paws: the Platform for Automated Workflows by SSRL

Release 0.7.10

Lenson A. Pellouchoud

Oct 12, 2017

CONTENTS

Contents:

CHAPTER ONE

INTRODUCTION

The PAWS package aims to provide a fast and lean platform for building and executing workflows for data processing. It was originally developed to process scattering and diffraction images for research purposes at SLAC/SSRL. At the core of PAWS is a library of operations, which are essentially interfaces for other useful Python packages.

Some of the core interests of PAWS:

- Portable workflows:

  equally useful for scripting at home, for sending code to your colleagues, for performing computations behind applications, or for remote execution of large jobs on high-performance clusters.

- Scalable workflows:

  develop and test on one sample, scale up to thousands without hitting barriers.

- Flexible plugins:

  plug-and-play clients for communicating with experimental equipment, moving data to and from databases or filesystems, communicating with remote PAWS instances, or anything else that should happen outside of your workflows.

The PAWS developers would love to hear from you if you have wisdom, haikus, bugs, artwork, or suggestions. Get in touch with us at *paws-developers@slac.stanford.edu*.

CHAPTER

TWO

INSTALLATION

The full PAWS package is available on PyPI. To install it in an existing Python environment, invoke pip: `pip install pypaws`

The only dependency of PAWS core packages is pyyaml, used for serializing and de-serializing workflow data. pip will automatically install this along with PAWS.

The dependencies of the PAWS Operations are not declared as dependencies of PAWS. This keeps the Python environment relatively lean and avoids installation overhead, but it means that users will have to prepare their own environments for the Operations they want to use.

The PAWS GUI modules are not explicitly supported by the package dependencies. To use PAWS GUI modules, install PySide into your Python environment: `pip install PySide`

CHAPTER

THREE

USAGE

TODO: Some comments about structure

API Usage

TODO: code examples to build and execute a workflow

GUI Usage

TODO: instructions with screenshots

CHAPTER

FOUR

PACKAGE DOCUMENTATION

.

paws

paws package

Subpackages

paws.api package

Module contents

This module defines a class that presents an API for paws.

class `paws.api.`**`PawsAPI`**

Bases: `object`

A container to facilitate interaction with a set of paws objects: an Operations Manager, a Workflow Manager, and a Plugins Manager.

`activate_op`(*op_uri*)

Import the Operation indicated by op_uri, and tag it as active. The Operation becomes available to add to workflows via paws.api.add_op()

`add_op`(*op_tag*, *op_spec*, *wfname=None*)

`add_plugin`(*pgin_tag*, *pgin_name*)

`add_wf`(*wfname*)

Adds a workflow to the workflow manager. Input the workflow name. If no current workflow is selected, calls self.select_wf(wfname) at the end, selecting the new workflow for subsequent api calls.

`add_wf_input`(*wf_input_name*, *input_uris*, *wfname=None*)

Add an input to the workflow specified by wfname, and specify its workflow routing by any number of input_uris, which should refer to the inputs of operations in the workflow. When the workflow is asked to set this input to some value x, it will set all of the provided input_uris to x.

`add_wf_output`(*wf_output_name*, *output_uris*, *wfname=None*)

Add an output to the workflow specified by wfname, and specify one or more output_uris for pieces of workflow data that will be referenced to this workflow output. If multiple output_uris are specified, they will be packed as a list.

`current_wf`()

current_wf_name()

deactivate_op(*op_uri*)
Disable the Operation indicated by op_uri. The Operation cannot be added to Workflows until it is enabled again.

disable_op(*op_tag*, *wfname=None*)

enable_op(*op_tag*, *wfname=None*)

enable_plugin(*pgin_name=’‘*)
This tests the compatibility between the environment and the named plugin by attempting to import the plugin. If this does not throw an ImportError, then the environment satisfies the plugin dependencies.

execute(*wfname=None*)

get_input_data(*opname*, *input_name*, *wfname=None*)

get_input_setting(*opname*, *input_name*, *wfname=None*)

get_op(*opname*, *wfname=None*)

get_output(*opname*, *output_name=None*, *wfname=None*)

get_plugin(*pgin_name*)

get_wf(*wfname=None*)

info()

list_op_tags(*wfname=None*)

list_plugin_tags()

list_wf_tags()

load_from_wfl(*wfl_filename*)

load_plugin(*pgin_module*)

n_wf()

op_count(*wfname=None*)

remove_op(*op_tag*, *wfname=None*)

remove_wf_input(*wf_input_name*, *wfname=None*)

remove_wf_output(*wf_output_name*, *wfname=None*)

save_config()

save_to_wfl(*wfl_filename*)
Save the current workflows and plugins to a .wfl (YAML) file, specified by wfl_filename. If the given filename does not have the .wfl extension, it will be appended.

select_wf(*wfname*)
Sets the current workflow for the API instance. This is only to simplify subsequent api calls: anywhere there is an optional workflow name input, the default behavior is to apply the call to the current workflow.

set_input(*opname*, *input_name*, *val=None*, *tp=None*, *wfname=None*)

set_logmethod(*lm*)
Sets the logmethod, which is the function that is called to handle messages.

Parameters **lm** (*function*) – function to be called for logging messages

set_plugin_input(*pgin_tag*, *input_name*, *val=None*, *tp=None*)

set_wf_input(*wf_input_name*, *val=None*, *wfname=None*)

start_plugin(*pgin_name*)

wfl_dict()

paws.api.**start**()

Instantiate and return a PawsAPI object.

paws.api.start() calls the PawsAPI constructor.

Returns a PawsAPI object

Return type paws.api.PawsAPI

paws.core package

Subpackages

paws.core.models package

Submodules

paws.core.models.DictTree module

class paws.core.models.DictTree.**DictTree**(*data={}*)

Bases: object

A tree as an ordered dictionary (root), extended by embedding other objects that are amenable to tree storage. Fetches items by a uri string that is a sequence of dict keys, connected by '.'s.

Child items (end nodes of the tree) can be anything. Parent items, in order to index their children, must be either lists, dicts, or objects implementing keys(), __getitem__(key) and __setitem__(key,value).

contains_uri(*uri*)

Returns whether or not input uri points to an item in this tree.

delete_uri(*uri='‘*)

Delete the given uri, i.e., remove the corresponding key from the embedded dict. This should not be relied on to be fast. It has to go through all of the uris to remove children.

get_from_uri(*uri='‘*)

Return the data stored at uri. Each data item in the lineage of the uri must implement __getitem__() with support for string-like keys, unless it is a list, in which case the key is cast as int(key) before using it as an index in the list.

is_tag_valid(*tag*)

Check for validity of a tag. The conditions for a valid tag are the same as for a valid uri, except that a tag should not contain period (.) characters.

is_uri_unique(*uri*)

Check for uniqueness of a uri.

is_uri_valid(*uri*)

Check for validity of a uri. Uris may contain upper case letters, lower case letters, numbers, dashes (-), and underscores (_). Periods (.) are used as delimiters between tags in the uri. Any whitespace or any character in the string.punctuation library (other than -, _, or .) results in an invalid uri.

make_unique_uri(*prefix*)
Generate the next unique uri from prefix by appending '_x' to it, where x is a minimal nonnegative integer.

print_tree(*root_uri='', rowprefix=''*)
Print the content of the tree rooted at root_uri, with each row of the string preceded by rowprefix.

root_keys()

set_uri(*uri='', val=None*)
Set the data at the given uri to provided value val.

tag_error_message(*tag*)
Provide a human-readable error message for bad tags.

uri_error_message(*uri*)
Provide a human-readable error message for bad uris.

paws.core.models.ListModel module

class paws.core.models.ListModel.**ListModel**(*input_list=[], parent=None*)
Bases: `PySide.QtCore.QAbstractListModel`

Class for list management with a QAbstractListModel. Implements required virtual methods rowCount() and data(). Resizeable ListModels must implement insertRows(), removeRows(). If a nicely labeled header is desired, implement headerData().

append_item(*thing*)

columnCount(*parent=<PySide.QtCore.QModelIndex(-1, -1, 0x0, QObject(0x0)) >*)

data(*idx, data_role*)

flags(*idx*)

get_item(*idx*)

headerData(*section, orientation, data_role*)

insertRows(*row, count*)

list_data()

n_items()

removeRows(*row, count, parent=<PySide.QtCore.QModelIndex(-1, -1, 0x0, QObject(0x0)) >*)

remove_item(*row*)

rowCount(*parent=<PySide.QtCore.QModelIndex(-1, -1, 0x0, QObject(0x0)) >*)

set_disabled(*row*)

set_enabled(*row*)

staticMetaObject = <PySide.QtCore.QMetaObject object>

class paws.core.models.ListModel.**PluginListModel**(*input_list=[], parent=None*)
Bases: `paws.core.models.ListModel.ListModel`

Just a ListModel with overloaded headerData

headerData(*section, orientation, data_role*)

staticMetaObject = <PySide.QtCore.QMetaObject object>

paws.core.models.TreeItem module

class `paws.core.models.TreeItem.`**`TreeItem`**(*parent_itm*, *tag*)

Bases: `object`

A structured container for indexing a TreeModel. A TreeItem keeps references to a parent TreeItem and a list of child TreeItems. It is labeled by a tag (TreeItem.tag) which must be unique across its sibling TreeItems. A root TreeItem should have None as its parent item.

`build_uri`()

Return the TreeModel uri of this TreeItem by following its parents up to a root item.

`n_children`()

paws.core.models.TreeModel module

class `paws.core.models.TreeModel.`**`TreeModel`**(*default_flags={}*)

Bases: `object`

This class indexes a DictTree with a set of TreeItems. TreeItems keep track of their lineage in the DictTree, and can be modified for additional functionality in subclasses of TreeModel by adding TreeItem.flags.

`build_tree`(*x*)

TreeModel.build_tree is called on some object x before x is stored in the tree. For subclasses of TreeModel to build tree data for data types other than dicts and lists, build_tree should be reimplemented. If data types other than dicts and lists have child items that should be accessible by TreeModel uris, they should implement __getitem__(tag).

`build_uri`(*itm*)

Build a URI for TreeItem itm by combining the tags of the lineage of itm, with '.' as a delimiter.

`contains_uri`(*uri*)

`create_tree_item`(*parent_itm*, *itm_tag*)

Build a TreeItem for use in this tree. Reimplement create_tree_item() in subclasses of TreeModel to add features to TreeItems, such as default values for TreeItem.flags. TreeModel implementation returns TreeItem(parent_itm,itm_tag).

`get_data_from_uri`(*uri*)

`get_from_uri`(*uri*)

`is_tag_valid`(*tag*)

`is_uri_valid`(*uri*)

`make_unique_uri`(*prefix*)

`n_children`(*parent_uri='‘*)

`remove_item`(*itm_uri*)

`root_tags`()

`set_item`(*itm_uri*, *itm_data=None*)

`tag_error_message`(*tag*)

`tree_update`(*parent_itm*, *itm_tag*, *itm_data*)

Update the tree structure rooted at parent_itm.children[itm_tag], such that TreeItems get built to index all of the items in itm_data that are supported by self.build_tree(). Assume build_tree was called on itm_data before passing it as an argument, so only need to recurse if itm_data is a dict.

Module contents

paws.core.operations package

Subpackages

paws.core.operations.EXECUTION package

Subpackages

paws.core.operations.EXECUTION.BATCH package

Submodules

paws.core.operations.EXECUTION.BATCH.BatchFromDirectory module

class `paws.core.operations.EXECUTION.BATCH.BatchFromDirectory.`**`BatchFromDirectory`**

Bases: *`paws.core.operations.Operation.Operation`*

Read a directory and filter its contents with a regular expression to form a list of file paths to be used as inputs for the repeated execution of a specified Workflow. Specify, by workflow uri, where this file path will be fed to the workflow. Collect outputs from the Workflow for each of the input files.

`run``()`

paws.core.operations.EXECUTION.BATCH.BatchFromFiles module

class `paws.core.operations.EXECUTION.BATCH.BatchFromFiles.`**`BatchFromFiles`**

Bases: *`paws.core.operations.Operation.Operation`*

Take a list of file paths and use them as inputs for the repeated execution of a specified Workflow. Specify, by workflow uri, where this file path will be fed to the workflow. Collect outputs from the Workflow for each of the input files.

`run``()`

paws.core.operations.EXECUTION.BATCH.BatchPostProcess module

class `paws.core.operations.EXECUTION.BATCH.BatchPostProcess.`**`BatchPostProcess`**

Bases: *`paws.core.operations.Operation.Operation`*

Take the batch output (list of dicts) from a previously completed Batch, and use each dict to form inputs for the execution of a post-processing workflow. For each item to be taken from the previous batch, two keys are needed: one key indicates a previous batch workflow output, and another indicates the corresponding current workflow input.

`run``()`

Build a list of [uri:value] dicts to be used in the workflow.

Module contents

paws.core.operations.EXECUTION.REALTIME package

Submodules

paws.core.operations.EXECUTION.REALTIME.RealtimeFromFiles module

class `paws.core.operations.EXECUTION.REALTIME.RealtimeFromFiles.`**`RealtimeFromFiles`**

Bases: *`paws.core.operations.Operation.Operation`*

Use file paths matching a regex to generate inputs for repeated execution of a workflow, as the files arrive in a specified directory. Collects the outputs produced for each of the inputs.

`run``()`

This should create an iterator whose next() gives a {uri:value} dict built from the latest-arrived file

Module contents

Module contents

paws.core.operations.IO package

Subpackages

paws.core.operations.IO.BL15 package

Submodules

paws.core.operations.IO.BL15.ReadHeader_SSRL15 module

class `paws.core.operations.IO.BL15.ReadHeader_SSRL15.`**`ReadHeader_SSRL15`**

Bases: *`paws.core.operations.Operation.Operation`*

Read a .txt header from beamline 1-5 at SSRL into a dict.

`run``()`

paws.core.operations.IO.BL15.ReadImageAndHeader_SSRL15 module

class `paws.core.operations.IO.BL15.ReadImageAndHeader_SSRL15.`**`ReadImageAndHeader_SSRL15`**

Bases: *`paws.core.operations.Operation.Operation`*

Read an image and header generated by beamline 1-5 at SSRL. Returns ndarray image and dictionary header.

`run``()`

Module contents

paws.core.operations.IO.CALIBRATION package

Submodules

paws.core.operations.IO.CALIBRATION.NikaToPONI module

class `paws.core.operations.IO.CALIBRATION.NikaToPONI.`**`NikaToPONI`**

Bases: *`paws.core.operations.Operation.Operation`*

Converts Nika calibration output (saved in a text file) to a dict of PyFAI PONI parameters, by first converting from Nika to Fit2D, then using a pyFAI.AzimuthalIntegrator to convert from Fit2D to PONI format.

WARNING: the map from Nika's horizontal and vertical tilts to Fit2D's tilt and tiltPlanRotation has not yet been verified by the developers. Use this operation with nonzero tilts at your own risk.

Input a text file expressing results of Nika automated calibration, and manually input polarization factor. Output a dict of pyFAI PONI calibration parameters. Format of text file for Nika output is expected to be: sample_to_CCD_mm=____ pixel_size_x_mm=____ pixel_size_y_mm=____ beam_center_x_pix=____ beam_center_y_pix=____ horizontal_tilt_deg=____ vertical_tilt_deg=____ wavelength_A=____

`run``()`

paws.core.operations.IO.CALIBRATION.ReadPONI module

class `paws.core.operations.IO.CALIBRATION.ReadPONI.`**`ReadPONI`**

Bases: *`paws.core.operations.Operation.Operation`*

Read in a dict of PyFAI PONI parameters. Input path to a .poni file representing a calibrated measurement geometry.

`run``()`

paws.core.operations.IO.CALIBRATION.WXDToPONI module

class `paws.core.operations.IO.CALIBRATION.WXDToPONI.`**`WXDToPONI`**

Bases: *`paws.core.operations.Operation.Operation`*

Convert WXDIFF .calib output to a dict of PyFAI PONI parameters, by first converting from WXDIFF to Fit2D, then using a pyFAI.AzimuthalIntegrator to convert from Fit2D to PONI format.

The conversion from WXDiff parameters to Fit2D parameters was originally contributed to paws by Fang Ren.

Input .calib file from WXDIFF automated calibration, input pixel size and polarization factor, output dict of pyFAI PONI calibration parameters.

`run``()`

Module contents

The INPUT.CALIBRATION category has operations for reading in calibration parameters and converting them between different formats. Some of the common formats are described here. Over time, these descriptions should improve. Contact the paws developers to contribute information or report inconsistencies.

PONI (PyFAI) FORMAT

PONI: point of normal incidence. This is the format used internally by the PyFAI (Python Fast Azimuthal Integration) package. PONI format projects the point-shaped sample orthogonally onto projector plane, and gives the coordinates of that projection as the PONI, such that the sample to PONI distance is the shortest distance from sample to detector plane. coordinate axes: x1 vertical, x2 and x3 horizontal, x3 along beam. detector axes: with zero rotations, d1 vertical, d2 horizontal, d3 along beam. axes defined on C format, first dimension is vertical, second dimension is horizontal. the first dimension (vertical) is fast, the second dimension (horizontal) is slow.

PONI dict keys and definitions: - 'dist': distance in meters from sample to PONI on detector plane - 'poni1': vertical coordinate of PONI on detector axes, in meters - 'poni2': horizontal coordinate of PONI on detector axes, in meters - 'rot1': rotation of detector body about x1, applied first, radians - 'rot2': rotation of detector body about x2, applied second, radians - 'rot3': rotation of detector body about beam axis x3, applied third, radians - 'pixel1': pixel dimension along d1 (vertical), meters - 'pixel2': pixel dimension along d2 (horizontal), meters - 'wavelength': wavelength in meters - 'fpolz': polarization factor- not actually a PONI parameter, but it's ok to put it here - 'detector': optional pyFAI detector object - 'splineFile' optional spline file describing detector distortion

NIKA FORMAT

The calibration performed by the Nika software package uses a calibrant image, the rectangular pixel dimensions (in mm), and the wavelength (in Angstrom), to solve the sample to CCD distance in mm, the position at which the beam axis intersects the detector plane in pixels, and the horizontal and vertical tilts of the detector in degrees.

Nika does not generate a file to save calibration parameters, so they have to be recorded by hand in a file. Paws Operations should be written to read them from a file in the following format (one parameter=value per line, no spaces): - sample_to_CCD_mm=____ - pixel_size_x_mm=____ - pixel_size_y_mm=____ - beam_center_x_pix=____ - beam_center_y_pix=____ - horizontal_tilt_deg=____ - vertical_tilt_deg=____ - wavelength_A=____

FIT2D FORMAT

Detector plane origin is the bottom left corner of the detector.

Fit2D dict keys and definitions: - 'directDist': direct distance to detector plane along beam axis, in mm - 'centerX': horizontal position on the detector plane where the beam intersects, in px - 'centerY': vertical position on the detector plan where the beam intersects, in px - 'pixelX': horizontal size of pixel, in um - 'pixelY': vertical size of pixel, in um - 'tilt': detector tilt in degrees (TODO:clarify) - 'tiltPlanRotation': detector rotation in degrees = 360 minus WXDIFF alpha (TODO:clarify) - 'splineFile' optional spline file describing detector distortion

WXDIFF FORMAT

Similar to Fit2D format, but knowledge about WXDIFF is hard to come by. I hope it can be cleanly documented here over time. Detector plane origin is the bottom left corner of the detector.

.calib file lines (and notes): - imagetype=uncorrected-q TODO: describe - dtype=uint16 img data type = unsigned 16-bit integers - horsize=___ horizontal extent of image, in pixels - versize=___ vertical extent of image, in pixels - region_ulc_x=___ TODO: describe - region_ulc_y=___ TODO: describe - bcenter_x=___ horizontal coordinate where the beam axis intersects the detector plane - bcenter_y=___ vertical coordinate where the beam axis intersects the detector plane - detect_dist=___ direct distance from the sample to the detector plane intersection, along the beam axis, in pixels - detect_tilt_alpha=___ rotation of detector tilt axis plane in radians = 360 minus Fit2D tiltPlanRotation - detect_tilt_delta=___ detector tilt in radians (TODO:clarify) - wavelenght=___ the typo 'wavelenght' is built into wxdiff, and it is reported in angstroms - Qconv_const=___ TODO: describe

paws.core.operations.IO.CSV package

Submodules

paws.core.operations.IO.CSV.CSVToArray module

class `paws.core.operations.IO.CSV.CSVToArray.`**`CSVToArray`**

Bases: *`paws.core.operations.Operation.Operation`*

Read a csv-formatted file into a numpy array.

`run``()`

paws.core.operations.IO.CSV.CSVToXYData module

class `paws.core.operations.IO.CSV.CSVToXYData.`**`CSVToXYData`**

Bases: *`paws.core.operations.Operation.Operation`*

Read a csv-formatted file as floats, package into arrays of x values and y values.

`run``()`

paws.core.operations.IO.CSV.WriteArrayCSV module

class `paws.core.operations.IO.CSV.WriteArrayCSV.`**`WriteArrayCSV`**

Bases: *`paws.core.operations.Operation.Operation`*

Write a 2d array to a csv file

`run``()`

Module contents

paws.core.operations.IO.IMAGE package

Submodules

paws.core.operations.IO.IMAGE.FabIOOpen module

class `paws.core.operations.IO.IMAGE.FabIOOpen.`**`FabIOOpen`**

Bases: *`paws.core.operations.Operation.Operation`*

Takes a filesystem path and calls fabIO to load it.

`run``()`

Call on fabIO to extract image data

paws.core.operations.IO.IMAGE.FabIOWrite module

class `paws.core.operations.IO.IMAGE.FabIOWrite.`**`FabIOWrite`**

Bases: *`paws.core.operations.Operation.Operation`*

Use FabIO to write out an image, given image data, directory path, filename, file tag, extension, an image header (dict), and a flag for whether or not to overwrite.

Outputs the full file path where the image was written, which should be dir_path+filename+filetag+ext.

run()

Call on fabIO to extract image data

paws.core.operations.IO.IMAGE.LoadTif module

class `paws.core.operations.IO.IMAGE.LoadTif.`**`LoadTif`**

Bases: *`paws.core.operations.Operation.Operation`*

Takes a filesystem path that points to a .tif, outputs image data from the file.

run()

paws.core.operations.IO.IMAGE.LoadTif_PIL module

class `paws.core.operations.IO.IMAGE.LoadTif_PIL.`**`LoadTif_PIL`**

Bases: *`paws.core.operations.Operation.Operation`*

Takes a filesystem path that points to a .tif, outputs image data and metadata from the file.

run()

Module contents

paws.core.operations.IO.MISC package

Submodules

paws.core.operations.IO.MISC.ReadNPSynthRecipe module

class `paws.core.operations.IO.MISC.ReadNPSynthRecipe.`**`ReadNPSynthRecipe`**

Bases: *`paws.core.operations.Operation.Operation`*

Read in a text file describing nanoparticle synthesis parameters. Package the recipe description in a dict.

run()

Module contents

paws.core.operations.IO.MODELS package

Subpackages

paws.core.operations.IO.MODELS.SAXS package

Submodules

paws.core.operations.IO.MODELS.SAXS.LoadSAXSClassifier module

class `paws.core.operations.IO.MODELS.SAXS.LoadSAXSClassifier.`**`LoadSAXSClassifier`**

Bases: *`paws.core.operations.Operation.Operation`*

Read files to load a set of classifiers to be used on 1-d saxs spectra.

`run``()`

Module contents

Module contents

paws.core.operations.IO.PIF package

Submodules

paws.core.operations.IO.PIF.CheckDataSet module

class `paws.core.operations.IO.PIF.CheckDataSet.`**`CheckDataSet`**

Bases: *`paws.core.operations.Operation.Operation`*

Take a Citrination client as input and use it to query a data set. Output some indication of whether or not the query was successful.

`run``()`

paws.core.operations.IO.PIF.SavePIFAsJSON module

class `paws.core.operations.IO.PIF.SavePIFAsJSON.`**`SavePIFAsJSON`**

Bases: *`paws.core.operations.Operation.Operation`*

Take a pypif.obj.System object and save it on the local filesystem in .json format

`run``()`

paws.core.operations.IO.PIF.ShipJSON module

class `paws.core.operations.IO.PIF.ShipJSON.`**`ShipJSON`**

Bases: *`paws.core.operations.Operation.Operation`*

Take a .json file containing a pif or array of pifs, ship it to a Citrination data set.

`run``()`

paws.core.operations.IO.PIF.ShipToDataSet module

class `paws.core.operations.IO.PIF.ShipToDataSet.`**`ShipToDataSet`**

Bases: *`paws.core.operations.Operation.Operation`*

Take a pypif.obj.System object and ship it to a given Citrination data set.

`run``()`

Module contents

paws.core.operations.IO.YAML package

Submodules

paws.core.operations.IO.YAML.LoadYAML module

class `paws.core.operations.IO.YAML.LoadYAML.`**`LoadYAML`**

Bases: *`paws.core.operations.Operation.Operation`*

Load a YAML file, save the output of yaml.load(open(file_path,'r'))

`run``()`

Module contents

Submodules

paws.core.operations.IO.BuildFilePath module

class `paws.core.operations.IO.BuildFilePath.`**`BuildFilePath`**

Bases: *`paws.core.operations.Operation.Operation`*

This operation helps to build file paths from workflow data. It takes a directory (full path), a filename, and an extension. The filename can optionally have a prefix or suffix inserted, to help with iteration of batches of files with similar names.

`run``()`

paws.core.operations.IO.ReadPONI module

class `paws.core.operations.IO.ReadPONI.`**`ReadPONI`**

Bases: *`paws.core.operations.Operation.Operation`*

Reads in a .poni file as output by pyFAI.geometry.Geometry.save(), outputs a poni dict as produced by pyFAI.geometry.Geometry.getPyFAI().

`run``()`

Module contents

paws.core.operations.PACKAGING package

Subpackages

paws.core.operations.PACKAGING.BATCH package

Submodules

paws.core.operations.PACKAGING.BATCH.BuildListFromBatch module

class `paws.core.operations.PACKAGING.BATCH.BuildListFromBatch.`**`BuildListFromBatch`**

Bases: *`paws.core.operations.Operation.Operation`*

Given a batch output and a batch output uri, harvest a list of outputs from the batch.

`run`()

paws.core.operations.PACKAGING.BATCH.XYDataFromBatch module

class `paws.core.operations.PACKAGING.BATCH.XYDataFromBatch.`**`XYDataFromBatch`**

Bases: *`paws.core.operations.Operation.Operation`*

Harvest two arrays from a batch output (a list of dicts). Takes a batch output, a key for x values, and a key for y values.

`run`()

Module contents

paws.core.operations.PACKAGING.BL15 package

Submodules

paws.core.operations.PACKAGING.BL15.TimeTempFromHeader module

class `paws.core.operations.PACKAGING.BL15.TimeTempFromHeader.`**`TimeTempFromHeader`**

Bases: *`paws.core.operations.Operation.Operation`*

Get time and temperature from a detector output header file. Return string time, float time (utc in seconds), and float temperature. Time is assumed to be in the format Day Mon dd hh:mm:ss yyyy.

`run`()

Module contents

paws.core.operations.PACKAGING.PIF package

Submodules

paws.core.operations.PACKAGING.PIF.EmptyPif module

class `paws.core.operations.PACKAGING.PIF.EmptyPif.`**`EmptyPif`**

Bases: *`paws.core.operations.Operation.Operation`*

Make and empty pypif.obj.ChemicalSystem object.

`run`()

`saxs_to_pif_properties`(*q_I*, *T_C*)

paws.core.operations.PACKAGING.PIF.PifNPSolutionSAXS module

class `paws.core.operations.PACKAGING.PIF.PifNPSolutionSAXS.`**`PifNPSolutionSAXS`**

Bases: *`paws.core.operations.Operation.Operation`*

Package SAXS results from a nanoparticle solution into a pypif.obj.ChemicalSystem record.

`feature_property`(*fval*, *fname*, *funits=''*)

`q_I_property`(*q_I*)

`run`()

paws.core.operations.PACKAGING.PIF.PifNPSynthExperiment module

class `paws.core.operations.PACKAGING.PIF.PifNPSynthExperiment.`**`PifNPSynthExperiment`**

Bases: *`paws.core.operations.Operation.Operation`*

Analyze a series of PIFs generated in a nanoparticle synthesis experiment and produce a master PIF that describes the overall experiment.

`run`()

`time_feature_property`(*t_f*, *fname*, *funits=''*)

Module contents

Submodules

paws.core.operations.PACKAGING.LogLogZip module

class `paws.core.operations.PACKAGING.LogLogZip.`**`LogLogZip`**

Bases: *`paws.core.operations.Operation.Operation`*

Take the base-10 logarithm of two 1d arrays, then zip them together. Any elements with non-positive or nan values are removed.

`run`()

paws.core.operations.PACKAGING.WindowZip module

class `paws.core.operations.PACKAGING.WindowZip.`**`WindowZip`**

Bases: *`paws.core.operations.Operation.Operation`*

From input sequences for x and y, produce an n-by-2 array where x is bounded by the specified limits

`run`()

`xy_zip`(*x*, *y*)

paws.core.operations.PACKAGING.Zip module

class `paws.core.operations.PACKAGING.Zip.`**`Zip`**

Bases: *`paws.core.operations.Operation.Operation`*

Zip two 1d arrays together.

run()

Module contents

paws.core.operations.PROCESSING package

Subpackages

paws.core.operations.PROCESSING.BACKGROUND package

Submodules

paws.core.operations.PROCESSING.BACKGROUND.BgSubtractByTemperature module

class `paws.core.operations.PROCESSING.BACKGROUND.BgSubtractByTemperature.`**`BgSubtractByTemperat`**

Bases: *`paws.core.operations.Operation.Operation`*

Originally contributed by Amanda Fournier.

Find a background spectrum from a batch of background spectra, where the temperature of the background spectrum is as close as possible to the (input) temperature of the measured spectrum. Then subtract that background spectrum from the input spectrum. The measured and background spectra are expected to have the same domain.

run()

paws.core.operations.PROCESSING.BACKGROUND.SubtractMaximumBackground module

class `paws.core.operations.PROCESSING.BACKGROUND.SubtractMaximumBackground.`**`SubtractMaximumBac`**

Bases: *`paws.core.operations.Operation.Operation`*

Subtract a background from a foreground, with scaling to prevent over-subtraction. Optionally, input an intensity error array, and get an error estimate for the background-subtracted intensity.

Operation originally contributed by Amanda Fournier.

run()

Module contents

paws.core.operations.PROCESSING.BASIC package

Submodules

paws.core.operations.PROCESSING.BASIC.ArrayLog module

class `paws.core.operations.PROCESSING.BASIC.ArrayLog.`**`ArrayLog`**

Bases: *`paws.core.operations.Operation.Operation`*

Take the base-10 logarithm of any array. Any elements with non-positive values are removed.

run()

paws.core.operations.PROCESSING.BASIC.ArrayMirrorHorizontal module

class `paws.core.operations.PROCESSING.BASIC.ArrayMirrorHorizontal.`**`ArrayMirrorHorizontal`**

Bases: *`paws.core.operations.Operation.Operation`*

Mirror an array across a horizontal plane, i.e., exchange indices along axis 0.

`run``()`

paws.core.operations.PROCESSING.BASIC.ArrayMirrorVertical module

class `paws.core.operations.PROCESSING.BASIC.ArrayMirrorVertical.`**`ArrayMirrorVertical`**

Bases: *`paws.core.operations.Operation.Operation`*

Mirror an array across a vertical plane, i.e., exchange indices along axis 1.

`run``()`

paws.core.operations.PROCESSING.BASIC.LogY module

class `paws.core.operations.PROCESSING.BASIC.LogY.`**`LogY`**

Bases: *`paws.core.operations.Operation.Operation`*

Take the base-10 logarithm of the second column of a n-by-2 array.

`run``()`

paws.core.operations.PROCESSING.BASIC.Rotation module

class `paws.core.operations.PROCESSING.BASIC.Rotation.`**`Rotation`**

Bases: *`paws.core.operations.Operation.Operation`*

Rotate an array by 90, 180, or 270 degrees.

`run``()`

Rotate self.inputs[’image_data’] and save as self.outputs[’image_data’]

Module contents

paws.core.operations.PROCESSING.FEATURE_EXTRACTION package

Submodules

paws.core.operations.PROCESSING.FEATURE_EXTRACTION.TextureFeatures module

class `paws.core.operations.PROCESSING.FEATURE_EXTRACTION.TextureFeatures.`**`TextureFeatures`**

Bases: *`paws.core.operations.Operation.Operation`*

Analyzes the texture of an integrated diffractogram (q, chi, and I(q,chi)).

Created on Mon Jun 06 2016.

Originally contributed by Fang Ren. Citation: Fang Ren, et al. ACS Comb. Sci., 2017, 19(6), pp 377-385.

`run``()`

Module contents

paws.core.operations.PROCESSING.INTEGRATION package

Submodules

paws.core.operations.PROCESSING.INTEGRATION.ApplyIntegrator1d module

Integrate an image, using an existing PyFAI.AzimuthalIntegrator, with a bunch of input parameters for calling AzimuthalIntegrator.integrate1d().

class `paws.core.operations.PROCESSING.INTEGRATION.ApplyIntegrator1d.`**`ApplyIntegrator1d`**

Bases: *`paws.core.operations.Operation.Operation`*

Input image data (ndarray), PyFAI.AzimuthalIntegrator, mask, ROI mask, dark field image, flat field image, q-range, chi-range, number of points for integration bin centers, polz factor, choice of unit (string), and choice of integration method (string).

Refer to the PyFAI documentation at ..... for parameter definitions and defaults. TODO: fill in web uri above.

Output arrays containing q and I(q)

`run``()`

paws.core.operations.PROCESSING.INTEGRATION.ApplyIntegrator2d module

Integrate an image to 2d, using an existing PyFAI.AzimuthalIntegrator, with a bunch of input parameters for calling AzimuthalIntegrator.integrate1d().

class `paws.core.operations.PROCESSING.INTEGRATION.ApplyIntegrator2d.`**`ApplyIntegrator2d`**

Bases: *`paws.core.operations.Operation.Operation`*

Input image data (ndarray), PyFAI.AzimuthalIntegrator, mask, ROI mask, dark field image, flat field image, q-range, chi-range, number of points for integration bin centers, polz factor, choice of unit (string), and choice of integration method (string).

Refer to the PyFAI documentation at ..... for parameter definitions and defaults. TODO: fill in web uri above.

Output arrays containing q, chi, and I(q,chi)

`run``()`

paws.core.operations.PROCESSING.INTEGRATION.BuildPyFAIIntegrator module

Produce a PyFAI.AzumthalIntegrator to use for calibrating and integrating images.

class `paws.core.operations.PROCESSING.INTEGRATION.BuildPyFAIIntegrator.`**`BuildPyFAIIntegrator`**

Bases: *`paws.core.operations.Operation.Operation`*

Input dict of calibration parameters Return AzimuthalIntegrator

`run``()`

paws.core.operations.PROCESSING.INTEGRATION.Integrate1d module

class `paws.core.operations.PROCESSING.INTEGRATION.Integrate1d.`**`Integrate1d`**

Bases: *`paws.core.operations.Operation.Operation`*

Integrate an image, given calibration parameters.

Input image data (ndarray) and a dict of .poni format calibration parameters Output q, I(q)

`run``()`

paws.core.operations.PROCESSING.INTEGRATION.Integrate2d module

Integrate an image, given calibration parameters.

This module builds a PyFAI.AzimuthalIntegrator to integrate an input image to I(q,chi).

class `paws.core.operations.PROCESSING.INTEGRATION.Integrate2d.`**`Integrate2d`**

Bases: *`paws.core.operations.Operation.Operation`*

Input image data (ndarray) and a dict of calibration parameters Return q, chi, I(q,chi)

`run``()`

paws.core.operations.PROCESSING.INTEGRATION.Remesh module

Calibrate and reduce an image, given calibration parameters.

This module calls on pipeline.remesh.remesh to correct the (GI) images for curvature of the Ewald's sphere.

class `paws.core.operations.PROCESSING.INTEGRATION.Remesh.`**`Remesh`**

Bases: *`paws.core.operations.Operation.Operation`*

Input image data (ndarray), pyFAI Geometory object, Angle of Incidence Return q_par, q_vrt, I(q_par, q_vrt)

`run``()`

paws.core.operations.PROCESSING.INTEGRATION.RemeshXIntegration module

Integrate an ROI on image in X-direction

class `paws.core.operations.PROCESSING.INTEGRATION.RemeshXIntegration.`**`RemeshXIntegration`**

Bases: *`paws.core.operations.Operation.Operation`*

Input image data (ndarray), mask, ROI mask, qvrt, qpar Output arrays containing q and I(q)

`run``()`

paws.core.operations.PROCESSING.INTEGRATION.RemeshZIntegration module

Integrate an ROI on image in Z-direction

class `paws.core.operations.PROCESSING.INTEGRATION.RemeshZIntegration.`**`RemeshZIntegration`**

Bases: *`paws.core.operations.Operation.Operation`*

Input image data (ndarray), mask, ROI mask, qvrt, qpar Output lists containing q and I(q)

`run``()`

Module contents

paws.core.operations.PROCESSING.PEAKS package

Submodules

paws.core.operations.PROCESSING.PEAKS.FindPeaksByWindow module

class `paws.core.operations.PROCESSING.PEAKS.FindPeaksByWindow.`**`FindPeaksByWindow`**

Bases: *`paws.core.operations.Operation.Operation`*

Walk a 1d array and find its local maxima. A maximum is found if it is the highest point within windowsize of itself. An optional threshold for the peak intensity relative to the window-average can be used to filter out peaks due to noise.

`run`()

paws.core.operations.PROCESSING.PEAKS.VoigtPeakFit module

class `paws.core.operations.PROCESSING.PEAKS.VoigtPeakFit.`**`VoigtPeakFit`**

Bases: *`paws.core.operations.Operation.Operation`*

Fit a set of x and y values to a Voigt distribution. Solves the best-fitting hwhm (half width at half max) of the gaussian and lorentzian distributions and shared distribution center. Takes as input a guess for the distribution center and hwhm. Range of fit is determined by weighting the objective by a Hann window centered at the distribution center, with a window width of the distribution's estimated full width at half max.

static **`gaussian`**(*x*, *hwhm_g*)

gaussian distribution at points x, center 0, half width at half max hwhm_g

static **`hann_voigt_fit`**(*x*, *y*, *xc*, *hwhm_g*, *hwhm_l*, *scl*)

static **`lorentzian`**(*x*, *hwhm_l*)

lorentzian distribution at points x, center 0, half width at half max hwhm_l

`run`()

static **`solve_voigt`**(*x*, *y*, *xc*, *hwhm_g*, *hwhm_l*, *scl*)

iteratively minimize an objective to fit x, y curve to a voigt profile

static **`voigt`**(*x*, *hwhm_g*, *hwhm_l*)

voigt distribution resulting from convolution of a gaussian with hwhm hwhm_g and a lorentzian with hwhm hwhm_l

Module contents

paws.core.operations.PROCESSING.SAXS package

Submodules

paws.core.operations.PROCESSING.SAXS.SpectrumClassifier module

class `paws.core.operations.PROCESSING.SAXS.SpectrumClassifier.`**`SpectrumClassifier`**

Bases: *`paws.core.operations.Operation.Operation`*

Identifies scatterer populations from features of SAXS spectra.

run()

paws.core.operations.PROCESSING.SAXS.SpectrumFit module

class paws.core.operations.PROCESSING.SAXS.SpectrumFit.**SpectrumFit**

Bases: *paws.core.operations.Operation.Operation*

Use a measured SAXS spectrum (I(q) vs. q), to optimize the parameters of a theoretical SAXS spectrum for one or several populations of scatterers. Works by minimizing an objective function that compares the measured spectrum against the theoretical result. TODO: document the algorithm here.

Input arrays of q and I(q), a string indicating choice of objective function, a dict of features describing the spectrum, and a list of strings indicating which keys in the dict should be used as optimization parameters. The input features dict includes initial fit parameters as well as the flags indicating which populations to include. The features dict is of the same format as SpectrumProfiler and SpectrumParameterization outputs.

Outputs a return code and the features dict, with entries updated for the optimized parameters. Also returns the theoretical result for I(q), and a renormalized measured spectrum for visual comparison.

run()

paws.core.operations.PROCESSING.SAXS.SpectrumParameterization module

class paws.core.operations.PROCESSING.SAXS.SpectrumParameterization.**SpectrumParameterization**

Bases: *paws.core.operations.Operation.Operation*

Determine approximate parameterization for a SAXS spectrum.

The algorithm for guessing parameters for the size distributions of spherical nanoparticles was developed and originally contributed by Amanda Fournier.

The inputs are a SAXS spectrum (I(q) vs. q) and population flags that indicate what scatterer populations to parameterize.

Any preprocessing (background subtraction, smoothing, and any other corrections or cleaning) should be performed beforehand.

run()

paws.core.operations.PROCESSING.SAXS.SpectrumProfiler module

class paws.core.operations.PROCESSING.SAXS.SpectrumProfiler.**SpectrumProfiler**

Bases: *paws.core.operations.Operation.Operation*

This operation profiles a SAXS spectrum (I(q) vs. q) by taking various scalar quantities from the data.

Outputs a dictionary of the results.

This Operation is somewhat robust for noisy data, but any preprocessing (background subtraction, smoothing, or other cleaning) should be performed beforehand.

run()

Module contents

paws.core.operations.PROCESSING.SMOOTHING package

Submodules

paws.core.operations.PROCESSING.SMOOTHING.MovingAverage module

class `paws.core.operations.PROCESSING.SMOOTHING.MovingAverage.`**`MovingAverage`**

Bases: *`paws.core.operations.Operation.Operation`*

Applies moving average smoothing filter to 1d array, optionally weighted by window shape and error values.

`run``()`

paws.core.operations.PROCESSING.SMOOTHING.SavitzkyGolay module

class `paws.core.operations.PROCESSING.SMOOTHING.SavitzkyGolay.`**`SavitzkyGolay`**

Bases: *`paws.core.operations.Operation.Operation`*

Applies a Savitzky-Golay (polynomial fit approximation) filter to 1d data. Uses error bars on intensity if available (default None).

`run``()`

Module contents

paws.core.operations.PROCESSING.ZINGERS package

Submodules

paws.core.operations.PROCESSING.ZINGERS.EasyZingers1d module

class `paws.core.operations.PROCESSING.ZINGERS.EasyZingers1d.`**`EasyZingers1d`**

Bases: *`paws.core.operations.Operation.Operation`*

This Operation attempts to remove zingers from 1d spectral data (I(q) versus q). Zingers are replaced with the average intensity in a window around where the zinger was found.

`run``()`

Module contents

Module contents

paws.core.operations.TESTS package

Submodules

paws.core.operations.TESTS.Identity module

class `paws.core.operations.TESTS.Identity.`**`Identity`**

Bases: *`paws.core.operations.Operation.Operation`*

An Operation testing class, loads its input into its output

`run``()`

paws.core.operations.TESTS.ListPrimes module

class `paws.core.operations.TESTS.ListPrimes.`**`ListPrimes`**

Bases: *`paws.core.operations.Operation.Operation`*

Makes a list of prime numbers in increasing order

`run``()`

paws.core.operations.TESTS.NoiseArray module

class `paws.core.operations.TESTS.NoiseArray.`**`NoiseArray`**

Bases: *`paws.core.operations.Operation.Operation`*

Creates and outputs a square array of noise

`run``()`

Module contents

paws.core.operations.TMP package

Submodules

paws.core.operations.TMP.GetSAXSFlags module

class `paws.core.operations.TMP.GetSAXSFlags.`**`GetSAXSFlags`**

Bases: *`paws.core.operations.Operation.Operation`*

Operation for retrieving SAXS population flags from a set of dicts read in from a previously saved YAML file

`run``()`

Module contents

Submodules

paws.core.operations.OpManager module

class `paws.core.operations.OpManager.`**`OpManager`**

Bases: *`paws.core.models.TreeModel.TreeModel`*

Tree structure for categorized storage and retrieval of Operations.

add_op (*cat*, *opname*)

Add op name to the tree under category cat. If ops.load_flags indicates that this op should be enabled, enable it (this causes it to import the module).

create_tree_item (*parent_itm*, *itm_tag*)

list_ops ()

load_cats (*cat_list*)

load_ops (*cat_op_list*)

Load OpManager tree from input cat_op_list. Format of cat_op_list is [(category1,opname1),(category2,opname2),...]. i.e. each operation in cat_op_list is specified by a tuple, where the first element is a category, and the second element is the name of the Operation. load_cats() should be called before load_ops() and should ensure that all cats in cat_op_list exist in the tree.

n_ops ()

print_cat (*cat_uri*, *rowprefix=' '*)

Generate a string that lists the contents of the operations category specified by cat_uri

remove_op (*op_uri*)

Remove op from the tree by its full category.opname uri

set_op_enabled (*op_uri*, *flag=True*)

paws.core.operations.Operation module

class paws.core.operations.Operation.**InputLocator** (*tp=0*, *val=None*)

Bases: object

Objects of this class are used as containers for inputs to an Operation. They contain the information needed to find the relevant input data.

class paws.core.operations.Operation.**Operation** (*input_names*, *output_names*)

Bases: object

Class template for implementing paws operations.

clear_outputs ()

classmethod clone ()

clone_op ()

Clone the Operation. This should be called after all inputs have been loaded, with the exception of workflow items, e.g. after calling WfManager.prepare_wf().

description ()

self.description() returns a string documenting the input and output structure and usage instructions for the Operation

doc_as_string ()

input_description ()

keys ()

load_defaults ()

Set default types and values into the Operation.input_locators.

output_description ()

run()

Operation.run() should use the Operation.inputs and set values for all of the items in Operation.outputs.

setup_dict()

paws.core.operations.Operation.**parameter_doc**(*name*, *value*, *doc*)

paws.core.operations.optools module

Various tools for working with Workflows and Operations

exception paws.core.operations.optools.**ExecutionError**(*msg*)

Bases: exceptions.Exception

class paws.core.operations.optools.**FileSystemIterator**(*dirpath*, *regex*, *include_existing_files=True*)

Bases: _abcoll.Iterator

next()

paws.core.operations.optools.**dict_contains_uri**(*uri*, *d*)

paws.core.operations.optools.**get_uri_from_dict**(*uri*, *d*)

Module contents

paws.core.operations.**disable_ops**(*disable_root*)

paws.core.operations.**load_ops_from_path**(*path_*, *pkg*, *cat_root=''*)

paws.core.operations.**save_config**()

Call save_config() before closing to save the state of which ops are enabled/disabled.

paws.core.plugins package

Submodules

paws.core.plugins.CitrinationPlugin module

class paws.core.plugins.CitrinationPlugin.**CitrinationPlugin**

Bases: *paws.core.plugins.PawsPlugin.PawsPlugin*

Wrapper contains a Citrination client and implements the PawsPlugin abc interface.

content()

description()

ship_dataset(*pifs*)

start()

stop()

paws.core.plugins.PawsPlugin module

class `paws.core.plugins.PawsPlugin.`**`PawsPlugin`**(*input_names*)

Bases: `object`

`content`()

PawsPlugin.content() returns a dict containing the meaningful objects contained in the plugin. The default implementation returns an empty dict.

`description`()

PawsPlugin.description() returns a string documenting the functionality of the PawsPlugin. The default implementation returns no description.

`keys`()

`start`()

PawsPlugin.start() should perform any setup required by the plugin, for instance setting up connections and reading files used by the plugin. The default implementation does nothing.

`stop`()

PawsPlugin.stop() should provide a clean end for the plugin, for instance closing all connections and files used by the plugin. The default implementation does nothing, assumes the plugin can be cleanly terminated by dereferencing.

paws.core.plugins.PluginManager module

class `paws.core.plugins.PluginManager.`**`PluginManager`**(***kwargs*)

Bases: *`paws.core.models.TreeModel.TreeModel`*

Tree structure for managing paws plugins.

`add_plugin`(*pgin_name*, *pgin*)

Add a plugin, with key specified by pgin_name. If pgin_name is not unique (i.e. a plugin with that name already exists), this method will overwrite the existing plugin with a new one.

`build_tree`(*x*)

Reimplemented TreeModel.build_tree() so that TreeItems are built from PawsPlugins and Workflows and Operations.

`get_plugin`(*pgin_tag*)

`list_plugin_tags`()

`load_from_dict`(*pgin_name*, *pgin_spec*)

Load plugins from a dict that specifies their setup parameters.

`load_plugin`(*pgin_module*)

`n_plugins`()

`plugin_setup_dict`(*pgin*)

`write_log`(*msg*)

paws.core.plugins.SpecClientPlugin module

class `paws.core.plugins.SpecClientPlugin.`**`SpecClientPlugin`**

Bases: *`paws.core.plugins.PawsPlugin.PawsPlugin`*

`content`()

description()

receiveLine()

sendCmd(*cmd*)

sendLine(*line*)

send_commands(*cmd_list*)

send_text(*txt*)

start()

stop()

paws.core.plugins.TCPClientPlugin module

class paws.core.plugins.TCPClientPlugin.**TCPClientFactory**(*protocol*)

Bases: twisted.internet.protocol.ClientFactory

buildProtocol(*addr*)

clientConnectionFailed(*connector*, *reason*)

Clients call this when they are unable to initialize their connection.

clientConnectionLost(*connector*, *reason*)

Clients call this when their connections are lost.

class paws.core.plugins.TCPClientPlugin.**TCPClientPlugin**

Bases: *paws.core.plugins.PawsPlugin.PawsPlugin*

content()

description()

send_text(*txt*)

start()

stop()

class paws.core.plugins.TCPClientPlugin.**TCPTestProtocol**

Bases: twisted.protocols.basic.LineReceiver

addCommand(*cmd*)

connectionLost()

connectionMade()

lineReceived(*line*)

send_lines()

Module contents

paws.core.plugins.**load_plugins**(*path_*, *pkg*)

paws.core.tools package

Module contents

paws.core.workflow package

Submodules

paws.core.workflow.WfManager module

class `paws.core.workflow.WfManager.`**`WfManager`**

Bases: `object`

Manager for paws Workflows. Stores a list of Workflow objects, performs operations on them. Keeps a reference to a PluginManager for access to PawsPlugins.

`add_wf`(*wfname*)

Add a workflow to self.workflows, with key specified by wfname. If wfname is not unique (i.e. a workflow with that name already exists), this method will overwrite the existing workflow with a new one.

`check_wf`(*wf*)

Check the dependencies of the workflow. Ensure that all loaded operations have inputs that make sense. Return a status code and message for each of the Operations.

`get_op`(*wfname*, *op_tag*)

`load_from_dict`(*wfname*, *wf_spec*, *op_manager*)

Create a workflow with name wfname. If wfname is not unique, self.workflows[wfname] is overwritten. Input dict wf_spec specifies Workflow setup, including all operations, Workflow.inputs, and Workflow.outputs.

`locate_input`(*il*)

Return the data pointed to by a given InputLocator object.

`n_wf`()

`prepare_wf`(*wf*, *stk*)

For all of the operations in stack stk, load all inputs that are not workflow items.

`run_wf`(*wfname*)

Execute the workflow indicated by input wfname

`update_embedded_dict`(*d*, *d_new*)

`uri_to_embedded_dict`(*uri*, *data=None*)

paws.core.workflow.Workflow module

class `paws.core.workflow.Workflow.`**`Workflow`**

Bases: *`paws.core.models.TreeModel.TreeModel`*

Tree structure for a Workflow built from paws Operations.

`add_op`(*op_tag*, *op*)

`break_wf_input`(*wf_input_name*)

`break_wf_output`(*wf_output_name*)

build_op_from_dict (*op_setup*, *op_manager*)

build_tree (*x*)
Reimplemented TreeModel.build_tree() so that TreeItems are built from Operations.

classmethod clone ()

clone_wf ()
Produce a Workflow that is a copy of this Workflow.

connect_wf_input (*wf_input_name*, *op_input_uris*)

connect_wf_output (*wf_output_name*, *op_output_uris*)

execute ()

execution_stack ()
Build a stack (list) of lists of Operation uris, such that each list indicates a set of Operations whose dependencies are satisfied by the Operations above them.

static get_valid_wf_inputs (*op_tag*, *op*)
Return the TreeModel uris of the op and its inputs/outputs that are eligible as downstream inputs in the workflow.

get_wf_output (*wf_output_name*)
Fetch and return the Operation output(s) indicated by self.outputs[wf_output_name].

is_op_enabled (*opname*)

static is_op_ready (*op_tag*, *wf*, *valid_wf_inputs*)

keys ()

list_op_tags ()

locate_input (*il*)

n_ops ()

op_dict ()

op_enable_flags ()

static print_stack (*stk*)

set_op_enabled (*opname*, *flag=True*)

set_op_item (*op_tag*, *item_uri*, *item_data*)

set_wf_input (*wf_input_name*, *val*)
Take the Operation input(s) indicated by self.inputs[wf_input_name], and set them to the input value val.

static stack_contains (*itm*, *stk*)

static stack_size (*stk*)

wf_outputs_dict ()

wf_setup_dict ()

Module contents

Submodules

paws.core.pawstools module

exception `paws.core.pawstools.`**`OperationDisabledError`**

Bases: `exceptions.Exception`

exception `paws.core.pawstools.`**`PluginLoadError`**

Bases: `exceptions.Exception`

exception `paws.core.pawstools.`**`PluginNameError`**

Bases: `exceptions.Exception`

exception `paws.core.pawstools.`**`WfNameError`**

Bases: `exceptions.Exception`

exception `paws.core.pawstools.`**`WorkflowAborted`**

Bases: `exceptions.Exception`

`paws.core.pawstools.`**`dtstr`**`()`

Return date and time as a string

`paws.core.pawstools.`**`load_cfg`**`(`*cfg_file*`)`

`paws.core.pawstools.`**`save_cfg`**`(`*cfg_data*, *cfg_file*`)`

`paws.core.pawstools.`**`save_file`**`(`*filename*, *d*`)`

Create or replace file indicated by filename, as a yaml serialization of dict d.

`paws.core.pawstools.`**`timestr`**`()`

Return time as a string

`paws.core.pawstools.`**`update_file`**`(`*filename*, *d*`)`

Save the items in dict d into filename, without removing members not included in d.

Module contents

paws.qt package

Subpackages

paws.qt.widgets package

Submodules

paws.qt.widgets.OpWidget module

class `paws.qt.widgets.OpWidget.`**`OpWidget`**`(`*op*`)`

Bases: `PySide.QtGui.QWidget`

`paintEvent``(`*evnt*`)`

`staticMetaObject` **= <PySide.QtCore.QMetaObject object>**

paws.qt.widgets.PifWidget module

class `paws.qt.widgets.PifWidget.`**`PifWidget`**`(`*itm*`)`

Bases: `PySide.QtGui.QTextEdit`

print_comp(*itm*, *indent*)

print_id(*id_*, *indent*)

print_matrix(*itm*, *indent*)

print_pif(*itm*, *indent*)

print_pifsrc(*itm*, *indent*)

print_procstep(*itm*, *indent*)

print_prop(*itm*, *indent*)

print_qty(*itm*, *indent*)

print_scalar(*itm*, *indent*)

print_value(*itm*, *indent*)

print_vector(*itm*, *indent*)

staticMetaObject = <PySide.QtCore.QMetaObject object>

paws.qt.widgets.WorkflowGraphView module

class paws.qt.widgets.WorkflowGraphView.**WorkflowGraphView**(*wf*, *parent=None*)
Bases: PySide.QtGui.QScrollArea

keyPressEvent(*evnt*)

staticMetaObject = <PySide.QtCore.QMetaObject object>

class paws.qt.widgets.WorkflowGraphView.**WorkflowGraphWidget**(*wf*, *parent=None*)
Bases: PySide.QtGui.QWidget

get_op_coords(*stk*)

op_dims(*op*)

paintEvent(*evnt*)

set_scale(*scl*)

staticMetaObject = <PySide.QtCore.QMetaObject object>

update_coords()

zoom_in()

zoom_out()

paws.qt.widgets.plotmaker_mpl module

paws.qt.widgets.plotmaker_mpl.**array_plot_1d**(*data_in*)

paws.qt.widgets.plotmaker_mpl.**array_plot_2d**(*data_in*)

paws.qt.widgets.plotmaker_mpl.**mpl_array_plot_1d**(*data_in*)

paws.qt.widgets.plotmaker_mpl.**mpl_array_plot_2d**(*data_in*)

paws.qt.widgets.plotmaker_mpl.**plot_mpl_fig**(*fig_in*)

paws.qt.widgets.plotmaker_pqg module

paws.qt.widgets.plotmaker_pqg.**array_plot_1d**(*data_in*)

paws.qt.widgets.plotmaker_pqg.**array_plot_2d**(*data_in*)

paws.qt.widgets.plotmaker_pqg.**plot_mpl_fig**(*fig_in*)

paws.qt.widgets.plotmaker_pqg.**pqg_array_plot_1d**(*data_in*)

paws.qt.widgets.plotmaker_pqg.**pqg_array_plot_2d**(*data_in*)

paws.qt.widgets.text_widgets module

Widgets for displaying text

paws.qt.widgets.text_widgets.**display_text**(*itm*, *indent='&nbsp;&nbsp;&nbsp;&nbsp;'*)

paws.qt.widgets.text_widgets.**display_text_fast**(*itm*, *indent='&nbsp;&nbsp;&nbsp;&nbsp;'*)

paws.qt.widgets.widget_launcher module

This runs various widgets built on the paws.api.

paws.qt.widgets.widget_launcher.**main**()
An entry point for paws full-featured interface.

Module contents

This package defines widgets that are used to communicate with paws.

paws.qt.widgets.**make_widget**(*itm*)

Submodules

paws.qt.QOpManager module

class paws.qt.QOpManager.**QOpManager**

Bases: *paws.core.operations.OpManager.OpManager*, *paws.qt.QTreeSelectionModel.QTreeSelectionModel*

A QTreeSelectionModel for interacting with TreeModel OpManager.

flags(*idx*)

headerData(*section*, *orientation*, *data_role*)

setData(*idx*, *val*, *data_role*)

staticMetaObject = <PySide.QtCore.QMetaObject object>

paws.qt.QPluginManager module

class `paws.qt.QPluginManager.`**`QPluginManager`**

Bases: *`paws.core.plugins.PluginManager.PluginManager`*, *`paws.qt.QTreeSelectionModel.QTreeSelectionModel`*

A Qt Signal-slot manager for a TreeModel PluginManager. Takes a reference to a PluginManager in the constructor. The QPluginManager works mostly by calling on the methods of the PluginManager.

`headerData` (*section*, *orientation*, *data_role*)

`staticMetaObject` **= <PySide.QtCore.QMetaObject object>**

`update_plugin` (*pgin_name*)

paws.qt.QTreeModel module

class `paws.qt.QTreeModel.`**`QTreeModel`** (*flag_dict*)

Bases: *`paws.core.models.TreeModel.TreeModel`*, `PySide.QtCore.QAbstractItemModel`

A Qt Model-View interface for a TreeModel. Required virtual methods: index(), parent(), rowCount(), columnCount(), and data(). Resizeable TreeModels should implement insertRows(), removeRows(), insertColumns(), and removeColumns(). To customize the header in QAbstractItemViews, implement headerData().

`columnCount` (*parent=<PySide.QtCore.QModelIndex(-1, -1, 0x0, QObject(0x0)) >*)

TreeModels by default have one column. More columns can be added by reimplementing columnCount() and then providing for them in TreeModel.data().

`data` (*itm_idx*, *data_role*)

TreeModel's implementation of data() returns the tag of the TreeItem at itm_idx. This is only if itm_idx.column() == 0. Subclasses can reimplement data() to provide meaningful output for other columns, and may consider falling back on super().data() if itm_idx.column() == 0.

`get_data_from_index` (*idx=<PySide.QtCore.QModelIndex(-1, -1, 0x0, QObject(0x0)) >*)

`get_from_index` (*idx=<PySide.QtCore.QModelIndex(-1, -1, 0x0, QObject(0x0)) >*)

For a valid QModelIndex, return idx.internalPointer(). For an invalid index, return None.

`get_index_of_item` (*itm*)

`get_index_of_uri` (*uri*)

`get_uri_of_index` (*idx*)

`headerData` (*section*, *orientation*, *data_role*)

`index` (*row*, *col*, *p_idx*)

Returns QModelIndex address of int row, int col, under QModelIndex p_idx. If a row, column, p_idx combination is invalid, return QModelIndex().

`parent` (*idx*)

Returns QModelIndex of parent of item at QModelIndex index

`remove_item` (*itm_uri*)

`root_index` ()

`root_item` ()

`rowCount` (*parent_idx=<PySide.QtCore.QModelIndex(-1, -1, 0x0, QObject(0x0)) >*)

Either give the number of top-level items, or count the children of parent

`set_item_at_uri` (*itm_uri*, *itm_data*)

staticMetaObject = <PySide.QtCore.QMetaObject object>

tree_dataChanged (*idx*)

tree_update (*parent_itm*, *itm_tag*, *treedata*)

tree_update_at_uri (*itm_uri*, *itm_data*)

paws.qt.QTreeSelectionModel module

class paws.qt.QTreeSelectionModel.**QTreeSelectionModel** (*flag_dict*)

Bases: *paws.qt.QTreeModel.QTreeModel*

QTreeSelectionModel extends QTreeModel by using TreeItem.flags to handle tree item selection.

check_state (*itm*, *flag_key*)

children_flagged (*itm*, *flag_key*)

columnCount (*parent*)

Let QTreeSelectionModel have n_flags+1 columns: one for the TreeItem tag, the rest for flags

data (*idx*, *data_role*)

flags (*idx*)

get_flagged_idxs (*flag_key*, *idx=None*, *val=True*)

headerData (*section*, *orientation*, *data_role*)

is_flagged (*itm*, *flag_key*)

n_flags ()

setData (*idx*, *val*, *data_role*)

set_all_flagged (*flag_key*, *val*, *itm=None*)

set_flagged (*itm*, *flag_key*, *val*)

staticMetaObject = <PySide.QtCore.QMetaObject object>

paws.qt.QWfManager module

class paws.qt.QWfManager.**QWfManager** (*qapp*)

Bases: *paws.core.workflow.WfManager.WfManager*, PySide.QtCore.QObject

A Qt Signal-slot manager for paws Workflows.

add_wf (*wfname*)

Add a QWorkflow to self.workflows, with key specified by wfname. If wfname is not unique (i.e. a workflow with that name already exists), this method will overwrite the existing workflow with a new one.

emitMessage = <PySide.QtCore.Signal object>

launchWorkflow (*wfname*)

relayMessage (*msg*)

run_wf (*wfname*, *pool=None*)

staticMetaObject = <PySide.QtCore.QMetaObject object>

stop_wf (*wfname*)

wfAdded = <PySide.QtCore.Signal object>

wfFinished = <PySide.QtCore.Signal object>

wfStopped = <PySide.QtCore.Signal object>

paws.qt.QWfWorker module

class paws.qt.QWfWorker.**QWfWorker**(*op_dict=None*, *parent_QObject=None*)

Bases: PySide.QtCore.QObject

Container for storing and executing parts of a workflow, to be pushed onto QtCore.QThread(s) as needed.

allDone = <PySide.QtCore.Signal object>

opDone = <PySide.QtCore.Signal object>

staticMetaObject = <PySide.QtCore.QMetaObject object>

work()

paws.qt.QWorkflow module

class paws.qt.QWorkflow.**QWorkflow**

Bases: *paws.core.workflow.Workflow.Workflow*, *paws.qt.QTreeSelectionModel.QTreeSelectionModel*

A QTreeSelectionModel representing a Workflow

add_op(*op_tag*, *op*)

emitData = <PySide.QtCore.Signal object>

emitMessage = <PySide.QtCore.Signal object>

execute()

headerData(*section*, *orientation*, *data_role*)

opFinished = <PySide.QtCore.Signal object>

relayMessage(*msg*)

relayOpData(*op_tag*, *data_uri*, *data*)

staticMetaObject = <PySide.QtCore.QMetaObject object>

updateItem(*item_uri*, *item_data*)

updateOpInput(*opnm*, *inpnm*, *inpdata*)

updateOpItem(*opnm*, *item_uri*, *item_data*)

updateOpOutput(*opnm*, *outnm*, *outdata*)

wfFinished = <PySide.QtCore.Signal object>

paws.qt.UiManager module

class paws.qt.UiManager.**UiManager**(*qpaw*)

Bases: PySide.QtCore.QObject

Uses the QPawsAPI and PySide Qt to provide a widget that controls PAWS.

add_plugin()

add_wf()
Method for adding workflows through the main UI. For this case, the workflow name is inspected to ensure that it doesn't clobber an existing workflow.

add_wf_tab(*wfname*)

append_to_wf_selector(*new_wfname*)

build()
Set up QObjects and model views for communicating with paws objects

display_op_item(*idx*)
Display selected item from the op tree in viewer layout

display_plugin_item(*idx*)
Display selected item from the plugin tree in viewer layout

display_wf_item(*idx*)
Display selected item from the workflow tree in viewer layout

finish_load_state(*ui*)

finish_save_state(*ui*)

load_state()
Start a modal window dialog to choose a .wfl to load a previously saved configuration

logMessage(*msg*)

main_display(*widg=None*)

make_viewer()
Set up the tab viewer widget and display the paws logo in the main viewer

msg_board_log(*msg*)
Print timestamped message to msg board

save_state()
Start a modal window dialog to choose a .wfl to save the current configuration

select_wf(*wfname*)

set_wf(*wf_selector_idx*)

set_wf_treeview(*wfname*)

start_wf(*wfname*)

staticMetaObject = <PySide.QtCore.QMetaObject object>

stop_wf(*wfname*)

toggle_run_wf(*wfname=None*)

update_run_wf_button(*wfname=None*)
If the input wfname indicates the currently selected workflow, make the self.ui.run_wf_button sane wrt this workflow's status.

paws.qt.qtapi module

This minimally enhances the paws.api module to interface with qt-based applications.

class `paws.qt.qtapi.`**`QPawsAPI`**(*app*)

Bases: `paws.api.PawsAPI`, `PySide.QtCore.QObject`

`get_op_from_index`(*idx*)

`get_op_uri_from_index`(*idx*)

`get_plugin_from_index`(*idx*)

`is_wf_running`(*wfname*)

`run_wf`(*wfname*, *pool=None*)

Run the workflow indicated by wfname. If optional threadpool is provided, the workflow attempts to run in that threadpool.

`select_wf`(*wfname*)

`staticMetaObject` = <PySide.QtCore.QMetaObject object>

`stop_wf`(*wfname*)

`wfSelectionChanged` = <PySide.QtCore.Signal object>

`paws.qt.qtapi.`**`start`**(*app*)

Instantiate and return a QPawsAPI object. Requires a valid QApplication as input.

paws.api.start() calls the QPawsAPI constructor.

Returns a QPawsAPI object

Return type paws.api.QPawsAPI

paws.qt.qttools module

Configuration flags, widgets, and functions for the paws qt layer

class `paws.qt.qttools.`**`QSourceEdit`**

Bases: `PySide.QtGui.QTextEdit`

`keyPressEvent`(*evnt*)

`staticMetaObject` = <PySide.QtCore.QMetaObject object>

class `paws.qt.qttools.`**`RunnableExecutor`**(*wf*)

Bases: `PySide.QtCore.QRunnable`

QRunnable that handles execution of a QWorkflow

`run`()

`paws.qt.qttools.`**`bigtext_widget`**(*text=None*)

`paws.qt.qttools.`**`hdr_widget`**(*text*)

`paws.qt.qttools.`**`load_path`**(*ui*, *idx=<PySide.QtCore.QModelIndex(-1, -1, 0x0, QObject(0x0)) >*)

`paws.qt.qttools.`**`message_ui`**(*parent*)

`paws.qt.qttools.`**`name_widget`**(*name*)

`paws.qt.qttools.`**`r_hdr_widget`**(*text*)

`paws.qt.qttools.`**`save_path`**(*ui*, *idx=<PySide.QtCore.QModelIndex(-1, -1, 0x0, QObject(0x0)) >*, *oldidx=<PySide.QtCore.QModelIndex(-1, -1, 0x0, QObject(0x0)) >*)

`paws.qt.qttools.`**`smalltext_widget`**(*text*) paws.qt.qttools.**start_load_ui**(*parent*, *fspath=None*)

paws.qt.qttools.**start_save_ui**(*parent*, *fspath=None*)

paws.qt.qttools.**text_widget**(*text*)

Produce a Read-only Center-aligned QtGui.QLineEdit from input text.

paws.qt.qttools.**toggle_expand**(*trview*, *idx*)

paws.qt.qttools.**toggle_save_button**(*ui*, *txt*)

paws.qt.qttools.**type_selection_widget**(*src*, *widg=None*)

Module contents

paws.qt.**ui_app**(*app_args=[]*)

Return a reference to a new QApplication or a currently running QApplication.

Input arguments are passed to the QApplication constructor. If any exception is thrown, try to find a running QCoreApplication.

Submodules

paws.paws_config module

Module contents

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX