lyse

Release 3.0.0rc2.dev65+gcb60667

labscript suite contributors

Jan 29, 2021

DOCUMENTATION

lyse is a component of the labscript suite. It is a combination API and GUI interface that leverages the API to run user provided analysis scripts of experiment shots. This documentation provides a brief outline of the use of lyse.

CHAPTER

ONE

INTRODUCTION

Lyse is a data analysis system which gets *your code* running on experimental data as it is acquired. It is fundamenally based around the ideas of experimental *shots* and analysis *routines*. A shot is one trial of an experiment, and a routine is a `Python` script, written by you, that does something with the measurement data from one or more shots.

Analysis routines can be either *single-shot* or *multi-shot*. This determines what data and functions are available to your code when it runs. A single-shot routine has access to the data from only one shot, and functions available for saving results only to the hdf5 file for that shot. A a multi-shot routine has access to the entire dataset from all the runs that are currently loaded into **lyse**, and has functions available for saving results to an hdf5 file which does not belong to any of the shots—it's a file that exists only to save the "meta results".

Actually things are far less magical than that. The only enforced difference between a single shot routine and a multi-shot routine is a single variable provided to your code when **lyse** runs it. Your code runs in a perfectly clean `Python` environment with this one exception: a variable in the global namespace called `path`, which is a path to an hdf5 file. If you have told **lyse** that your routine is a singleshot one, then this path will point to the hdf5 file for the current shot being analysed. On the other hand, if you've told **lyse** that your routine is a multishot one, then it will be the path to an h5 file that has been selected in **lyse** for saving results to.

The other differences listed above are conventions only (though **lyse**'s design is based around the assumption that you'll follow these conventions most of the time), and pertain to how you use the API that **lyse** provides, which will be different depending on what sort of analysis you're doing.

1.1 The lyse API

So grea, you've got a single filepath. What data analysis could you possibly do with that? It might seem like you have to still do the same amount of work that you would without an analysis system! Whilst that's not quite true, it's intentionally been designed that way so that you can run your code outside **lyse** with very little modification. Another motivating factor is to minimise the amount of magic black box behaviour, such that an analysis routine is actually just an ordinary `Python` script which makes use of an API designed for our purposes. **lyse** is both a program which executes your code, and an API that your code can call on.

To use the API in an analysis routine, begine your code with:

```
from lyse import *
```

The details of the API are found in the *API reference*.

1.2 lyse GUI

The **lyse** GUI uses the API to apply single and multi-shot routines to collections of shot files, added either manually by the user or automatically by BLACS after shot completion.

Here's a screenshot of **lyse**:

1. Here's where single shot routines can be added and removed, with the plus and minus buttons. They will be executed in order on each shot (more on how that works shortly). They can be reordered, or enabled/disabled with the checkboxes on the left. The checkboxes to the right, underneath the plot icons don't currently do anything, but they are intended to provide control over how plots generated by the analysis routines are displayed and updated.
2. Here is where multi-shot routines can be added or removed. The file selection button at the top allows you to select what hdf5 file multi-shot routines will get given (to which they will save their results).
3. Allows pausing of analysis. **lyse** by default will run all single-shot routines on a shot when it arrives (either via the HTTP server or having been manually added). After all the shots have been processed, only then will the multi-shot routines be executed. So if you load ten shots in quickly, the multi-shot routines won't run until they've all been processed by the single-shot routines. However most of the time there will be sufficient delay in between shots arriving that multi-shot routines will be executed pretty much every time a new shot arrives.
4. If you want to re-run single-shot analyses on some shots, select them and click this button. They'll then be processed in order.
5. This will rerun all the multi-shot analyses.
6. Here is where shots appear, either having arrived over HTTP of having been added manually via the file browser (by clicking the plus button). Many columns will populate this part of the screen, one for each global and each of the results (as saved by single-shot routines) present in the shots. A high-priority planned feature is to be able to choose exactly which globals and results are displayed. Otherwise this display is overwhelming to the point of uselessness. The data displayed here represents the entirety of what is available to multi-shot routines via the API provided by **lyse**.
7. This is where the output of routines is displayed, errors in red. If you're putting `print` statements in your analysis code, here is where to look to see them. Likewise if there's an exception and analysis stops, look here to see why.

CHAPTER

TWO

EXAMPLES

2.1 An analysis on a single shot

```
from lyse import *
from pylab import *

# Let's obtain our data for this shot -- globals, image attributes and
# the results of any previously run single-shot routines:
ser = data(path)

# Get a global called x:
x = ser['x']

# Get a result saved by another single-shot analysis routine which has
# already run. The result is called 'y', and the routine was called
# 'some_routine':
y = ser['some_routine','y']

# Image attributes are also stored in this series:
w_x2 = ser['side','absorption','OD','Gaussian_XW']

# If we want actual measurement data, we'll have to instantiate a Run object:
run = Run(path)

# Obtaining a trace:
t, mot_fluorecence = run.get_trace('mot fluorecence')

# Now we might do some analysis on this data. Say we've written a
# linear fit function (or we're calling some other libaries linear
# fit function):
m, c = linear_fit(t, mot_fluorecence)

# We might wish to plot the fit on the trace to show whether the fit is any good:

plot(t,mot_fluorecence,label='data')
plot(t,m*t + x,label='linear fit')
xlabel('time')
ylabel('MOT flourescence')
legend()

# Don't call show() ! lyse will introspect what figures have been made
# and display them once this script has finished running.  If you call
# show() it won't find anything. lyse keeps track of figures so that new
```

(continues on next page)

(continued from previous page)

```
# figures replace old ones, rather than you getting new window popping
# up every time your script runs.

# We might wish to save this result so that we can compare it across
# shots in a multishot analysis:
run.save_result('mot loadrate', c)
```

2.2 An analysis on multiple shots

```
from lyse import *
from pylab import *

# Let's obtain the dataframe for all of lyse's currently loaded shots:
df = data()

# Now let's see how the MOT load rate varies with, say a global called
# 'detuning', which might be the detuning of the MOT beams:

detunings = df['detuning']

# mot load rate was saved by a routine called calculate_load_rate:

load_rates = df['calculate_load_rate', 'mot loadrate']

# Let's plot them against each other:

plot(detunings, load_rates,'bo',label='data')

# Maybe we expect a linear relationship over the range we've got:
m, c = linear_fit(detunings, load_rates)
# (note, not a function provided by lyse, though I'm sure we'll have
# lots of stock functions like this available for import!)

plot(detunings, m*detunings + c, 'ro', label='linear fit')
legend()

#To save this result to the output hdf5 file, we have to instantiate a
#Sequence object:
seq = Sequence(path, df)
seq.save_result('detuning_loadrate_slope',c)
```

CHAPTER

THREE

API REFERENCE

3.1 Lyse Helper Functions

lyse.**data**(*filepath=None*, *host='localhost'*, *port=42519*, *timeout=5*, *n_sequences=None*, *filter_kwargs=None*)

Get data from the lyse dataframe or a file.

This function allows for either extracting information from a run's hdf5 file, or retrieving data from lyse's dataframe. If `filepath` is provided then data will be read from that file and returned as a pandas series. If `filepath` is not provided then the dataframe in lyse, or a portion of it, will be returned.

Often only part of the lyse dataframe is needed, so the `n_sequences` and `filter_kwargs` arguments provide ways to restrict what parts of the lyse dataframe are returned. The dataframe can be quite large, so only requesting a small part of it can speed up the execution of `lyse.data()` noticeably. Setting `n_sequences` makes this function return only the rows of the lyse dataframe that correspond to the `n_sequences` most recent sequences, where one sequence corresponds to one call to engage in runmanager. Additionally, the `Dataframe.filter()` method can be called on the dataframe before it is transmitted, and the arguments specified in `filter_kwargs` are passed to that method.

Parameters

- **filepath** (*str, optional*) – The path to a run's hdf5 file. If a value other than `None` is provided, then this function will return a pandas series containing data associated with the run. In particular it will contain the globals, singleshot results, multishot results, etc. that would appear in the run's row in the Lyse dataframe, but the values will be read from the file rather than extracted from the lyse dataframe. If `filepath` is `None`, then this function will instead return a section of the lyse dataframe. Note that if `filepath` is not None, then the other arguments will be ignored. Defaults to `None`.
- **host** (*str, optional*) – The address of the computer running lyse. Defaults to `'localhost'`.
- **port** (*int, optional*) – The port on which lyse is listening. Defaults to the entry for lyse's port in the labconfig, with a fallback value of 42519 if the labconfig has no such entry.
- **timeout** (*float, optional*) – The timeout, in seconds, for the communication with lyse. Defaults to 5.
- **n_sequences** (*int, optional*) – The maximum number of sequences to include in the returned dataframe where one sequence corresponds to one call to engage in runmanager. The dataframe rows for the most recent `n_sequences` sequences are returned. If the dataframe contains fewer than `n_sequences` sequences, then all rows will be returned. If set to `None`, then all rows are returned. Defaults to `None`.

- **filter_kwargs** (*dict, optional*) – A dictionary of keyword arguments to pass to the `Dataframe.filter()` method before the lyse dataframe is returned. For example to call `filter()` with `like='temperature'`, set `filter_kwargs` to `{'like':'temperature'}`. If set to `None` then `Dataframe.filter()` will not be called. See `pandas.DataFrame.filter()` for more information. Defaults to `None`.

Raises **ValueError** – If `n_sequences` isn't `None` or a nonnegative integer, then a `ValueError` is raised. Note that no `ValueError` is raised if `n_sequences` is greater than the number of sequences available. In that case as all available sequences are returned, i.e. the entire lyse dataframe is returned.

Returns If `filepath` is provided, then a pandas series with the data read from that file is returned. If `filepath` is omitted or set to `None` then the lyse dataframe, or a subset of it, is returned.

Return type `pandas.DataFrame` or `pandas.Series`

lyse.**delay_results_return**()

lyse.**figure_to_clipboard**(*figure=None*, ***kwargs*)
Copy a matplotlib figure to the clipboard as a png. If figure is None, the current figure will be copied. Copying the figure is implemented by calling figure.savefig() and then copying the image data from the resulting file. Any keyword arguments will be passed to the call to savefig(). If bbox_inches keyword arg is not provided, bbox_inches='tight' will be used

lyse.**get_plot_class**(*identifier*)

lyse.**globals_diff**(*run1*, *run2*, *group=None*)

lyse.**register_plot_class**(*identifier*, *cls*)

3.2 Run

class lyse.**Run**(*h5_path*, *no_write=False*)
Bases: `object`

A class for saving/retrieving data to/from a shot's hdf5 file.

This class implements methods that allow the user to retrieve data from a shot's hdf5 file such as images, traces, and the values of globals. It also provides methods for saving and retrieving results from analysis.

Parameters

- **h5_path** (*str*) – The path, including file name and extension, to the hdf5 file for a shot.
- **no_write** (*bool, optional*) – Set to `True` to prevent editing the shot's hdf5 file. Note that doing so prohibits the ability to save results to the file. Defaults to `False`.

__init__(*h5_path*, *no_write=False*)
Initialize self. See help(type(self)) for accurate signature.

get_all_image_labels()

get_attrs(*group*)
Returns all attributes of the specified group as a dictionary.

get_globals(*group=None*)

get_globals_expansion()

get_globals_raw(*group=None*)

get_image(*orientation*, *label*, *image*)

get_image_attributes(*orientation*)

get_images(*orientation*, *label*, **images*)

get_result(*group*, *name*)
Return 'result' in 'results/group' that was saved by the save_result() method.

get_result_array(*group*, *name*)

get_result_arrays(*group*, **names*)

get_results(*group*, **names*)
Iteratively call get_result(group,name) for each name provided. Returns a list of all results in same order as names provided.

get_trace(*name*)

get_traces(**names*)

get_units(*group=None*)
Get the units of globals.

This method retrieves the values in the "Units" column of runmanager for this shot. The values are returned in a dictionary where the keys are the names of globals and the values are the corresponding units.

Parameters **group** (*str, optional*) – The name of the globals group for which the units will be retrieved. Globals and units from other globals groups will not be included in the returned dictionary. If set to `None` then all globals from all globals groups will be returned. If `group` is set to a value that isn't the name of a globals group, then an empty dictionary will be returned, but no error will be raised. Defaults to `None`.

Returns A dictionary in which each key is a string giving the name of a global, and each value is a string specifying the corresponding value in the "Units" column of runmanager. An empty dictionary will be returned if `group` is set to a value that isn't the name of a globals group.

Return type dict

globals_diff(*other_run*, *group=None*)

globals_groups()

property group
The group in the hdf5 file in which results are saved by default.

When a `Run` instance is created from within a lyse singleshot or multishot routine, `group` will be set to the name of the running routine. If created from outside a lyse script it will be set to `None`. To change the default group for saving results, use the `set_group()` method. Note that if `self.group` is `None` and no value is provided for the optional `group` argument used by the `save...()` methods, a `ValueError` will be raised.

Attempting to directly set `self.group`'s value will automatically call `self.set_group()`.

Type str

property h5_path
The value provided for `h5_path` during instantiation.

Type str

property no_write
The value provided for `no_write` during instantiation.

Type bool

save_result(*name*, *value*, *group=None*, *overwrite=True*)
Save a result to the hdf5 file.

With the default argument values this method saves to `self.group` in the `'/results'` group and overwrites any existing value. Note that the result is saved as an attribute and overwriting attributes causes hdf5 file size bloat.

Parameters

- **name** (*str*) – The name of the result. This will be the name of the attribute added to the hdf5 file's group.
- **value** (*any*) – The value of the result, which will be saved as the value of the hdf5 group's attribute set by `name`. However note that when saving large arrays, it is better to use the `self.save_result_array()` method which will store the results as a dataset in the hdf5 file.
- **group** (*str, optional*) – The group in the hdf5 file to which the result will be saved as an attribute. If set to `None`, then the result will be saved to `self.group` in `'/results'`. Note that if a value is passed for `group` here then it will NOT have `'/result'` prepended to it which allows the caller to save results anywhere in the hdf5 file. This is in contrast to using the default group set with `self.set_group()`; when the default group is set with that method it WILL have `'/results'` prepended to it when saving results. Defaults to `None`.
- **overwrite** (*bool, optional*) – Sets whether or not to overwrite the previous value if the attribute already exists. If set to `False` and the attribute already exists, a `PermissionError` is raised. Defaults to `True`.

Raises

- **PermissionError** – A `PermissionError` is raised if `self.no_write` is `True` because saving the result would edit the file.
- **ValueError** – A `ValueError` is raised if `self.group` is `None` and no value is provided for `group` because the method then doesn't know where to save the result.
- **PermissionError** – A `PermissionError` is raised if an attribute with name `name` already exists but `overwrite` is set to `False`.

save_result_array(*name*, *data*, *group=None*, *overwrite=True*, *keep_attrs=False*, ***kwargs*)
Save an array of data to the hdf5 h5 file.

With the default argument values this method saves to `self.group` in the `'/results'` group and overwrites any existing value without keeping the dataset's previous attributes. Additional keyword arguments are passed directly to `h5py.create_dataset()`.

Parameters

- **name** (*str*) – The name of the result. This will be the name of the dataset added to the hdf5 file.
- **data** (`numpy.array`) – The data to save to the hdf5 file.
- **group** (*str, optional*) – The group in the hdf5 file in which the result will be saved as a dataset. If set to `None`, then the result will be saved in `self.group` in `'/results'`. Note that if a value is passed for `group` here then it will NOT have `'/result'` prepended to it which allows the caller to save results anywhere in the hdf5 file. This is in contrast to using the default group set with `self.set_group()`; when the default group is set with that method it WILL have `'/results'` prepended to it when saving results. Defaults to `None`..

- **overwrite** (*bool, optional*) – Sets whether or not to overwrite the previous value if the dataset already exists. If set to `False` and the dataset already exists, a `PermissionError` is raised. Defaults to `True`.
- **keep_attrs** (*bool, optional*) – Whether or not to keep the dataset's attributes when overwriting it, i.e. if the dataset already existed. Defaults to `False`.

Raises

- **PermissionError** – A `PermissionError` is raised if `self.no_write` is `True` because saving the result would edit the file.
- **ValueError** – A `ValueError` is raised if `self.group` is `None` and no value is provided for `group` because the method then doesn't know where to save the result.
- **PermissionError** – A `PermissionError` is raised if a dataset with name `name` already exists but `overwrite` is set to `False`.

save_result_arrays(**args*, ***kwargs*)

Iteratively call save_result_array() on multiple data sets. Assumes arguments are ordered such that each dataset to be saved is preceded by the name to save it as. All keyword arguments are passed to each call of save_result_array().

save_results(**args*, ***kwargs*)

Save multiple results to the hdf5 file.

This method Iteratively call `self.save_result()` on multiple results. It assumes arguments are ordered such that each result to be saved is preceded by the name of the attribute to save it under. Keywords arguments are passed to each call of `self.save_result()`.

Parameters

- ***args** – The names and values of results to be saved. The first entry should be a string giving the name of the first result, and the second entry should be the value for that result. After that, an arbitrary number of additional pairs of result name strings and values can be included, e.g. `'name0', value0, 'name1', value1,...`.
- ****kwargs** – Keyword arguments are passed to `self.save_result()`. Note that the names and values of keyword arguments are NOT saved as results to the hdf5 file; they are only used to provide values for the optional arguments of `self.save_result()`.

Examples

```
>>> run = Run('path/to/an/hdf5/file.h5')
>>> a = 5
>>> b = 2.48
>>> run.save_results('result', a, 'other_result', b, overwrite=False)
```

save_results_dict(*results_dict*, *uncertainties=False*, ***kwargs*)

set_group(*groupname*)

Set the default hdf5 file group for saving results.

The `save...()` methods will save their results to `self.group` if an explicit value for their optional `group` argument is not given. This method updates `self.group`, making sure to create the group in the hdf5 file if it does not already exist.

Parameters **groupname** (*str*) – The name of the hdf5 file group in which to save results by default. The group will be created in the `'/results'` group of the hdf5 file.

trace_names()

3.3 Sequence

class lyse.**Sequence**(*h5_path*, *run_paths*, *no_write=False*)

Bases: *lyse.Run*

__init__(*h5_path*, *run_paths*, *no_write=False*)

Initialize self. See help(type(self)) for accurate signature.

get_all_image_labels()

get_attrs(*group*)

Returns all attributes of the specified group as a dictionary.

get_globals(*group=None*)

get_globals_expansion()

get_globals_raw(*group=None*)

get_image(**args*)

get_image_attributes(*orientation*)

get_images(*orientation*, *label*, **images*)

get_result(*group*, *name*)

Return 'result' in 'results/group' that was saved by the save_result() method.

get_result_array(**args*)

get_result_arrays(**args*)

get_results(*group*, **names*)

Iteratively call get_result(group,name) for each name provided. Returns a list of all results in same order as names provided.

get_trace(**args*)

get_traces(**args*)

get_units(*group=None*)

Get the units of globals.

This method retrieves the values in the "Units" column of runmanager for this shot. The values are returned in a dictionary where the keys are the names of globals and the values are the corresponding units.

Parameters **group** (*str, optional*) – The name of the globals group for which the units will be retrieved. Globals and units from other globals groups will not be included in the returned dictionary. If set to `None` then all globals from all globals groups will be returned. If `group` is set to a value that isn't the name of a globals group, then an empty dictionary will be returned, but no error will be raised. Defaults to `None`.

Returns A dictionary in which each key is a string giving the name of a global, and each value is a string specifying the corresponding value in the "Units" column of runmanager. An empty dictionary will be returned if `group` is set to a value that isn't the name of a globals group.

Return type dict

globals_diff(*other_run*, *group=None*)

globals_groups()

property group

The group in the hdf5 file in which results are saved by default.

When a `Run` instance is created from within a lyse singleshot or multishot routine, `group` will be set to the name of the running routine. If created from outside a lyse script it will be set to `None`. To change the default group for saving results, use the `set_group()` method. Note that if `self.group` is `None` and no value is provided for the optional `group` argument used by the `save...()` methods, a `ValueError` will be raised.

Attempting to directly set `self.group`'s value will automatically call `self.set_group()`.

Type str

property h5_path
The value provided for `h5_path` during instantiation.

Type str

property no_write
The value provided for `no_write` during instantiation.

Type bool

save_result(*name*, *value*, *group=None*, *overwrite=True*)
Save a result to the hdf5 file.

With the default argument values this method saves to `self.group` in the `'/results'` group and overwrites any existing value. Note that the result is saved as an attribute and overwriting attributes causes hdf5 file size bloat.

Parameters

- **name** (*str*) – The name of the result. This will be the name of the attribute added to the hdf5 file's group.
- **value** (*any*) – The value of the result, which will be saved as the value of the hdf5 group's attribute set by `name`. However note that when saving large arrays, it is better to use the `self.save_result_array()` method which will store the results as a dataset in the hdf5 file.
- **group** (*str, optional*) – The group in the hdf5 file to which the result will be saved as an attribute. If set to `None`, then the result will be saved to `self.group` in `'/results'`. Note that if a value is passed for `group` here then it will NOT have `'/result'` prepended to it which allows the caller to save results anywhere in the hdf5 file. This is in contrast to using the default group set with `self.set_group()`; when the default group is set with that method it WILL have `'/results'` prepended to it when saving results. Defaults to `None`.
- **overwrite** (*bool, optional*) – Sets whether or not to overwrite the previous value if the attribute already exists. If set to `False` and the attribute already exists, a `PermissionError` is raised. Defaults to `True`.

Raises

- **PermissionError** – A `PermissionError` is raised if `self.no_write` is `True` because saving the result would edit the file.
- **ValueError** – A `ValueError` is raised if `self.group` is `None` and no value is provided for `group` because the method then doesn't know where to save the result.
- **PermissionError** – A `PermissionError` is raised if an attribute with name `name` already exists but `overwrite` is set to `False`.

save_result_array(*name*, *data*, *group=None*, *overwrite=True*, *keep_attrs=False*, ***kwargs*)
Save an array of data to the hdf5 h5 file.

With the default argument values this method saves to `self.group` in the `'/results'` group and overwrites any existing value without keeping the dataset's previous attributes. Additional keyword arguments are passed directly to `h5py.create_dataset()`.

Parameters

- **name** (*str*) – The name of the result. This will be the name of the dataset added to the hdf5 file.
- **data** (`numpy.array`) – The data to save to the hdf5 file.
- **group** (*str, optional*) – The group in the hdf5 file in which the result will be saved as a dataset. If set to `None`, then the result will be saved in `self.group` in `'/results'`. Note that if a value is passed for `group` here then it will NOT have `'/result'` prepended to it which allows the caller to save results anywhere in the hdf5 file. This is in contrast to using the default group set with `self.set_group()`; when the default group is set with that method it WILL have `'/results'` prepended to it when saving results. Defaults to `None`..
- **overwrite** (*bool, optional*) – Sets whether or not to overwrite the previous value if the dataset already exists. If set to `False` and the dataset already exists, a `PermissionError` is raised. Defaults to `True`.
- **keep_attrs** (*bool, optional*) – Whether or not to keep the dataset's attributes when overwriting it, i.e. if the dataset already existed. Defaults to `False`.

Raises

- **PermissionError** – A `PermissionError` is raised if `self.no_write` is `True` because saving the result would edit the file.
- **ValueError** – A `ValueError` is raised if `self.group` is `None` and no value is provided for `group` because the method then doesn't know where to save the result.
- **PermissionError** – A `PermissionError` is raised if a dataset with name `name` already exists but `overwrite` is set to `False`.

save_result_arrays(**args*, ***kwargs*)

Iteratively call save_result_array() on multiple data sets. Assumes arguments are ordered such that each dataset to be saved is preceded by the name to save it as. All keyword arguments are passed to each call of save_result_array().

save_results(**args*, ***kwargs*)

Save multiple results to the hdf5 file.

This method Iteratively call `self.save_result()` on multiple results. It assumes arguments are ordered such that each result to be saved is preceded by the name of the attribute to save it under. Keywords arguments are passed to each call of `self.save_result()`.

Parameters

- ***args** – The names and values of results to be saved. The first entry should be a string giving the name of the first result, and the second entry should be the value for that result. After that, an arbitrary number of additional pairs of result name strings and values can be included, e.g. `'name0', value0, 'name1', value1,...`.
- ****kwargs** – Keyword arguments are passed to `self.save_result()`. Note that the names and values of keyword arguments are NOT saved as results to the hdf5 file; they are only used to provide values for the optional arguments of `self.save_result()`.

Examples

```
>>> run = Run('path/to/an/hdf5/file.h5')
>>> a = 5
>>> b = 2.48
>>> run.save_results('result', a, 'other_result', b, overwrite=False)
```

save_results_dict(*results_dict*, *uncertainties=False*, ***kwargs*)

set_group(*groupname*)
Set the default hdf5 file group for saving results.

The `save...()` methods will save their results to `self.group` if an explicit value for their optional `group` argument is not given. This method updates `self.group`, making sure to create the group in the hdf5 file if it does not already exist.

Parameters **groupname** (*str*) – The name of the hdf5 file group in which to save results by default. The group will be created in the `'/results'` group of the hdf5 file.

trace_names()

3.4 Dataframe Utilities

lyse.dataframe_utilities.**asdatetime**(*timestr*)

lyse.dataframe_utilities.**concat_with_padding**(**dataframes*)
Concatenates dataframes with MultiIndex column labels, padding shallower hierarchies such that the MultiIndexes have the same nlevels.

lyse.dataframe_utilities.**flat_dict_to_flat_series**(*dictionary*)

lyse.dataframe_utilities.**flat_dict_to_hierarchical_dataframe**(*dictionary*)
Make all the keys tuples of the same length

lyse.dataframe_utilities.**flatten_dict**(*dictionary*, *keys=*)
Takes a nested dictionary whose keys are strings, and returns a flat dictionary whose keys are tuples of strings, each element of which is the key for one level of the hierarchy.

lyse.dataframe_utilities.**get_dataframe_from_shot**(*filepath*)

lyse.dataframe_utilities.**get_dataframe_from_shots**(*filepaths*)

lyse.dataframe_utilities.**get_nested_dict_from_shot**(*filepath*)

lyse.dataframe_utilities.**get_series_from_shot**(*filepath*)

lyse.dataframe_utilities.**pad_columns**(*df*, *n*)
Add depth to hiererchical column labels with empty strings

lyse.dataframe_utilities.**replace_with_padding**(*df*, *row*, *index*)

3.5 Analysis Subprocess

class lyse.analysis_subprocess.**AnalysisWorker**(*filepath*, *to_parent*, *from_parent*)
Bases: object

do_analysis(*path*)

mainloop()

new_figure(*fig*, *identifier*)

post_analysis_plot_actions()

pre_analysis_plot_actions()

reset_figs()

class lyse.analysis_subprocess.**Plot**(*figure*, *identifier*, *filepath*)
Bases: object

analysis_complete(*figure_in_use*)
To be overriden by subclasses. Called as part of the post analysis plot actions

clear()

draw()

get_window_state()
Called when the Plot window is about to be closed due to a change in registered Plot window class

Can be overridden by subclasses if custom information should be saved (although bear in mind that you will passing the information from the previous Plot subclass which might not be what you want unless the old and new classes have a common ancestor, or the change in Plot class is triggered by a reload of the module containing your Plot subclass).

Returns a dictionary of information on the window state.

If you have overridden this method, please call the base method first and then update the returned dictionary with your additional information before returning it from your method.

property is_shown

on_close()
Called when the window is closed.

Note that this only happens if the Plot window class has changed. Clicking the "X" button in the window title bar has been overridden to hide the window instead of closing it.

on_copy_to_clipboard_triggered()

on_lock_axes_triggered()

restore_axis_limits()

restore_window_state(*state*)
Called when the Plot window is recreated due to a change in registered Plot window class.

Can be overridden by subclasses if custom information should be restored (although bear in mind that you will get the information from the previous Plot subclass which might not be what you want unless the old and new classes have a common ancestor, or the change in Plot class is triggered by a reload of the module containing your Plot subclass).

If overriding, please call the parent method in addition to your new code

Parameters **state** – A dictionary of information to restore

save_axis_limits()

set_window_title(*identifier*, *filepath*)

show()

update_window_size()

class lyse.analysis_subprocess.**PlotWindow**(*plot*, **args*, ***kwargs*)
Bases: PyQt5.QtWidgets.QWidget

closeEvent(*self*, *QCloseEvent*)

close_signal

class lyse.analysis_subprocess.**PlotWindowCloseEvent**(*force*, **args*, ***kwargs*)
Bases: PyQt5.QtGui.QCloseEvent

CHAPTER

FOUR

LABSCRIPT SUITE COMPONENTS

The *labscript suite* is modular by design, and is comprised of:

Table 1: Python libraries

	labscript — Expressive composition of hardware-timed experiments
	labscript-devices — Plugin architecture for controlling experiment hardware
	labscript-utils — Shared modules used by the *labscript suite*

Table 2: Graphical applications

	runmanager — Graphical and remote interface to parameterized experiments
	blacs — Graphical interface to scientific instruments and experiment supervision
	lyse — Online analysis of live experiment data
	runviewer — Visualize hardware-timed experiment instructions

PYTHON MODULE INDEX

l

INDEX

Symbols

A

C

D

F

G