Pylux Developer Reference

for Pylux v0.1-alpha2

J. Page

2016

Contents

1 Introduction

1.1 About this Manual

This manual is intended for developers who wish to contribute to the Pylux system, either through adding to Pylux itself, developing extensions or creating fixture and symbol files. The User Manual looks at Pylux from the perspective of a user, if that's what you're looking for.

This reference goes through all the developer-oriented modules included in Pylux, then how to use the API to create a functional extension, and finally a reference for those wishing to make fixture or symbol files.

You can also access most of the contents of this reference using Python's built-in `pydoc` command.

2 Basic Concepts

At its most basic level, Pylux is a program for the manipulation of XML files called plots. Extensions then allow for the creation of other files from the data in these XML plots.

Both Pylux and any extensions produced for it should follow the standard procedure for the accessing and editing of XML data. You should not access the data directly in your extension, you should instead use the methods provided by the Pylux API. There is a standard sequence for accessing data:

1. The XML data is read from the file and parsed into Python objects using the `xml.etree.ElementTree` module.
2. The data is extracted from these Python objects (which are types defined by the `xml.etree.ElementTree` module) into data types defined by Pylux. These types make editing and accessing data far easier.
3. These objects are edited by the user using an on-screen interface.
4. These objects are written back to the XML tree object, usually using a function called `save()`.
5. The XML tree object is written back to the file on user request or exit.

The main module you will use in your extensions is `plot`, however there are some other modules which you may find useful. These are all documented below.

3 The `plot` Module

3.1 The `PlotFile` Class

This class deals with the Pylux plot file that is being used. There is no initialisation function.

3.1.1 Functions of `PlotFile`

`load(self, path)` Loads the file with location `path` as the working plot file. It will then parse the file into a Python-usable XML tree and store it in `tree`. It will then get the root element from the tree and store it in `root`.

`save()` Saves the current state of the XML working tree to the location from which the plot file was originally loaded.

`saveas(path)` Saves the current state of the XML working tree to the location with path `path`.

`generate(path)` Creates an empty plot file containing the `olplot`, `metadata` and `fixtures` tags at the location with path `path`. This does not load the file as the working file.

3.1.2 Attributes of `PlotFile`

Whilst there are attributes defined in the `PlotFile` class, in general they should not be used by extensions as this breaks down the tiered structure explained in the opening paragraph.

`file` The path of the plot file.

`tree` The Python XML tree element that was parsed from the file using `xml.etree.ElementTree`.

`root` The root element (`olplot`) of the plot file.

3.2 The `DmxRegistry` Class

The `DmxRegistry` class is Pylux's method for managing DMX registries. If you are using more than one DMX registry, each registry should call a new instance of `DmxRegistry`.

3.2.1 Functions of `DmxRegistry`

`__init__(plot_file, universe)` This creates a new Python registry and sets its `universe` attribute to universe that was given as a parameter. It will then search the XML tree for a DMX registry with this universe id. If it finds one with the correct universe id, it will load the contents of that XML registry into the Python registry (a dictionary in the form {`address: (uuid, func)`, ...}). If it cannot find a registry with this universe id, it will instead create a new XML DMX registry object and add this to the tree.

`save()` This function simply saves the current state of the DmxRegistry object to the XML tree.

`get_occupied()` Returns a list of the occupied DMX channels in the Python registry.

`get_start_address(n)` Returns a recommeded start address for a fixture. This function will search through the registry to find the next `n` free DMX addresses in a row.

`address(fixture, start_address)` Assigns DMX addresses to the fixture `fixture`. The function will start assigning at `start_address` if it is an integer, or will call `get_start_address` if the string `auto` is given instead. The number of addresses to assign and the functions of these addresses is deterimned by accessing the fixture's attributes. Additionally, if the fixture has been previously addressed (whether it has or not is determined by getting the value of `universe` from its data), those channels will be removed before the new ones are assigned.

`unaddress(fixture)` Removes the DMX addresses from a fixture `fixture` and from the registry. Removes based on the fixture's `dmx_channels` data tag.

3.2.2 Attributes of `DmxRegistry`

`plot_file` The current plot file is stored as an attribute simply to make accessing it easier, it should not be used externally.

`registry` A dictionary containing all of the registry information, in the form {`address: (uuid, func), ...`}.

universe The name of the universe id of this registry.

xml_registry This is another attribute which is present for internal puproses and should not be referenced by external programs; simply the XML data of the registry.

3.3 The `FixtureList` Class

The `FixtureList` class manages all the fixtures in a plot as a single entity to make filtering and listing easier. This is a small class present to make code cleaner.

3.3.1 Functions of `FixtureList`

`__init__(plot_file)` Finds all the fixtures in XML and makes a `Fixture` object for them, then appends them to a list.

`remove(fixture)` Removes the fixture `fixture` from the plot.

`get_data_values(data_type)` Returns a list of all the values that exist for `data_type` in all the fixtures in the plot.

3.3.2 Attributes of `FixtureList`

`xml_fixture_list` The fixtures list in XML. Should not be referenced externally.

`fixtures` A list of `Fixture` objects, containing all the fixtures in the plot.

3.4 The `Fixture` Class

This class manages individual fixtures and their data.

3.4.1 Functions of `Fixture`

`__init__(plot_file, uuid=None)` Creates a new `Fixture` object in Python and, if a fixture with the UUID `uuid` can be found in the XML file, loads the data of this fixture into the object.

`new(olid, fixtures_dir)` Initialises this fixture as a new fixture. This will load information from a file located in `fixtures_dir` with the name `olid.olf` and copy it into the `Fixture` object.

`add()` Adds the fixture, in its current state to the XML tree as a new fixture. In other words, it appends to the XML tree rather than checking if the fixture already exists.

`load(xml_fixture)` Initialises this fixture as an existing fixture from the XML tree with XML fixture object `xml_fixture`.

`clone(src_fixture)` Clones the contents of `src_fixture` into this fixture.

`save()` Saves the fixture to the XML tree. This will not add the fixture to the tree if it does not already exist in the tree. However, it will manage all other fixture data whether they are existing, new or removed.

`generate_rotation()` Returns a value for rotation based on the fixture's position and focus position. These values must be defined in the fixture's data dictionary before this function is called.

3.4.2 Attributes of `Fixture`

plot_file Like all other instances of `plot_file` as class attributes, this should not be called externally.

data A dictionary containing all the child data of the fixture, except its DMX functions, in the form {`data_name: data_value, ...`}.

dmx_functions A list of all the DMX functions that this fixture has.

olid This fixture's olid.

uuid This fixture's uuid.

xml_fixture Another attribute which should not be referenced externally: the fixture in XML.

dmx_num The number of DMX channels required by this fixture, calculated from `dmx_functions`.

3.5 The `Metadata` Class

The `Metadata` class deals with the metadata section of the plot file.

3.5.1 Functions of `Metadata`

`__init__(plot_file)` Loads the metadata from the XML tree into a Python dictionary.

`save()` Saves the current metadata dictionary to the XML tree.

3.5.2 Attributes of `Metadata`

xml_meta The metadata in XML. Should not be externally referenced.

meta A dictionary containing all the metadata information in the form {`meta_name: meta_value, ...`}.

3.6 The `FixtureSymbol` Class

The `FixtureSymbol` class prepares plain SVG fixtures symbols so that they are ready to be inserted into an SVG plot.

3.6.1 Functions of `FixtureSymbol`

`__init__(path)` Extracts the base group from the SVG image with path `path`.

`prepare(posX, posY, rotation, colour)` Prepares a symbol for inserting into the plot by setting transformations as given by the `rotation` and position parameters and sets the fill of outer paths to `colour`.

3.6.2 Attributes of `FixtureSymbol`

image_group The base image group of the SVG symbol file. This is what should be inserted into the plot file once it has been prepared.

4 The `clihelper` Module

As well as the base `plot` module, the Pylux API includes a helper module to assist in the management of the command-line interface. This doesn't generate an interface like that generated by `curses`, it mainly assists with the processing of on-screen references.

4.1 The `Interface` Class

For your program or extension, if you wish to use the on-screen references, you will need to create one global instance of the `Interface` class which you will use to access and save to the references dictionary.

4.1.1 Functions of `Interface`

`__init__()` Creates a dictionary ready to populate with references, and adds the special reference 'this' with the value `None`.

`append(ref, object)` Append an item to the dictionary with reference `ref` that points to `object`. `object` can be any Python object although is usually a fixture. `ref` should be an integer.

`get(refs)` Return the object associated with the references specified in the string `refs`. `refs` should be a string formatted as a comma separated list of integers and ranges, which are indicated by colon separated limits.

`clear()` Clears all values from the reference dictionary. This should be called before you run any listing action that will add items to the dictionary, unless it is a cummulative list.

`update_this(reference)` Update the special 'this' value so that it points to `reference` which will in turn point to an object. If `reference` is 'this', the value will not be updated as it should already be correct. Generally you would pipe the user input into this each time the user performs an action on a referenced object.

4.1.2 Attributes of `Interface`

`options_list` The current dictionary of options that have interface references. This does not need to be called outside of the module as the provided functions allow for complete manipulation.

4.2 Other Functions

`resolve_references(user_input)` Given `user_input` which is a string of numbers formatted as described above, return a list of every integer that this string represents. The `Interface` class calls this with the `get` function anyway so it shouldn't really be needed elsewhere.

`resolve_input(inputs_list, number_args)` From a list of each input word that has been separated using `split`, return a list of the words, but where the last argument can be multiple words long. This will essentially concatenate any arguments after `number_args` into a single final argument, so if the user inputs `ms production Oedipus Rex` and your program splits this into a list containing the four words separately, you can then run it through the `resolve_input` function, with the number of arguments being two and it will return `['ms', 'production', 'Oedipus Rex']`.

5 Creating Extensions

Currently, when Pylux runs an extension, it looks for extensions in the `/usr/share/pylux/extension` folder. This will change soon to allow for Windows support. Your extension

will not be imported into the Pylux program itself, instead, `runpy` is used to execute your extension. This means that any top-level code in your extension will be executed every time your extension is called. When your extension is called, it will be given the name `pyext`. Therefore, you should have a statement that only runs your main code when `__name__` is `pyext`.

Once your extension is called, it has control over everything. Pylux will no longer provide an interface, or any way to exit your extension. You need to provide all of this in your extension. If your extension is completely command-line driven, you should have the prompt `(pylux:EXTENSION)` to indicate to the user where they are. You should also be able to terminate your main loop and return to Pylux using the `::` action.

6 Creating Fixture Files

A fixture file is simply an XML file giving a template of a fixture that can be used multiple times by the user to speed up the fixture configuration process. Generally a fixture file will contain common constants that will not change from fixture to fixture, such as `type`, `power`, etc. These are defined as top-level XML objects. In the fixture file, there is also a top-level `dmx_functions` object. The sub-elements of this object are loaded as the fixture's DMX functions when it is created. Currently, the user cannot set these options using the program, so fixture files must be used for this. See **??** for a list of standard tags that are applied to fixtures. Of course, any tag can be applied in reality, it just may not be used.

7 Creating Symbol Files

A symbol file defines how a fixture will look when it is rendered on a plan-view plot. This is an SVG file, layed out in a specific way to allow manipulation by the plotting program. Directly below the base `svg` element, there should be a `g` element enclosing the entire fixture with the class `fixture`. This is the element that the plotting program will perform transformations on when it inserts it into the plot.

Beneath that, the fixture can contian any number of `path` elements, provided that they meet the following critera:

1. Any outer paths have the class `outer` and any inner paths have the class `inner`;
2. All paths have `stroke=black` and `stroke-width=1`;
3. An SVG unit represents 1mm in reality;
4. The fixture points towards the right hand side of the image to allow for rotation compatibility;
5. The centre of rotation of the fixture is located at (0,0).