*)
Link a collection of *HWIRun* instances together by spectrum. All non-visible *HWIRun* instances are linked together in a monodirectional circular linked list. Each visible *HWIRun* instance will then point to the same non-visible run through their alt_spectrum attribute as a way to access the non-visible linked list.

Parameters **runs** (*list*) – List of runs to link together

Returns List of runs linked by spectrum

Return type list

static the_big_link(*runs*)
Wrapper method to do all the linking required for a collection of runs. First calls *unlink_runs_completely()* to separate any existing links so things do not get tangled. Then calls *link_runs_by_date()* and *link_runs_by_spectrum()*.

Parameters **runs** (*list*) – List of runs to link

Returns List of runs with links made

Return type list

static unlink_runs_completely(*runs*)
Cuts all links between the *HWIRun* instances passed through the *runs* argument and the *Image* instances in those runs.

Parameters **runs** (*list*) – List of runs

Returns List of runs without any links

Return type list

class polo.utils.io_utils.**RunSerializer**(*run*)
Bases: object

classmethod make_thread(*job_function*, ***kwargs*)
Creates a new QuickThread object. The job function is the function the thread will execute and and arguments that the job function requires should be passed has keyword arguments. These are stored as a dictionary in the new thread object until the thread is activated and they are passed as arguments.

static path_suffix_checker(*path*, *desired_suffix*)
Check is a file path has a desired suffix, if not then replace the current suffix with the desired suffix. Useful for checking filenames that are taken from user input.

Parameters **desired_suffix** – File extension for given file path.

static path_validator(*path*, *parent=False*)
Tests to ensure a path exists. Passing parent = True will check for the existence of the parent directory of the path.

class polo.utils.io_utils.**SceneExporter**(*graphics_scene=None*, *file_path=None*)
Bases: object

write()

static write_image(*scene*, *file_path*)
Write the contents of a QGraphicsScene to a png file.

Parameters

- **scene** (*QGraphicsScene*) – QGraphicsScene to convert to image file.
- **file_path** (*str*) – Path to save image to.

Returns File path to saved image if successful, Exception otherwise.

Return type str or Exception

class polo.utils.io_utils.**XmlReader**(*xml_path=None*, *xml_files=[]*)
Bases: object

discover_xml_files(*parent_dir*)
Look for xml files in a given directory.

Parameters **parent_dir** (*str or Path*) – Directory to look for xml files in

Returns List of xml file paths, if any exist

Return type list

find_and_read_plate_data(*parent_dir*)
Find xml metadata files in a given directory. Read the data from xml files that contain the plate_def key string.

Parameters **parent_dir** (*Path or str*) – Directory to look for xml files

Returns Dict if xml file found and read successfully, False otherwise

Return type dict or bool

static get_data_from_xml_element(*xml_element*)
Return the data stored in an *xml_element*. Helper method for reading xml files.

Parameters **xml_element** (*[type]*) – xml element to read data from

Returns Dictionary of data stored in xml element

Return type dict

platedef_key = 'platedef'

read_plate_data_xml(*xml_path=None*)
Read the data stored in an xml document. HWI includes metadata about samples, imaging dates and other plate information in each rar archive that is distrubted. This method is used to read that data so it can be incorporated into HWIRun objects.

Parameters **xml_path** (*str or Path, optional*) – Path to xml file to read, defaults to None. If None uses the xml path stored in *xml_path* attribute.

Returns Dictionary of xml data if read was successful, Exception otherwise

Return type dict or Exception

class polo.utils.io_utils.**XtalWriter**(*run*, *main_window*, ***kwargs*)
Bases: *polo.utils.io_utils.RunSerializer*

static clean_run_for_save(*run*)
Remove circular references from the run passed through the *run* argument to avoid issues when writing to json files.

Parameters **run** (Run *or* HWIRun) – Run to clean (remove circular references)

Returns Run, free from circular references

Return type *Run* or *HWIRun*

file_ext = '.xtal'

finished_save()

header_flag = '<>'

header_line = '{}{}:{}\n'

static json_encoder(*obj*)
Use instead of the default json encoder when writing an xtal file. If the encoded object is from a module within Polo will include a module and class identifier so it can be more easily deserialized when loaded back into the program.

Param obj: An object to serialize to json.

Returns A dictionary or string version of the passed object

run_to_dict()
Create a json string from the run stored in the run attribute.

Returns Run instance serialized to json

Return type str

write_xtal_file(*output_path=None*)
Method to serialize run object to xtal file format.

Parameters **output_path** (*str*) – Xtal file path

Returns path to xtal file

Return type str

write_xtal_file_on_thread(*output_path*)
Wrapper method around *write_xtal_file()* that executes on a QuickThread instance to prevent freezing the GUI when saving large xtal files

Parameters **output_path** (*str*) – Path to xtal file

property xtal_header
Creates the header for an xtal file when called. Header lines are indicated as such by the string in the header_line constant, which should be '<>'. The last line of the header will be a row of equal signs and then the actual json content begins on the next line.

polo.utils.io_utils.**check_for_missing_images**(*dir_path*, *expected_image_count*)

polo.utils.io_utils.**directory_validator**(*dir_path*)

polo.utils.io_utils.**if_dir_not_exists_make**(*parent_dir*, *child_dir=None*)
If only parent_dir is given attempts to make that directory. If parent and child are given tries to make a directory child_dir within parent dir.

polo.utils.io_utils.**list_dir_abs**(*parent_dir*, *allowed=False*)

polo.utils.io_utils.**parse_HWI_filename_meta**(*HWI_image_file*)
HWI images have a standard file nameing schema that gives info about when they are taken and well number and that kind of thing. This function returns that data

polo.utils.io_utils.**parse_hwi_dir_metadata**(*dir_name*)

polo.utils.io_utils.**run_name_validator**(*new_run_name*, *current_run_names*)

polo.utils.io_utils.**write_screen_html**(*plate_list*, *well_number*, *run_name*, *x_reagent*, *y_reagent*, *well_volume*, *output_path*)

polo.utils.math_utils module

polo.utils.math_utils.**best_aspect_ratio**(*w*, *h*, *n*)

polo.utils.math_utils.**factors**(*n*)

polo.utils.math_utils.**get_cell_image_dims**(*w*, *h*, *n*)

polo.utils.math_utils.**get_image_cell_size**(*cell_aspect*, *w*, *h*)

polo.utils.unrar_utils module

polo.utils.unrar_utils.**test_for_working_unrar**(*unrar_exe='unrar'*)
Tests if a working unrar installation exists on the machine.

Parameters **unrar_exe** (*Path or str, optional*) – Path to unrar executable file, defaults to UNRAR_EXE

Returns True if working installation exists, False otherwise

Return type bool

polo.utils.unrar_utils.**unrar_archive**(*rar_path*, *target_dir=None*)
De-compress a rar archive and return the path to the uncompressed archive if it exists. All unrar functions including this one are dependent of their being a working unrar installation. Unrar is included for both Windows and Mac operating systems but not for Linux.

Parameters

- **rar_path** (*Path or str*) – Path to rar archive file
- **target_dir** (*Path or str, optional*) – Location to place the unrared file, defaults to None

Returns Path if unrar is successful, error code if unrar fails or Exception if

exception is raised in the unrar process. :rtype: Path, str or Exception

Module contents

6.1.5 polo.widgets package

Submodules

polo.widgets.file_browser module

class polo.widgets.file_browser.**fileBrowser**(*parent=None*)
Bases: PyQt5.QtWidgets.QTreeWidget

DATA_INDEX = 5

DIR_ICON = '/home/ethan/Documents/github/Marco_Polo/src/data/images/icons/dir.png'

FILE_ICON = '/home/ethan/Documents/github/Marco_Polo/src/data/images/icons/file.png'

get_checked_files(*home_dir*)
Traverse the file tree and return the full paths to files that have been selected by the user.

Parameters **home_dir** (*str or Path*) – User's home directory. This path is the parent of all returned files.

Returns List of checked Paths

Return type list

grow_tree_using_mlsd(*ftp*, *home_dir*, *set_checkable=True*)
Rescursively add child nodes to the tree by traversing a user's home directory at a remote ftp server. Filepaths are read using mlsd formating.

Parameters

- **ftp** (*FTP*) – FTP object with valid connection
- **home_dir** (*str or Path*) – Path to the user's home directory
- **set_checkable** (*bool, optional*) – Set files and dirs to checkable, defaults to True

polo.widgets.map_box module

class polo.widgets.map_box.**MapBox**
Bases: PyQt5.QtWidgets.QComboBox

_MapBox__init(*parent=None*, *mapping={}*, *sorter=<function MapBox.<lambda>>*)

current_value()

property mapping

polo.widgets.optimize_widget module

class polo.widgets.optimize_widget.**OptimizeWidget**(*parent*, *run=None*)
Bases: PyQt5.QtWidgets.QWidget

GRID_ICON = '/home/ethan/Documents/github/Marco_Polo/src/data/images/icons/grid.png'
The OptimizeWidget is a primary run interface widget that allows users to create optimization screens around the crystallization conditions that yielded xtal hits. The concept is very similar to the program MakeTray available from Hampton Research. Currently, users cannot specify their own conditions and are limited to the predetermined conditions of the *Menu* that was selected when the run was originally imported into Polo. Additionally, the OptimizeWidget is only available to HWIRun instances as the cocktail to well mapping cannot be inferred for other more general Run types.

Parameters

- **parent** (*QWidget*) – Parent Widget
- **run** (HWIRun, *optional*) – Run to screen for hits from, defaults to None

HTML_ICON = '/home/ethan/Documents/github/Marco_Polo/src/data/images/icons/html.png'

_check_for_overflow(*volume_list*)
Private method to check if the volume of `Reagent` instancess in a given well exceeds the total well volume. If an overflow is detected, return False otherwise return the volume of H20 that should be added to the well as a *UnitValue* instance.

Parameters **volume_list** (*list*) – List of *UnitValues* that consitute the contents of a well in the optimization plate

Returns *UnitValue* describing the volume of water that should be added to the well if it does not overflow in liters, False otherwise

Return type *UnitValue* or False

_error_checker()
Private method to check if all widgets and attributes have allowed values before calculating the actual grid screen. Show error message if there is a conflict.

_export_screen()
Private method to write the current optimization screen to a html file.

_gradient(*reagent*, *num_wells*, *step*, *stock=False*)
Private method for calculating a concentration gradient for a given `Reagent` using a given step size as a proportion of the `Reagent` instance's *concentration* attribute.

Parameters

- **reagent** (`Reagent`) – Reagent to vary concentration
- **num_wells** (*int*) – Number of wells to vary concentration across
- **step** (*float < 1*) – Proportion of hit concentration to vary each well by
- **stock** (*bool, optional*) – If True, vary the stock volume not the hit concentration unit, defaults to False

Returns List of UnitValues that make up the _gradient

Return type list

_handle_reagent_change(*x=False*, *y=False*, *const=False*)
Private method that handles when a reagent is changed. The arguments indicate which reagent has been changed.

Parameters

- **x** (*bool, optional*) – If True update the x reagent, defaults to False
- **y** (*bool, optional*) – If True update the y reagent, defaults to False
- **const** (*bool, optional*) – If True update the constant reagents, defaults to False

_make_plate_list()
Private method that converts the concents of the `tableWidget` UI (assuming that a optimization screen has been already rendered to the user) to a list of lists that is easier to write to html using the jinja2 template.

Returns tableWidget contents converted to list

Return type list

_make_well_html(*x_con*, *x_stock*, *y_con*, *y_stock*, *constants*, *water*)
Private method to format the information that describes the contents of an individual well into prettier html that can be displayed to the user in a `textBrowser` widget.

Parameters

- **x_con** (`UnitValue`) – Concentration of x reagent in this well

- **x_stock** (UnitValue) – Volume of x reagent stock in this well
- **y_con** (UnitValue) – Concentration of y reagent in this well
- **y_stock** (UnitValue) – Volume of y reagent stock in this well
- **constants** (*list of tuples*) – Tuples of constant reagents to be included in each well
- **water** (*Signed Value*) – Volume of water to be added to this well

Returns Html string to be rendered to the user

Return type str

_set_constant_reagents()
Private method that populates the listWidget with constant reagents to display to the user.

_set_hit_well_choices()
Private method that sets the hit well comboBox widget choices based on the images in the *run* attribute that are human classified as crystal. Wells are identified in the comboBox by their well number.

_set_reagent_choices()
Private method that sets Reagent choices for the x and y reagents based on the currently selected well. Reagents must come from the class:*Cocktail* instance associated with the selected well.

TODO: Add the option to vary pH instead of a reagent along either axis. This would also mean that the constant reagents would need to be updated.

_set_reagent_stock_con()
Private method. If a Reagent has already been assigned a stock concentration this method displays that concentration to the user through the appropriate UnitCombobBox instance. Only displays concentrations for the x and y reagents.

_set_reagent_stock_con_values(*x=False*, *y=False*, *const=False*)
Private method to update the stock concentations of current reagents through their stock_con attribute. The Reagent to update is indicated by the flag set to True at the time the method is called. The stock concentration value is pulled from each reagent's respective *unitComboBox* instance.

Parameters

- **x** (*bool, optional*) – If True, set x reagent stock con, defaults to False
- **y** (*bool, optional*) – If True, set the y reagent stock con, defaults to False
- **const** (*bool, optional*) – If True, sets the constant reagents stock con, defaults to False

_set_up_unit_comboboxes()
Private method that sets the base unit and the scalers of all unitComboBox instances that are apart of the UI.

_update_current_reagents(*image_index=None*)
Private method that updates x and y reagent comboBox widgets to reflect what Reagent instances are contained in the currently selected well.

Parameters **image_index** (*int, optional*) – Index of the Image to set Reagent choices from, defaults to None.

_write_optimization_screen()
Private method to write the current optimization screen to the tableWidget UI for display to the user.

adjust_unit(*signed_value*, *new_unit*)

property constant_reagents
Retrieve a set of `Reagents` that are not included as either the x reagent or the y reagent but are still part of the crystallization cocktail and therefore need to be included in the screen. Unlike either the x or y reagents, constant reagents do not change their concentration across the screening plate.

property hit_images
Retrieves a list of *Image* object instances that have human classification (*human_class* attribute) == 'Crystals'. Used to determine what wells to allow the user to optimize. Currently, only allow the user to optimize wells they have marked as crystal.

property run

property selected_constant
Return the constant `Reagent` that is currently selected by the user.

Returns Currently selected constant reagent if exists and selected, None otherwise

Return type *Reagent* or None

update()
Method to update reagents and selectable wells to the user after they have made additional classifications that would increase or decrease the pool of crystal classified images.

property well_volume
Returns the well volume set by the user modifed by whatever well volume unit is currently selected.

property x_reagent
Used to retrieve the `Reagent` object that is to be varied along the x axis of the screen.

property x_step
The percent variance between x reagent wells.

property x_wells
Returns spinBox value that representing the number wells on the x axis of the screen.

property y_reagent
Used to retreive the `Reagent` object that is to be varied along the y axis of the optimization plate

property y_step
The percent variance between y reagent wells.

property y_wells
Returns spinBox value representing the number of wells on the y axis of the screen.

polo.widgets.plate_inspector_widget module

class polo.widgets.plate_inspector_widget.**PlateInspectorWidget**(*parent*, *run=None*)
Bases: `PyQt5.QtWidgets.QWidget`

_apply_color_mapping()
Applies the current color mapping to displayed images. Images are colored based on either their human or marco classification.

_apply_image_filters()
Wrapper function around *plateViewer* `deemphasize_filtered_images()` which changes the opacity of currently displayed images based on their classifications.

_parse_label_checkboxes()
Private method that reads values from `QCheckBox` instances related to image filtering. Returns a dictionary where keys are the labels of the `QCheckBox` instances which should also be the possible image classifications and values are the state of the `QCheckBox` (True or False).

Returned dictionary will have following structure.

```
{
'Crystals': True,
'Clear': False,
'Precipitate': True,
'Other': False
}
```

Returns Dict of `QCheckBox` states.

Return type dict

_set_alt_spectrum_buttons()
Private helper function similar to `_set_time_resolved_buttons()` that determines if the navigation button that allows users to view alt spectrum images should be enabled. If conditions are not met then the button is disabled.

_set_color_comboboxs()
Private method that sets the label text associated with each color selector `QComboBox`. Should be called in the *__init__* method before the widget is shown to the user.

_set_color_options()
Private methods that uses the `COLORS` constant to set the color options for each color selector `QComboBox` instance in the image coloring tab.

_set_current_page(*page_number*)
Set the current page number and show the view for that page by calling `show_current_page()`

Parameters **page_number** (*int*) – The new page number

_set_image_count_options()
Private method to be called in the *__init__* method that sets the allowed number of images per page.

_set_images_per_page()
Private method that tells the `plateViewer` UI widget to set its *images_per_page* atttribute to the value specified in the images per page `QComboBox`.

_set_plate_label()
Private method to change the plate label to tell the user what view or "page" they are currently looking at.

_set_spin_box_range()
Set the allowed range for the page navigation spinbox.

_set_time_resolved_buttons()
Private helper function that determines if navigation buttons that display alt spectrum images, previous and next date images can be used.

apply_plate_settings()
Parses `QCheckBox` instances in the Plate View tab to determine what behavior of the `plateViewer` widget is requested by the user.

property color_mapping
Creates a color mapping dictionary that reflects the currently selected color selector `QComboBox` instances. The dictionary maps each image classifications to a `QColor` instance that can then be used to color images in the plate viewer.

property favorite
Status of the *favorite* `QCheckBox` filter.

Returns State of the favorite `QCheckBox`

Return type bool

property human
Status of human image classification QCheckBox.

Returns State of the human filter QCheckBox

Return type bool

images_per_page = [16, 64, 96]

property marco
Status of marco image classification QCheckBox.

Returns State of the marco filter QCheckBox

Return type bool

reset_all()
Method to un-check all user selected settings.

property run

property selected_classifications
Image classifications that are selected via the image filtering QCheckBox instances. Also see the image_type_checkboxes property.

Returns List of currently selected image classifications. Images who's classification is in this list should be shown / emphasized to the user.

Return type list

set_aspect_ratio_mode()
Sets the *preserve_aspect* attribute based on the status of the preserve aspect ratio QCheckBox. Preserving the aspect ratio results in displaying undistorted crystallization images but utilizes available display space less efficiently.

show_current_plate(*next_view=False*, *prev_view=False*, *next_date=False*, *prev_date=False*, *alt_spec=False*)
Show the images belonging to the current plate view to the user.

Parameters

- **next_date** (*bool, optional*) – Flag, if True show equivalent images from future date, defaults to False
- **prev_date** (*bool, optional*) – Flag, if True shows equivalent images from past date, defaults to False
- **alt_spec** (*bool, optional*) – Flag if True shows equivalent images in alternative imaging spectrum, defaults to False

polo.widgets.plate_viewer module

class polo.widgets.plate_viewer.**PlateGraphicsItem**(*pixmap*, *parent=None*)
Bases: PyQt5.QtWidgets.QGraphicsPixmapItem

contextMenuEvent(*self*, *QGraphicsSceneContextMenuEvent*)

class polo.widgets.plate_viewer.**plateViewer**(*parent*, *run=None*, *images_per_page=24*)
Bases: PyQt5.QtWidgets.QGraphicsView

_get_visible_wells(*page=None*)
Return indices of images that should be shown in the current page. A page is equivalent to a subsection of a larger screening plate.

Parameters **page** (*int, optional*) – Page number to find images for, defaults to None

Yield image index

Return type int

_make_image_label(*image*, *label_dict*, *font_size=35*)
Private helper method for creating label strings to overlay onto each image in the view.

Parameters

- **image** (Image) – Image to create label from
- **label_dict** (*dict*) – Dictionary of image attributes to include in the label
- **font_size** (*int, optional*) – Font size for the label, defaults to 35

Returns QGraphicsTextItem with label text set

Return type QGraphicsTextItem

_set_prerender_info(*item*, *image*)
Private helper method that sets flags and the tooltip for GraphicsItems before they are added to the GraphicsScene.

Parameters

- **item** (*QGraphicsItem*) – GraphicsItem that is to be added to the scene
- **image** (Image) – Image who's data will be used to label the GraphicsItem

Returns QGraphicsItem

Return type QGraphicsItem

property aspect_ratio
Current "best" aspect ratio for the view given the size of the view and the number of images that need to be fit into the view.

Returns Dimensions of the image grid, in images

Return type tuple

changed_images_per_page_signal

changed_page_signal

property current_page
Current page

Returns Current page

Return type int

decolor_all_images()
Removes all coloring from images in the *_scene* attribute.

emphasize_all_images()
Returns the opacity of all images in the *_scene* attribute to 1, or fully opaque.

export_current_view(*save_path=None*)
Exports the current content of the QGraphicsScene *_scene* attribute to a png file.

Parameters **save_path** (*str or Path, optional*) – Path to save the image to, defaults to None. If kept as None opens a QFileDialog to get a save file path.

fitInView(*scene*, *preserve_aspect=False*)
Fit items added to *_scene* attribute into the available display space.

Parameters

- **scene** (*QGraphicsScene*) – QGraphicsScene to fit
- **preserve_aspect** (*bool, optional*) – If True, preserves the aspect ratio of item is the scene, defaults to False

property images_per_page
Number of images in the current page.

Returns Number of images

Return type int

pop_out_selected_well()
Helper method to handle image selection and open an ImagePopDialog that displays the selected image in a pop out view.

property run
The current run being displayed.

Returns The current run

Return type *HWIRun*

set_scene_colors_from_filters(*color_mapping*, *strength=0.5*, *human=False*)
Set the color of images based on their current classifications. Very similar to `set_opacity_from_filters()`. Images can be colored by their MARCO or human classification.

Parameters

- **color_mapping** (*dict*) – Dictionary that maps image classifications to QColors
- **strength** (*float, optional*) – Image color strength, defaults to 0.5
- **human** (*bool, optional*) – If True, use the human classification to color images, defaults to False

set_scene_opacity_from_filters(*image_types*, *human=False*, *marco=False*, *favorite=False*, *filtered_opacity=0.2*)
Sets the opacity of all items in the current scene (*_scene* attribute) based on image filtering criteria. Allows for highlighting images that meet specific qualifications such has having a MARCO classification of crystals. Images that do not meet the set filter requirements will have their opacity set to the value specificed by the *filtered_opacity* argument.

Parameters

- **image_types** (*set of list*) – Image classifications to select for.
- **human** (*bool, optional*) – If True, use human classification to determine image classification, defaults to False
- **marco** (*bool, optional*) – If True, use MARCO classification to determine image classification , defaults to False
- **favorite** (*bool, optional*) – If True, image must be favorited to be selected, defaults to False

- **filtered_opacity** (*float, optional*) – Set the opacity for images that do not need the filtering requirements, defaults to 0.2

subgrid_dict = {16: (4, 4), 64: (8, 8), 96: (8, 12), 1536: (32, 48)}

tile_images_onto_scene(*label_dict={}*)

Calculates images that should be shown based on the current page and the number of images per page. Then tiles these images into a grid, adding them to *_scene* attribute.

Parameters **label_dict** (*dict, optional*) – Dictionary of Image attributes to pass along to *_make_image_label()* to create image labels, defaults to {}

property total_pages

Total number of pages based on the number of images per page and the number of images in the current run.

Returns Number of pages

Return type int

property view_dims

Current view dimensions in pixels.

Returns Width and height of the view

Return type tuple

static well_index_to_subgrid(*i*, *c_r*, *c_c*, *p_r*, *p_c*)

Find the linear index of the subgrid that a particular index belongs to within a larger grid. For example, ou are given a list of length 16. The list is reshaped into a 4 x 4 2D list. We divide the new grid into 4 quadrants each 2 X 2 and label them with an index (0, 1, 2, 3). Given an index of the original list we want to find the subgrid it belongs to.

Parameters

- **i** (*int*) – Index of point to locate in the 1D list
- **c_r** (*int*) – Number of rows in each subgrid
- **c_c** (*int*) – Number of columns in each subgrid
- **p_r** (*int*) – Number or rows in the entire grid
- **p_c** (*int*) – Number of columns in the entire grid

Returns Index of the subgrid the index *i* belongs to

Return type int

wheelEvent(*event*)

Handle Qt wheelEvents by setting the *_zoom* attribute. Allows users to zoom in and out of the current view.

Parameters **event** (*QEvent*) – event

polo.widgets.plate_visualizer module

class polo.widgets.plate_visualizer.**PlateVisualizer**(*parent=None*)
Bases: PyQt5.QtWidgets.QGraphicsView

The PlateVisualizer is a small widget to assist users understand what part of the screening plate they are currently viewing. It renders a grid of rectangles (blocks) that each represent one view (page) in the *PlateInspector* widget. The page that is currently being viewed is highlighted to show the user what part of the plate they are looking at.

Parameters **parent** (*QWidget, optional*) – Parent widget, defaults to None

_block_size(*x*, *y*)
Private method to calculate the size of individual blocks to render in the QGraphicsView.

Parameters

- **x** (*int*) – Length of x-axis in blocks
- **y** (*int*) – Length of y-axis in blocks

Returns tuple, length of block x-axis in pixels, length of block y-axis in pixels

Return type tuple

_handle_block_selection()
Private helper method to handle when a user selects a block. In theory should open the view that the selected block corresponds to but currently having some issues with this causing segmentation faults so it is disabled for now.

_highlight_block(*block*)
Private method that highlights a block in the QGraphicsScene.

Parameters **block** (*QGraphicsRectItem*) – Block to highlight

static block_dims(*plate_x*, *plate_y*, *grid_x*, *grid_y*)
Helper method to calculate the size of plate section blocks

Parameters

- **plate_x** (*int*) – Number of wells plate has on its x axis
- **plate_y** (*int*) – Number of wells plate has on it s y axis
- **grid_x** (*int*) – Number of wells in the subgrid on its x axis
- **grid_y** (*int*) – Number of wells in the subgrid on its y axis

Returns tuple, first item being length of x axis in blocks and second being length of y axis in blocks

Return type tuple

default_brush = <PyQt5.QtGui.QBrush object>

default_pen = <PyQt5.QtGui.QPen object>

plate_size = (32, 48)

plate_view_requested

selected_brush = <PyQt5.QtGui.QBrush object>

set_selected_block(*block_id*)
Sets the currently selected block based on its ID.

Parameters **block_id** (*int*) – Block ID

setup_view(*grid_cords*, *plate_size=None*)
set up the intail view based on the current plate size (normally 32 * 48 wells for 1536 well plate) and the subgrid size in wells.

Parameters

- **grid_cords** (*tuple*) – Subgrid size tuple (x, y) in wells
- **plate_size** (*tuple, optional*) – Size of entire plate (x, y) in wells, defaults to None. If None used the default 1536 well plate size of 32 * 48.

polo.widgets.run_organizer module

class polo.widgets.run_organizer.**RunOrganizer**(*parent=None*, *auto_link_runs=True*)
Bases: PyQt5.QtWidgets.QWidget

Widget for organizing and importing runs into Polo.

Parameters

- **parent** (*QWidget, optional*) – Parent widget, defaults to None
- **auto_link_runs** (*bool, optional*) – If True runs are automatically linked as they are loaded in, defaults to True

_add_run_from_directory(*dir_path*)
Private method to add a run to the runTree from a directory path.

Parameters **dir_path** (*str or Path*) – Path to directory to import

Returns Run or HWIRun created from directory if successful

Return type *Run* or *HWIRun*

_add_runs_to_tree(*runs*)
Private method to add a set of runs to the runTree.

Parameters **runs** (*list*) – List of runs to add to the runTree

_check_for_existing_backup(*run*)
Check the directory specified by the *BACKUP_DIR* constant for a backup mso file that matches the run passed through the *run* argument. Run's are matched to mso backups by their run name so it the user has renamed their run after the backup is saved it will not be found.

See *backup_classifications()* for details on how the mso files are written.

Parameters **run** (HWIRun) – Run to search for mso backup with

Returns Path to mso backup if one exists that matches the *run*, else return None

Return type str or None

_clear_current_run(*run_list*)
Clear out the current run from other widgets by emiting a *opening_run* signal with a list that does not contain a Run or HWIRun object.

Parameters **run_list** (*list*) – List of runs

_handle_classification_request()
Private method to open a classification thread of the currently selected run. Calls *_open_classification_thread()* to start the classification thread.

_handle_opening_run(**args*)
Private method that signal to other widgets that the current run should be opened for analysis and viewing by emiting the *opening_run* signal containing the selected run.

_open_classification_thread(*run*)
Private method to create and run a classification thread which will run the MARCO model on all images in the run passed to *run* argument.

Parameters **run** (*Run or HWIRun*) – Run or HWIRun instance to run MARCO on

_remove_run(*run=None*)
Private method to completely remove a run from Polo.

Parameters **run** (*Run or HWIRun, optional*) – Run to remove from current session, defaults to None

_set_estimated_classification_time(*time*, *num_images_remain*)
Display the estimated classification time to the user. Time remaining is calculated by multiplying the time it took to classify a representative image by the number of images that remain to be classified.

Parameters

- **time** (*int*) – Time to classify latest image
- **num_images_remain** (*int*) – Number of images that still require classification

_set_progress_value(*val*)
Private helper method to increment the classification progress bar.

Parameters **val** (*int*) – Value to set progress bar to

add_run_from_directory(*dir_path*)

backup_classifications(*run*)
Write the human classifications of the images in the *run* argument to an mso file and store it in the directory specified by the BACKUP_DIR constant. Does not store MARCO classifications because these can be much more easily recreated than human classifications. Additionally, when a run is loaded back in and a backup mso exists for it Polo assumes the classifications in that mso file are human classifications.

Currently only HWIRun instances can be written as mso files because of mso's integration with cocktail data and well assignments. Need a different format for non-HWI runs that would map filenames to classifications and ignore cocktail data / well assignments.

Parameters **run** (*HWIRun*) – HWIRun to backup human classifications

backup_classifications_on_thread(*run*)
Does the exact same thing as *backup_classifications()* except excutes the job on a *QuickThread* instance to avoid slow computers complaining about the GUI being frozen. This has been especially prevelant on Windows machines.

Parameters **run** (*HWIRun*) – Run to save as mso file

classify_run

finished_ftp_download()

ftp_download_status

handle_ftp_download(*file_path*)

import_run_from_dialog()
Import a run from a file dialog.

import_run_from_ftp()
Import runs from an FTP server. If an FTP download thread is not already running creates an FTPDialog instances and opens it to the user. FTP functions are then taken over by the FTPDialog until it is closed.

import_saved_runs(*xtal_files=[]*)
Import runs saved to xtal files.

Parameters **xtal_files** (*list, optional*) – List of xtal files to import runs from, defaults to []

opening_run

remove_run(*run=None*)

polo.widgets.run_tree module

class polo.widgets.run_tree.**RunTree**(*parent=None*, *auto_link=True*)
Bases: PyQt5.QtWidgets.QTreeWidget

Inherits the QTreeWidget class and acts as the sample and run display. The User uses the RunTree to open and classify runs they load into Polo.

Parameters

- **parent** (*QWidget, optional*) – Parent widget, defaults to None
- **auto_link** (*bool, optional*) – If True automatically link runs together, defaults to True

_add_classifications_from_mso_slot(*event=None*)
Add classifications to an existing Run from the contents of an MSO file. Intended to be connected to the *classify_from_mso* QAction that is defined in the : *contextMenuEvent()* method.

Parameters **event** (*QEvent, optional*) – QEvent, defaults to None

_add_run_node(*run*, *tree=None*)
Private method that adds a new run node.

Parameters

- **run** (*Run or HWIRun*) – Run to add to the tree
- **tree** (*QTreeWidgetItem, optional*) – *QTreeWidgetItem* to act as parent node, defaults to None. If None uses the root as the parent node.

Returns Node added to the tree

Return type QTreeWidgetItem

_edit_data_slot(*event=None*)
Private method used to update the data in a Run after it has been modified by the user through the RunUpdater dialog.

Parameters **event** (*QEvent, optional*) – QEvent, defaults to None

_get_run_node(*run*)
Private helper method that returns the QTreeWidgetItem corresponding to a given Run. Returns None if a node cannot be found.

Parameters **run** (*Run or HWIRun*) – Run to search for

Returns Given run's corresponding QTreeWidgetItem if it exists

Return type QTreeWidgetItem

_open_run_slot(*event=None*)
Private method that emits the *opening_run* signal when called. This signal can be connected to other widgets to communicate that the user has selected a run and wants to open it for analysis.

Parameters **event** (*QEvent, optional*) – QEvent, defaults to None

_remove_run(*run_name*)

Private method to remove a Run completely from the Polo interface.

Parameters **run_name** (*str*) – Run name of Run instance to remove

_remove_run_slot(*event=None*)
Slot to connect to contextMenu popup to remove the selected run.

add_classified_run(*run*)
Marks a Run instance as classified by adding it to the classified_status dictionary.

Parameters **run** (*Run or HWIRun*) – Run to mark as classified

add_run_to_tree(*new_run*)
Add a new Run instance to the tree. Uses the Run instance's sampleName attribute to determine what sample node the Run instance should be added to. If the sample name does not exist in the tree a new sample node is added. If the Run instance lacks the sampleName attribute as is the case for non-HWIRuns the sampleName attribute is set to "Non-HWI Runs". If the Run instance is an HWIRun and lacks the sampleName attribute sampleName is set to "Sampleless Runs".

Parameters **new_run** (*Run or HWIRun*) – Run to add to the tree

add_sample(*sample_name*, **args*)
Adds a new sample to the tree. Samples are the highest level node in the *RunTree*.

Parameters **sample_name** (*str*) – Name of sample to add, acts as key so should be unique.

contextMenuEvent(*event*)
Handle left click events by creating a popup context menu.

Parameters **event** (*QEvent*) – QEvent

property current_run_names
List of all currently loaded Run names.

Returns List of :class:Run` names

Return type list

link_sample(*sample_name*)
Links all Run instances in a given sample together by both date and spectrum using the *the_big_link()* method.

Parameters **sample_name** (*str*) – Name of the sample who's runs should be linked

opening_run

remove_run_from_view(*run_name*)
Remove a Run instance using its run_name attribute. Does not effect any other widgets. Calling this method only removes the Run instance from the display. If a Run instance is removed from successfully it is returned.

Parameters **run_name** (*str*) – Name of run to remove

Returns Removed run

Return type *Run* or *HWIRun*

remove_run_signal

save_run_signal

property selected_run
The: class:*Run* that is currently selected. If no `Run` is selected returns False.

Returns The currently selected run, if one exists, otherwise returns False

Return type *Run*, *HWIRun* or False

polo.widgets.slideshow_inspector module

class polo.widgets.slideshow_inspector.**slideshowInspector**(*parent*, *run=None*)
Bases: `PyQt5.QtWidgets.QWidget`

The slideshowInspector widget is a primary run interface that allows users to view their screening images in a standard slideshow format. If multiple imaging runs of the sample sample exist it also allows the user to navigate between or simultaneously view these images.

Parameters

- **parent** (*QtWidget*) – Parent widget
- **run** (*Run or HWIRun, optional*) – Run to show to the user, defaults to None

_classify_image(*classification*)
Private method to change the human classification of the current image.

Parameters **classification** (*str*) – Image classification

_display_current_image()
Private method that displays the current image as determined by the *current_image* attribute of the *slideshowViewer* widget and populates any widgets that display current image metadata.

_mark_current_image_as_favorite()
Private method that sets the favorite label on the current image to the current value of the favorite `QCheckBox`.

Parameters **favorite_status** (*bool*) – Whether this image is a favorite or not

_navigate_carousel(*next_image=False*, *prev_image=False*)
Private method to control the carousel using boolean flags. Calls `carousel_controls()`.

Parameters

- **next_image** (*bool, optional*) – If True navigates to next image in carousel, defaults to False
- **prev_image** (*bool, optional*) – If True navigates to previous image in carousel, defaults to False

_set_alt_image(*next_date=False*, *prev_date=False*, *alt_spec=False*)
Display an image linked to the current image based on boolean flags.

Parameters

- **next_date** (*bool, optional*) – If True show the current image's next image by date, defaults to False
- **prev_date** (*bool, optional*) – If True, show the current image's previous image by date, defaults to False

- **alt_spec** (*bool, optional*) – If True, show the current image's alt spectrum image, defaults to False

_set_alt_spectrum_buttons()
Turns alt spectrum functions on or off depending on contents of the *Run* instance referenced by the *run* attribute. Alt spectrum buttons will be enabled if the *run* is a part of an alt spectrum linked list. This means another *Run* instance is referenced by it's *alt_spectrum* attribute.

_set_classification_button_labels()
Private method that sets the labels of image classification buttons based on the IMAGE_CLASSIFICATIONS constant. Should be called in the *__init__* method.

_set_favorite_checkbox()
Private method that sets the value of the favorite QCheckBox based on whether the current image is marked as a favorite or not. Should be used when loading an image into the view.

An image is considered a favorite if it's *favorite* attribute == True.

_set_image_class_checkbox_labels()
Private method to the QCheckBox labels for imaging filtering from the *IMAGE_CLASSIFICATIONS* constant. Should be called in the *__init__* method.

_set_image_name()
Private method that sets current image label to the image's filepath basename.

_set_slideshow_mode(*show_all_dates=False*, *show_all_specs=False*)
Private method to set the slideshowViewer mode. Either to display a single image, all dates or all spectrums.

Parameters

- **show_all_dates** (*bool, optional*) – If True sets slideshowViewer to show all dates, defaults to False
- **show_all_specs** (*bool, optional*) – If True sets slideshowViewer to show all spectrums, defaults to False

_set_time_resolved_functions()
Private method that turns time resolved functions on or off depending on contents of the *Run* instance referenced by the *run* attribute. Time resolved functions are enabled when the *run* is part of a time resolved linked list. This means another *Run* instance is referenced by it's *next_run* and / or *previous_run* attributes.

_show_image_from_well_number(*well_number*)
Private method to display an image by well number.

Parameters **well_number** (*int*) – Well number of image to display

_submit_filters()
Private method that passes the current user selected image filters to the slideshowViewer so the current slideshow contents can be adjusted to reflect the new filters. Displays the current image after filtering.

property current_image
Current *Image* object being displayed in the *slideshowViewer* widget.

Returns The current image

Return type *Image*

property current_sort_function
Return a function to use for image sorting based on current user radiobutton sort selections.

Returns Sort function

Return type func

export_current_view()
Export the current view to a png file. Show the user a message box to tell them if the export succeeded or failed.

property favorites
Returns the state of the favorite `QCheckBox`.

Returns Favorite `QCheckBox` state

Return type bool

property human
State of the human classifier `QCheckBox`. If True, assume the user wants their selected image classifications to be in reference to image's human classification.

Returns State of the `QCheckBox`

Return type bool

property marco
State of the MARCO classifier `QCheckBox`. If True, assume the user wants their selected image classifications to be in reference to image's MARCO classification.

Returns State of the `QCheckBox`

Return type bool

property run

property selected_classifications
Returns image classification keywords for any image classification `QCheckBox` instances that are checked.

Returns List of selected images classifications

Return type list

static sort_images_by_cocktail_number(*images*)
Helper method that sorts a collection of images by their cocktail number. Returns False if the images cannot be sorted by this parameter.

Parameters **images** (*list*) – List of images to be sorted

Returns List of images sorted by cocktail number, False if cannot be sorted

Return type list, bool

static sort_images_by_marco_confidence(*images*)
Helper method to sort a collection of images by their MARCO classification confidence. Does not descriminate based on image classification.

Parameters **images** (*list*) – List of images to sort

Returns List of images sorted by prediction confidence

Return type list

static sort_images_by_well_number(*images*)
Helper method to sort a collection of images by their well number. If images cannot be sorted by well number (which in theory shouldn't happen) returns False

Parameters **images** (*list*) – List of images to be sorted

Returns List os images sorted by well number

Return type list

polo.widgets.slideshow_viewer module

class polo.widgets.slideshow_viewer.**Carousel**
Bases: object

The Carousel class handles navigation between *Slide* instances.

add_slides(*ordered_images*, *sort_function=None*)
Sets up linked list consisting of nodes of Slide instances. The list is circular and bi-directional. Sets self.current_slide to the first slide in the linked list. The order of the slides in the linked list will reflect the order of the images in the *ordered_images* argument.

Parameters **ordered_images** – a list of Image objects to create the linked list from. The order of the images will be reflected by the linked list.

Returns First slide in linked list

Return type *Slide*

controls(*next_slide=False*, *prev_slide=False*)
Controls the navigation through the slides in the carousel. Does not control access to alternative images that may be available to the user.

Parameters

- **next_slide** (*bool*) – If set to True, tells the carousel to advance one Slide
- **prev_slide** (*bool*) – If set to True, tells the carousel to retreat by one Slide

property current_slide
Current slide, the slide that should be displayed to the user.

Returns The current slide

Return type *Slide*

class polo.widgets.slideshow_viewer.**PhotoViewer**(*parent*)
Bases: PyQt5.QtWidgets.QGraphicsView

add_pixmap(*pixmap*)
Adds a *Pixmap* instances to the current sene.

Parameters **pixmap** (*Pixmap*) – Pixmap to add to the sene

fitInView(*self*, *QRectF*, *mode: Qt.AspectRatioMode = Qt.IgnoreAspectRatio*)
fitInView(self, QGraphicsItem, mode: Qt.AspectRatioMode = Qt.IgnoreAspectRatio) fitInView(self, float, float, float, float, mode: Qt.AspectRatioMode = Qt.IgnoreAspectRatio)

hasPhoto()

mousePressEvent(*event*)
Handles mouse press events.

Parameters **event** (*QEvent*) – Mouse press event

photoClicked

toggleDragMode()
Turns drag mode on and off.

wheelEvent(*event*)
Handles mouse wheel events to allow for scaling for zooming in and out of the currently displayed image.

Parameters **event** (*QEvent*) – Mouse scroll wheel event

class polo.widgets.slideshow_viewer.**Slide**(*image*, *next_slide=None*, *prev_slide=None*, *slide_number=None*)

Bases: object

Acts like a slide in a slideshow carousel. Holds an Image object instance as the contents of the slide. Forms a linked list with other slides through the *next_slide* and *prev_slide* attributes which act as the forwards and backwards pointers to other slides.

Parameters

- **image** (Image) – Image that this slide will display
- **next_slide** (Slide, *optional*) – Next slide in the slideshow, defaults to None
- **prev_slide** (Slide, *optional*) – Previous slide in the slideshow, defaults to None
- **slide_number** (*int*, *optional*) – Index of this slide in the slideshow, defaults to None

class polo.widgets.slideshow_viewer.**SlideshowViewer**(*parent*, *run=None*, *current_image=None*)

Bases: *polo.widgets.slideshow_viewer.PhotoViewer*

_add_text_to_scene(*text*, *x*, *y*, *size=40*)

Private method to add text on top of an image. Adds the text to the current scene at the position specified by the *x* and *y* arguments.

Parameters

- **text** (*str*) – Text to add to image
- **x** (*int*) – X cordinate of text
- **y** (*int*) – Y cordinate of text
- **size** (*int*, *optional*) – Size of text, defaults to 40

_set_all_dates_scene(*image*)

Private method that creates a time resolved view from the *Image* instance passed through the *image* argument.

Parameters **image** (Image) – Image to create time resolved view from

_set_all_spectrums_scene(*image*)

Private method that creates a view that includes all alt spectrum images the *Image* instance is linked to.

Parameters **image** (Image) – Image to create the view from

_set_single_image_scene(*image*)

Private method that creates a standard single image view from the *Image* instance passed to the *image* argument.

Parameters **image** (Image) – Image to display

arrange_multi_image_scene(*image_list*, *render_date=False*)

Helper method to arrange multiple images into the same view.

Parameters

- **image_list** (*list*) – List of images to add to the view
- **render_date** (*bool*, *optional*) – If True adds a date label to each image, defaults to False

carousel_controls(*next_image=False*, *previous_image=False*)
Wrapper around the *controls()* method that allows image navigation. Does not actually display the image.

Parameters

- **next_image** (*bool*) – If True, tells carousel to advance by one slide.
- **previous_image** (*bool*) – If True, tells carousel to retreat by one slide.

:returns The current image. :rtype Image

classify_current_image(*classification*)
Changes the human classification of the current image.

display_current_image()
Renders the Image instance currently stored in the *current_image* attribute.

get_cur_img_cocktail_str()
Retruns the *current_image* cocktail information as a string.

Returns Cocktail information string

Return type str

get_cur_img_meta_str()
Returns the *current_image* metadata as a string.

Returns Metadata string

Return type str

photoClicked
Wrapper class around QGraphicsView and displays image to the user in the slideshow viewer tab of the main window.

Parameters

- **run** (Run) – Current run whose images are to be shown by the viewer.
- **parent** (*QWidget*) – Parent Widget of this instance.
- **current_image** (Image) – Image that is currently displayed by the viewer.

property run

set_alt_image(*next_date=False*, *prev_date=False*, *alt_spec=False*)
Sets the *current_image* attribute to a linked image specified by one of the three boolean flags.

Parameters

- **next_date** – If True, sets the *current_image* to the next image by date
- **prev_date** – If True, sets the *current_image* to the previous image by date
- **alt_spec** – If True, sets the *current_image* to an alt spectrum image

set_current_image_by_well_number(*well_number*)
Set the current image to the *Image* instance associated with a specific well number.

Parameters **well_number** (*int*) – Well number to display

update_slides_from_filters(*image_types*, *human*, *marco*, *favorite=False*, *sort_function=None*)
Creates new *Carousel* slides based on selected image filters. Sets the *current_image* attribute to the *Image* instance at the the *current slide* attribute of *_carousel* attribute.

Parameters

- **image_types** (*set or list*) – Set of image classifications to include in results.
- **human** (*bool*) – If True, *image_types* refers to human classification of the image.
- **marco** (*bool*) – If True, *image_types* refers to the machine (MARCO) classification of the image.

polo.widgets.table_inspector module

class polo.widgets.table_inspector.**TableInspector**(*parent=None*, *run=None*)
Bases: PyQt5.QtWidgets.QWidget

TableInspector class displays a run's data in a spreadsheet type view.

Parameters

- **parent** (*QWidget, optional*) – Parent widget, defaults to None
- **run** (Run *or* HWIRun, *optional*) – Run to display in the table, defaults to None

_assign_checkboxes_to_class()
Private method that assigns filtering checkboxs to an image classification from the *IMAGE_CLASSIFICATION* constant.

_set_column_options()
Private method that sets the availabe columns to display based on the attributes of the run stored in the *run* attribute. Adds a QCheckBox widget for each attribute.

TODO: formating for private attributes to make them prettier

property run

property selected_classifications
Return image classifications that are currently selected.

Returns List of selected image classifications

Return type set

property selected_headers
Return the headers that have been selected by the user.

Returns Names of column headers that are currently selected

Return type set

update_table_view()
Private method that updates the data being displayed in the tableViewer.

polo.widgets.table_viewer module

class polo.widgets.table_viewer.**TableViewer**(*parent*, *run=None*)
Bases: PyQt5.QtWidgets.QTableWidget

TableViewer instances override QTableWidget and provide a more convenient interface for translating the data in *Run* and *HWIRun* objects into a table format.

Parameters

- **parent** (*QtWidget*) – Parent widget
- **run** (Run, *optional*) – Run to show in this table view, defaults to None

property fieldnames
Return the fieldnames for the current run. Should only be used when setting the values for the listWidget in a *tableInspector* instance as is expensive to call.

Returns list of fieldnames

Return type list

static filter(*row*, *image_classes*, *human*, *marco*)
Helper method to determine if a row should be included based on the image filters the user has selected

Parameters

- **row** (*dict*) – row data
- **image_classes** (*set or list*) – types of images to include, i.e Crystals, Clear
- **human** (*Bool*) – If image_classes should be in reference to human classifier
- **marco** (*Bool*) – If image_classes should be in reference to machine classifier

Returns If image should be filtered, False means do not filter image

Return type Bool

make_header_map(*headers*)
Helper method to map header keywords to their index (order). This method is required as headers are delivered as a set and we want them to be presented in a consistent order to the user.

Parameters **headers** (*set*) – Set of headers strings

Returns Dictionary of header strings mapped to indices

Return type dict

populate_table(*image_classes*, *human*, *marco*)
Populates the table and displays data to the user based on their header and image filtering selections.

Parameters

- **image_classes** (*set or list*) – types of images to include, i.e Crystals, Clear
- **human** (*Bool*) – If image_classes should be in reference to human classifier
- **marco** (*Bool*) – If image_classes should be in reference to machine classifier

Returns If image should be filtered, False means do not filter image

property run
Return the run object

Returns Run object

Return type *Run*

property table_data
Property to retrieve the current table fieldnames and table data using *get_csv_data* function of the *RunCsvWriter* class.

Returns fieldnames, table data

Return type tuple

polo.widgets.unit_combo module

class polo.widgets.unit_combo.**UnitComboBox**(*parent=None*, *base_unit=None*, *scalers={}*)
Bases: PyQt5.QtWidgets.QWidget

Widget that is a combination of a spinbox and a comboBox that allows a user to select a value using the spinBox and a unit using the comboBox.

Example:

Lets say we want to create a UnitComboBox that allows someone to select a Molar concentration using micro-molar, milli-molar, centi-molar or molar.

```
# create the scaler dictionary
s = {
    'u': 1e-6, 'm': 1e-3, 'c': 1e-2
}
# values are in reference to the base unit
unit_combo = UnitComboBox(
    parent=None, base_unit='M', scalers=s
    )
```

Parameters

- **parent** (*QWidget, optional*) – Parent widget, defaults to None
- **base_unit** (*str, optional*) – Base unit string, defaults to None
- **scalers** (*dict, optional*) – Dictionary of prefixes to apply to the baseunit. Keys should be string prefixes and values should be value that scales the baseunit, defaults to {}

_set_unit_combobox_text()
Private method to add units to the unit comboBox based on the *base_unit* and the *scalers* attributes.

Returns Items added to the comboBox

Return type list

get_value()
Return a UnitValue constructed from the value of the spinBox value and unit from the comboBox.

Returns UnitValue constructed from current spinBox value and comboBox unit

Return type *UnitValue*

saved_scalers = {'c': 0.01, 'm': 0.001, 'u': 1e-06}

property scalers
The current scalers.

Returns List of scaler values

Return type list

set_value(*value*, **args*)
Set the spinBox value and the comboBox unit based on the value and unit of a *UnitValue* instance

Parameters **value** (UnitValue) – UnitValue

set_zero()
Set the spinbox value to 0

property sorted_scalers
Scalers sorted by their magnitude.

Returns List of scalers

Return type list

property unit_combobox_text
The text in unit comboBox which corresponds to a specific scaler.

Returns List of all scalers in the unit comboBox

Return type list

unit_text_parser(*unit_text=None*)

Module contents

6.1.6 polo.windows package

Submodules

polo.windows.ftp_dialog module

class polo.windows.ftp_dialog.**FTPDialog**(*ftp_connection=None*, *parent=None*)
Bases: PyQt5.QtWidgets.QDialog

FTPDialog class acts as the interface for interacting with a remote FTP server. Allows for browsing and downloading files stored on the server.

Parameters

- **ftp_connection** (*FTP, optional*) – Existing FTP connection, defaults to None
- **parent** (*QWidget, optional*) – Parent widget, defaults to None

CONNECTED_ICON = '/home/ethan/Documents/github/Marco_Polo/src/data/images/icons/connec

DISCONNECTED_ICON = '/home/ethan/Documents/github/Marco_Polo/src/data/images/icons/dis

DOWNLOAD_ICON = '/home/ethan/Documents/github/Marco_Polo/src/data/images/icons/downloa

connect_ftp()
Attempt to establish a connection to an ftp server. If the connection is successful then recursively walk through the user's home directory and display available directories and files via the *fileBrowser* widget. If the user has an extremely large number of files this can take a while. If the connection fails show the user the error code thrown by ftplib.

download_selected_files()
Signals to the *fileBrowser* widget to download all files / dirs the user has selected. Downloading occurs in the background and the FTP browser dialog is closed after a download has successfully begun. Another download should not be initiated while one is already in progress.

property host
Get user entered FTP host.

Returns host address

Return type str

property password
Return user entered password.

Returns password

Return type str

set_connection_status(*connected=False*)
Change the Qlabel that displays the current connection status to the user.

Parameters **connected** (*bool, optional*) – If FTP connection is successful, defaults to False

property username
Return username.

Returns username

Return type str

polo.windows.image_pop_dialog module

class polo.windows.image_pop_dialog.**ImagePopDialog**(*image*, *parent=None*)
Bases: PyQt5.QtWidgets.QDialog

Pop up that displays a selected image in a larger view. Intended to be used with the *PlateViewer* widget when a user selects an image from the grid.

Parameters **image** (Image) – Image to show

_change_favorite_status()
Private method that updates the favorite status of the current *image* attribute to the state of the favorite QRadioButton.

_set_allowed_navigation_functions()
Private method to enable or disable navigation by date or spectrum buttons based on the content of the current image. Tests the *Image* instance referenced by the *image* attribute to determine if it is linked to a future date, previous date or alt spectrum image through it's next_image , previous_image and alt_image attributes respectively. If an attribute == None, then the button that requires that attribute will be disabled.

_set_cocktail_details()
Private method that shows the *image* attribute metadata in the text display widgets.

_set_groupbox_title()
Private method that set the the title of main groupbox to the basename of the *path* attribute of the *Image* instance referenced by the *image* attribute.

_set_image_details()
Private method that displays the *Image* instance referenced by the *image* attribute.

classify_image(*crystals=False*, *clear=False*, *precipitate=False*, *other=False*)
Set the human classification of the *Image* instances referenced by the *image* attribute.

Parameters

- **crystals** (*bool, optional*) – If True, classify the *image* as crystal, default False
- **clear** (*bool, optional*) – If True, classify the *image* as clear, defaults to False
- **precipitate** (*bool, optional*) – If True, classify the *image* as precipitate, defaults to False
- **other** (*bool, optional*) – If True, classify as the *image* as other, defaults to False

property image

The *Image* being displayed.

Returns The *Image* instance to be displayed

Return type *Image*

show()
Shows the dialog window.

show_alt_image(*next_date=False*, *prev_date=False*, *alt=False*)
Show a linked image based on boolean flags.

Parameters

- **next_date** (*bool, optional*) – If True, set *image* attribute to next the available imaging date, defaults to False
- **prev_date** (*bool, optional*) – If True, set *image* attribute to previous imaging date, defaults to False
- **alt** (*bool, optional*) – If True, set *image* attribute to an alt spectrum image, defaults to False

show_image()
Show the *Image* instance referenced by the *image* attribute.

polo.windows.log_dialog module

class polo.windows.log_dialog.**LogDialog**(*parent=None*)
Bases: PyQt5.QtWidgets.QDialog

Small dialog for displaying the contents of the Polo log file.

clear_log()
Deletes the contents of the log file.

display_log_text()
Opens the log file and writes the contents to textBrowser widget for display to the user.

save_log_file()
Saves the current log file contents to a new location.

polo.windows.main_window module

class polo.windows.main_window.**MainWindow**
Bases: PyQt5.QtWidgets.QMainWindow, polo.designer.UI_main_window.Ui_MainWindow

QMainWindow that ultimately is the parent of all other included widgets.

BAR_COLORS = [15, 13, 14, 4]

CRYSTAL_ICON = '/home/ethan/Documents/github/Marco_Polo/src/data/images/icons/crystal.

_handle_delete_backups()
Private method that handles a user request to delete all backup mso files. If backups cannot be deleted shows a message box indicating failure to delete.

_handle_export(*action*, *export_path=None*)
Private method to handle when a user requests to export a run to a non-xtal file format.

Parameters

- **action** (*QAction*) – QAction that describes the export type the user has requested

- **export_path** (*str or Path, optional*) – Path to export file to, defaults to None

_handle_file_menu(*selection*)
Private method that handles user interaction with the file menu; this usually means saving a run as an xtal file.

Parameters **selection** (*QAction*) – QAction that describes user selection

_handle_help_menu(*action*)
Private method that handles user interaction with the help menu. All selections open links to various pages of the documentation website.

Parameters **action** (*QAction*) – QAction that describes the user's selection

_handle_image_import(*selection*)
Private method that handles when the user attempts to import images into Polo. Effectively a wrapper around other methods that provide the functionality to each option in the import menu.

Parameters **selection** – QAction. QAction from user menu selection.

_handle_opening_run(*new_run*)
Private method that handles opening a run. For the most part, this means setting the `run` attribute of other widgets to the *new_run* argument. The setter methods of these widgets should then handle updating their interfaces to reflect the new run being opened. Also calls *_tab_limiter()* and *_plot_limiter()* to set allowed functions for the user based on the type of run they open.

Additionally, if this is not the first run to be opened, before the *new_run* is set as the `current_run` the pixmaps of the `current_run` are unloaded to free up memory.

Parameters **q** (*list*) – List containing the run to be opened. Likely originating from the `RunOrganizer` widget.

_handle_plot_selection()
Private method to handle user plot selections.

TODO: Move all plot methods into their own widget

_handle_tool_menu(*selection*)
Private method that handles selection of all options available to the user in the *Tools* section of the main window menu.

Parameters **selection** (*QAction*) – User's menu selection

_on_changed_tab(*i*)
Private method that handles GUI behavior when a user switches from one tab to another.

Parameters **i** – Int. The index of the current tab, after user has changed tabs.

_plot_limiter()
Private method to limit the types of plots that can be shown based on the type of the *current_run*.

_save_file_dialog()
Private method to open a QFileDialog to get a location to save a run to.

Returns Path to save file to

Return type str

_set_tab_icons()
Private method that assigns icons to each of the main run interface tabs. Should be called in the *__init__* method before the main window is shown to the user.

_tab_limiter()
Private method that limits the interfaces that a user is allowed to interact with based on the type of `Run` they have loaded and selected. Currently, `Run` functionality is limited due to the fact cocktails cannot be mapped to images.

closeEvent(*event*)
Handle main window close events. Writes mso backup files of all loaded runs that have human classifications so they can be restored later.

Parameters **event** (*QEvent*) – QEvent

static delete_all_backups()
Deletes all backup mso files.

Raises **e** – Any exceptions thrown by the function call

Returns True, if backups are deleted

Return type bool

static get_widget_dims(*self*, *widget*)
Returns the width and height of a `QWidget` as a tuple.

Parameters **widget** (*QWidget*) – QWidget

Returns width and height of the widget

Return type tuple

static layout_widget_lister(*self*, *layout*)
List all widgets in a given layout.

Parameters **layout** (*QLayout*) – QLayout that contains widgets

Returns Tuple of widgets in the given layout

Return type tuple

polo.windows.pptx_dialog module

class polo.windows.pptx_dialog.**PptxDesignerDialog**(*runs*, *parent=None*)
Bases: `PyQt5.QtWidgets.QDialog`

_browse_and_update_line_edit()
Private method that calls `_get_save_path()` to open a file browser. If the user selects a save path using the file browser then displays this path in the filepath `QLineEdit` widget.

_get_save_path()
Private method that opens a file browser and returns the selected save filepath.

Returns Filepath if one is specified by the user, empty string otherwise

Return type str

_parse_image_classifications()
Private method to get all currently selected image classifications by reading the state of all image classification `QCheckBox` instances.

Returns Set of all selected image classifications

Return type set

_set_up_image_classification_checkboxes()
Private method that sets up the labels for the image classifications QCheckBox instances. Should be called in the *__init__* function before displaying the dialog to the user.

Returns Dictionary of image classifications which map to the QCheckBox that corresponds to that image classification

Return type dict

_validate_typed_path()
Private method that validates that a filepath in the filepath QLineEdit widget is actually a valid path that a pptx file could be saved there.

Returns True if the path is valid, otherwise returns None and shows a message box to the user.

Return type True or None

_write_presentation(*run=None*)
Private method that actually does the work of generating a presentation from a Run or HWIRun instance.

Parameters **run** (Run *or* HWIRun, *optional*) – Run to create a presentation from, defaults to None

Returns Path to the pptx presentation is write is successful, Exception otherwise.

Return type str or Exception

property all_dates
The state of the "Include all Dates" QCheckBox. If it is checked this indicates that a time resolved slide should be included in the presentation.

Returns State of the QCheckBox

Return type bool

property all_specs
The state of the "Include all Spectrums" QCheckBox. If it is checked this indicates that a multi-spectrum slide should be included in the presentation.

Returns State of the QCheckBox

Return type bool

check_for_warnings()

property human
State of the human classifier QCheckBox.

Returns State of the QCheckBox

Return type bool

property marco
State of the MARCO classifier QCheckBox.

Returns State of the QCheckBox

Return type bool

set_default_titles()

setup_run_tree()

property subtitle
Subtitle the user has entered for the presentation via the subtitle QLineEdit widget. If no string has been entered will return the empty string.

Returns The presentation subtitle

Return type str

property title
Title the user has entered for the presentation via the title `QLineEdit` widget. If no string has been entered will return the empty string.

Returns The presentation title

Return type str

polo.windows.run_importer module

class polo.windows.run_importer.**ImportCandidate**(*path*)
Bases: `object`

assign_run_type()
If the *path* attribute is verified as importable assigns a run class (Run or HWIRun) to the `import_type` attribute. Later on in the import pipeline this attribute tells other methods how the *ImportCandidate* should be imported as different operations are required to create *Run* instances then *HWIRun* instances.

The *ImportCandidate* is assigned to a HWIRun is its metadata is successfully parsed.

Returns The import type

Return type *Run* or *HWIRun*

property cocktail_menu
If the *ImportCandidate* instances's `import_type` attribute is the `HWIRun`class and the candidate has a valid date then returns a :class:`polo.utils.io_utils.Menu` instance that was in use at the *ImportCandidate* instanes's date. If anyof these conditions are not meet then returns None.

Returns *polo.utils.io_utils.Menu* or None

Return type *polo.utils.io_utils.Menu* or None

property is_rar
Import candidate is a rar file. True if the file is rar file, False otherwise.

Returns Rar status

Return type bool

property path
Return the *ImportCandidate* instances's path.

Returns Path to the file which will actually be imported

Return type str

read_xmldata(*dir_path*)
Attempts to read any xml metadata files in the path referenced by the `dir_path` argument.

Parameters **dir_path** (*str or Path*) – Path to check for xml files in

Returns Dictionary of data pulled from xml files. If no data is found then returns an empty dictionary.

Return type dict

set_cocktail_menu()

unrar()
If the *ImportCandidate* instances's *is_rar* property is True then this method will attempt to decompress the rar archive referenced by the instance's *path* attribute.

Returns Path to un-compressed rar archive if unrar was successful, otherwise returns False

Return type Path or str

verify_path()
Verifies that the path referenced by the *path* attribute could potentially be imported into Polo.

Returns True if *path* is verified, False otherwise

Return type bool

class polo.windows.run_importer.**RunImporterDialog**(*current_run_names*, *parent=None*)
Bases: PyQt5.QtWidgets.QDialog

RunImporterDialog instances are the user interface for importing runs from rar archives or directories stored on the local machine.

Parameters **current_run_names** (*list or set*) – Runnames that are already in use by the current Polo session (Run names should be unique)

HWI_INDEX = 0

NON_HWI_INDEX = 1

RAW_INDEX = 2

_add_import_candidates(*new_candidates*)
Adds *ImportCandidate* instances to the import_candidates attribute.

Parameters **new_candidates** (*list*) – List of *ImportCandidates*

_could_not_import_message(*prefix*, *paths*)
Private method that creates a message box popup for when imports fail.

Parameters

- **prefix** (*str*) – First part of the error message. Something like "Could not import the following files:"
- **paths** (*list*) – List of filepaths that could not be imported

Returns QMessageBox

Return type QMessageBox

_disable_hwi_import_tools()
Private method to disable widgets that should only be used for HWIRun imports.

_display_candidate_paths()
Private method that updates the dialog's QListWidget with the file paths of the current *ImportCandidate* instances referenced by the import_candidates attribute.

_display_cocktail_files(*menu_type=None*)
Private method that displays the available cocktail files to the user via the *Menu* QComboBox widget.

Parameters **menu_type** (*str, optional*) – Key for which kind of cocktail screens to display, defaults to None. "m" for membrane screens and "s" for soluble screens.

_enable_hwi_import_tools()
Private method to enable widgets that should only be used for HWIRun imports.

_handle_candidate_change()
Private method that calls *_update_selected_candidate()* and then *_populate_fields()*. This updates the data of the previously selected *ImportCandidate* if it has been changed and then updates data display widgets with the information from the currently selected *ImportCandidate* instance.

_import_files(*rar=True*)
Private method that attempts to import a collection of file paths specified by the user. If importing a rar archive the method creates a *polo.threads.thread.QuickThread* instance and runs all rar operations on that thread to avoid freezing the GUI on slower machines. Imported runs are added to the `import_candidates` attribute dictionary and then displayed to the user by calling `_display_candidate_paths()`.

Parameters **rar** (*bool, optional*) – If True opens the filebrowser for rar archives and filters out all other import types, defaults to True

_import_run(*import_candidate*)
Private helper method that is called by *_import_runs()* that attempts to import a run from an :class:`ImportCandidate.

Parameters **import_candidate** (`ImportCandidate`) – *ImportCandidate* to create run from

Returns Run or HWIRun if successful

Return type *Run* or *HWIRun*

_import_runs()
Private method that attempts to create run objects from all available *ImportCandidate* instances.

_open_browser(*rar=True*)
Private method that opens a `QFileBrowser` instance that allows the user to select files for import. The allowed filetype is set using the *rar* flag.

Parameters **rar** (*bool, optional*) – If True, allow user to only import Rar archive files defaults to True. If False only allows the user to import directories.

Returns List of files the user has selected for import

Return type list

_populate_fields(*import_candidate*)
Private method to display *ImportCandidate* data to the user.

Parameters **import_candidate** (`ImportCandidate`) – ImportCandidate to display

_remove_run()
Removes a run as an import candidate and refreshes the `QlistWidget` to reflect the removal.

_restore_defaults()
Restore suggested import settings for an *ImportCandidate* in case the user has changed them and then wants to undo those changes.

_set_cocktail_menu()
Private method that sets the cocktail `QComboBox` based on the *Menu* instance referenced by the `selected_candidate`attribute. This method is used to convey to the user which :class:`~polo.utils.io_utils.Menu has been selected for a given *ImportCandidate*.

_set_cocktail_menu_type_radiobuttons(*type_*)
Private method that sets the *Menu* type `QRadioButtons` given a *Menu* type key.

Parameters **type** (*str*) – Menu type key. If *type_* == 's' then soluble menu radioButton state is set to True. If 'type_` == 'm' then membrane radiobutton state is set to True

_set_image_spectrum(*spectrum*)
Private method that sets the image spectrum comboBox based on the *spectrum* argument. Should be used to display the inferred spectrum of an import candidate to the user when that candidate is selected.

Parameters **spectrum** (*str*) – Spectrum key

_test_candidate_paths(*file_paths*)
Private method that validates filepaths to ensure they could be imported into Polo.

Parameters **file_paths** (*list*) – List of filepaths to be imported

Returns Tuple with first item being verified paths and second being list of paths that failed verification tests.

Return type tuple

_unrar_candidate_paths(*candidate_paths*)
Private method that attempts to un-compress a collection of rar archive files.

Parameters **candidate_paths** (*list*) – List of filepaths to unrar

Returns Tuple, first being list of paths that were successfully unrared and the second being list of filepaths that could not be unrared

Return type tuple

_update_candidate_run_data()
Private method that allows the user to update an *ImportCandidate* instance's data from the widgets in the *RunImporterDialog* by updating the dictionary referenced by an *ImportCandidate* instacnes's `data` attribute with user entered values.

_update_selected_candidate()
Private method that updates currently selected *ImportCandidate* by calling *_update_candidate_run_data()* and then updating the display by calling *_populate_fields()*.

_verify_run_name()
Private method to verify a run name. If run name fails verification clears the runname `QLineEdit` widget and shows an error message to the user.

property all_run_names
All run names of all current :class:ImportCandidate instances.

Returns Set of all run names

Return type set

property selected_candidate
The currently selected *ImportCandidate* if one exists, otherwise returns None.

Returns Currently selected candidate

Return type *ImportCandidate*

property selection_dict
Returns a dictionary who's keys are `Run` attributes and values are the values of *RunImporterDialog* widgets that correspond to these attributes.

Example of the dictionary returned below.

```
{
    'cocktail_menu': Menu,
    'date': datetime,
    'run_name': str,
    'image_spectrum': str
}
```

Returns dict

Return type dict

polo.windows.run_updater_dialog module

class polo.windows.run_updater_dialog.**RunUpdaterDialog**(*run*, *run_names*, *parent=None*)

Bases: PyQt5.QtWidgets.QDialog

Small dialog for updating basic information about a run after it has been imported. Includes updating the plate ID, the cocktail menu used and the image spectrum.

Parameters

- **run** (*Run or HWIRun*) – Run to update
- **run_names** (*list or set*) – Names of already loaded runs.
- **parent** (*QWidget, optional*) – Parent widget, defaults to None

_select_run_menu()
Private method that sets the current index of the QComboBox based on the current cocktail_menu attribute of the Run instance referenced by the *run* attribute.

_set_cocktail_menu()
Private method that display cocktails in the *Menu* QComboBox based on the current menu type selection. Either displays soluble or membrane cocktail menus.

_set_run_date()
Set the date attribute of the Run referenced by the *run* attribute from the value in the QDateEdit widget.

_update_plate_id()
Private method that updates the *plate_id* attribute of the Run instance references by the *run* attribute based on the contents of the plate ID QLineEdit widget.

_update_run()
Private wrapper method that calls all other *_update* methods and then closes the dialog.

_update_run_cocktail_menu()
Private method that updates the *cocktail_menu* attribute of the *Run* instance referenced by the *run* attribute based on the current *Menu* QComboBox selection.

_update_spectrum()
Private method that update the spectrum of the *run* attribute and the images in that run based on the current selection of the spectrum QComboBox.

property current_menus
The *polo.utils.io_utils.Menu* instances that are currently being displayed to the user via the *Menu* :QComboBox widget.

Returns List of *polo.utils.io_utils.Menu* instances

Return type list

property run
The run being updated.

Returns The run being updated

Return type *Run* or *HWIRun*

polo.windows.secure_dave_dailog module

polo.windows.settings_dialog module

polo.windows.spectrum_dialog module

class polo.windows.spectrum_dialog.**SpectrumDialog**(*loaded_runs*)
Bases: PyQt5.QtWidgets.QDialog

Small dialog used to link runs together by image spectrum. This is generally done when the same plate has been imaged using different photographic technologies. Linking the runs together allows the user to switch between the images in either run easily.

Parameters **loaded_runs** (*list*) – List of runs that have been loaded into Polo

display_suggestion()
Show the link suggestion to the user by selecting suggested links.

get_selections()
Retrieve the runs that have been selected by the user or by suggestion.

Returns list of selected run names

Return type list

get_spectrum_list(*run*)
Returns the listwidget that a run should be assigned to based on the run's image type.

Parameters **run** (Run) – Run object to assign to a listWidget

Returns QListWidget to place that run into

Return type QListWidget

link_current_selection()
Link the currently selected runs together. Creates a circular linked list structure.

populate_list_widgets()
Adds items to each image spectrum type list widget based on the Run objects stored in the *loaded_runs* attribute.

show_error_message(*message=':('*)
Helper method for showing a QErrorMessage dialog to the user.

Parameters **message** – String. The message text to show to the user.

suggest_links()
Suggest runs to link together based on their imaging dates. A link suggestion will be made if the images were taken on the same day but the runs are labeled as different image types.

Returns Suggested links as list of tuples, each tuple containing two runs that are suggested for linking.

Return type list

validate_selection(*selected_runs*)

polo.windows.time_res_dialog module

class polo.windows.time_res_dialog.**TimeResDialog**(*available_runs*)
Bases: PyQt5.QtWidgets.QDialog

auto_detect_time_links()

display_runs()

get_HWI_runs()

sort_available_runs_by_date()

validate_user_selection()

Module contents

6.2 Module contents

polo.**make_default_logger**(*name*)

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p