Pylux Documentation

Release 0.3.4

Jack Page

May 09, 2020

User Guide

CHAPTER 1

Introduction

1.1 Synopsis

Pylux is a program designed for the creation and management of documentation for entertainment purposes. Its primary purpose is the creation of documentation for theatrical lighting scenarios.

Pylux can be easily extended to encompass additional functionality and the default installation contains the necessary modules to create documentation in plaintext format of any style using simple templating tools.

1.2 Interface

Everything you would need to do can be done through submitting commands on the command line. In the current implementation, this is done through an interactive prompt, although could be extended to incorporate a GUI, non-interactive CLI or even a web interface.

1.3 Basic Concepts

Pylux is centered around a JSON file which contains all the information about your show. The JSON file is fundamentally unstructured and consists of an unordered list of fundamental objects, which will take one of the types below.

Fixture A single physical lighting fixture, such as a PAR can. A fixture will further contain a list of DMX functions, which act in the same way as the other fundamental objects, but will never appear outside of a fixture object in the show file itself.

Registry A mapping of functions to addresses within a single DMX universe.

Cues A snapshot of the levels of some functions.

Groups A list of fixtures in a certain order.

These data types are referred to often in the remainder of this guide.

Each of these object types will have an arbitrary number of key/value pairs associated with them. These pairs may contain any type of information, but there are four which are common to all object types:

type The type of object. For example metadata or fixture.

uuid A universally unique identifier pointing to this object.

ref A human-readable identifier unique to this object within others of its type. This will be the number which is used to call and pass this object to commands.

label A non-required field which acts as the fallback when no other information about the object is available.

CHAPTER 2

Getting Started

2.1 Invoking

Launch the program by running `pylux` as a module. Alternatively, you can add an entry point into your system PATH.

-h	Print the usage message then exit
-v	Print the version number then exit
-f FILE	Load FILE as the current show file

2.2 File Management

In addition to loading a file whlist launching, you can also load a file by issuing the `File Open path` command once open, which will discard the current file buffer and load the file at `path`. When you need to save the file, run `File Write path`.

If you do not have an existing file, you can begin working straight away. If no file is specified on startup, the program will load `autosave.json`.

2.3 The CLI

The CLI is the default and only included interface to the command interpreter. It is a curses-style interface which will completely take over your terminal window. The screen is split into four areas: a large pane on the left called the Fixed Output Pane, a large pane on the right called the Dynamic Output Pane, a single line at the bottom which is your command-line entry and a line above the command line which displays command history and feedback.

The contents of the Dynamic Output Pane will change based on the commands you run and will display any output the interpreter sends from commands.

The contents of the Fixed Output Pane are dependent on the context you are in. The current context is given by the word preceding the command line. By default this is Fixture. In the Fixture context, the Fixed Output Pane will display a list of all fixtures in your show file. Similarly for cues, groups, etc. There is a special context, All, which will display all items in your show file.

You can change the context by typing the name of the new context twice and pressing enter. For example to change to the cue context type `Cue Cue`. This is a function specific to the CLI and is not sent to the interpreter so is not considered a 'command' as such.

You will notice as you type that many keys do not function as normal. That is because there is a substantial autofill provision. For example, pressing the key `x` will type `Fixture` in the command line for you, to save time typing out the entire word. You can enable and disable autofill by pressing Ctrl+A. This will change the letter preceding the command line from an A (indicating autofill is active) to an X.

2.4 Syntax

Most commands take the form `object refs action params` where:

- `object` is the type of object you will be acting on, for example Fixture.
- `refs` is a single or list of references to these objects, for example 1.
- `action` is what you are doing to this object, for example CopyTo.
- `params` is any further information the command requires. The number of parameters will vary from command to command. For example, CopyTo takes one parameter: the destination references.

References can be a single number:

```
1
```

A range:

```
1>10
```

A list of numbers:

```
1,8,11,15
```

Any combination of the two:

```
1,3>10,13,15
```

A special character meaning all:

```
*
```

A filtered list of numbers or ranges (this means apply filter 1 to the range in brackets):

```
1[2>8,10]
```

A combination of filtered and unfiltered ranges:

```
1[2>8],11,12,2[22,26,29>40]
```

You can also apply a filter to the all character:

```
1[*]
```

Or combine a filter of everything with unfiltered references too (this means show everything which matches filter 1, and also show 8 and 9, regardless of whether they meet the requirements of filter 1 or not:

```
1[*],8,9
```

CHAPTER 3

Using Metadata

Metadata is a special form of data which exists outside of the normal object structure. There are no references. It is simply a list of key/value pairs used to store extra data about a file. There is only one command:

```
Metadata Set title Romeo & Juliet
```

This gives the tag with key ‘title’ the value ‘Romeo & Juliet’. If you want to delete the tag, just run:

```
Metadata Set title
```

That’s it.

CHAPTER 4

Using Fixtures

Fixtures are an important part of a plot. They represent a single physical lighting instrument and are used to create plot drawings and hanging documentation.

Fixtures contain quite a bit more information than metadata: they consist of a data dictionary and a DMX functions list. The data dictionary is simply a key/value list of information about the fixture. The DMX functions list is actually a subvalue of the dictionary and describes how the fixture can be controlled by the DMX protocol.

4.1 Creating Fixtures

Because of the complexity of fixtures, especially those that contain DMX functions, it is not recommended to create them from scratch. Instead, create one from a template then edit from there:

```
Fixture 1 CreateFrom Generic/Parcan
```

This creates a new fixture from the `Generic/Parcan` template. This is an included fixture template with Pylux. 1 is the reference given to this new fixture.

4.2 Displaying Fixtures

You will have seen the fixture appear in the Fixed Output Pane if you are in the Fixture context. You can also show the fixture in the Dynamic Output Pane by running:

```
Fixture 1 Display
```

If you want a bit more information on the fixture, such as additional data tags and DMX functions, you can run:

```
Fixture 1 About
```

4.3 Setting Attributes

By default fixtures do not have names, but it may be useful to give them a label so they are easily identifiable when you have many fixtures of the same type:

```
Fixture 1 Set label SL pipe end
```

Now you will see your fixture has the label SL pipe end, when using both Display and About.

In place of `label`, you may put any arbitrary tag you like, such as `gel`, `posX` etc. For a list of suggested and reserved attributes, see the appendicies.

4.4 Cloning Fixtures

Say we have five more PAR cans that we wish to add, we can use the cloning command to quickly add these between references 2 and 6:

```
Fixture 1 CopyTo 2>6
```

Notice that whenever you supply a unique reference, you can usually supply a range of references to run the command in bulk.

Get information about all of these by running:

```
Fixture * About
```

4.5 Assigning DMX Addresses to Fixtures

The data patching a fixture function to a DMX address exists in Registry objects, although it is a fixture command which is used to assign these addresses:

```
Fixture 1 Patch 0 0
```

This will patch your fixture in universe 0 at address 0. Of course address 0 does not exist, 0 in this case means, the next available set of addresses where this fixture will fit. This is obviously 1 in this case.

The program will automatically create the required registry object for you.

CHAPTER 5

Importing Data

You can import data from an Eos ASCII export:

```
File ImportAscii export.asc eos_patch
```

This imports the Eos patch from the export.asc file. In place of `eos_patch`, you can also specify `cues` or `groups`. Make sure you import the patch before groups or cues, otherwise none of the fixtures you reference in these cues or groups will exist yet.

CHAPTER 6

Generating Reports

Pylux can generate all sorts of plaintext reports for display or printing. All reports are made using a Jinja 2 template. Jinja is a templating software that allows you to create any sort of plaintext template then populate it with the contents of the effects plot.

Create a report from an existing template:

```
Report Create fixturelist.html
```

This creates a report from the fixturelist.html template and stores it in memory. Save it to disk by running:

```
Report Write output.html
```

CHAPTER 7

Generating Plots

If you have SVG image files for your fixtures, you can create a 2D plot of your rig. You need to have given all fixtures you wish to plot in the rig, at minimum a posX and posY value. If you want them to be orientated correctly, you will also need to give them either a rotation value or focusX and focusY values.

Create a new plot and save it in memory:

```
Plot Create
```

Write the plot to disk:

```
Plot Write output.svg
```

There are many many options you can change when creating a plot. You can see what they are by running:

```
Plot About
```

This will display the options in your Dynamic Output Pane. To change any of these, for example scaling, run:

```
Plot Set scale 25
```

You can also change the defaults in the program configuration file.

7.1 Customising the Plot

The base defaults given in the configuration are optimised for the closest output to the USITT standard possible. However, you can change these to suit your particular plot better. You may find that some changes to these options require simultaneous changes to the default stylesheet in order to maintain a cohesive look.

For boolean options, any boolean equivalent is acceptable, for example true, yes, 1, and on are all acceptable in place of True.

7.1.1 Page Layout

paper-size The size of paper to fit the plot to. Accepted are ISO A[0-4]. Note, changing this option should be preferred to just scaling the entire plot to a different size after it has been converted to PDF. The `paper-size` attribute will ensure that line weights and font sizes are kept to standard, and also dynamically resizes the title block based on the paper size. Default `A3`.

orientation Specify landscape or portrait. Default `landscape`.

margin Leave spacing between the edge of the paper and print area. No fixtures will be drawn outside this margin. Some other components may extend beyond the margin if they are set to do so. Measured in millimeters. Default `10`.

page-border Draw a black border around the page. This is drawn inside the margin. Default `True`.

7.1.2 Drawing Options

scale The scale at which to produce the plot. Only metric scales are acceptable, although this number can be a decimal. Default `50`.

plaster-line-padding By default, the centre of the plot area is the intersection of the plaster line and centre line. You can however offset the centre vertically through this option. Positive numbers will increase the area visible in the positive y direction. i.e. the plaster line will be moved down in the output. Measured in unscaled metres. Default `0`.

background-image A path to an vector file which is placed on the drawing surface before anything else. This image must be in the prescribed format and centred about the plaster line / centre line intersection. Default `plot_background.svg`.

line-weight-light, line-weight-medium, line-weight-heavy The plot is based on a three-weight drawing, as prescribed by the USITT standard. Refer to section 6.18 of the standard for which each weight is used for. In addition to this, `line-weight-light` is also used for any general components. Measured in millimeters. Defaults `0.4`, `0.6`, `0.8`

style-source A path to an external CSS file, which defines necessary external styling for both the SVG file itself and the foreignObject HTML injections inside it. Primarily used for text formatting. A default style file is provided with the installation, which can be freely edited by the user. Default `style.css`.

centre-line-dasharray, plaster-line-dasharray An SVG dasharray specification to use for the centre and plaster lines. The default is designed to closely match the USITT specification. Defaults `4, 0.5, 1, 1.5, 3, 0.7`.

centre-line-extend, plaster-line-extend When set to True, the centre and plaster lines will printed beyond the page border into the margin, to the extent of the actual paper. Defaults `False`.

draw-structures When enabled, correctly formatted and tagged structure objects in the file will be rendered onto the plot. Default `True`.

7.1.3 Fixture Icon Options

fallback-symbol Plot will attempt to draw every fixture with position values, even if they do not have a symbol. Specify here the symbol that should be used in the event that the fixture does not have a symbol tag. Default `Generic/Parcan`.

colour-fixtures If set to True, fixtures will be coloured in according to their gel tag. Any gels which can't be converted to RGB, or any fixtures without a gel tag, will be displayed in the default of white. This colouring is applied to all parts of the fixture icon with the `outer` class, whilst white is applied to all parts with the `inner` class. Default `False`.

fallback-handle-north, fallback-handle-south, fallback-handle-east, fallback-handle-west If fixture symbols are being used that do not contain correctly tagged handles, these fallback handles will be used in their place. Primarily used for latching additional data to icons and calculating icon size. Measured in unscaled millimetres. Defaults `0, -200`, `0, 200`, `150, 0`, `-150, 0`

7.1.4 Additional Component Settings

show-channel-number, show-circuit-number, show-dimmer-number These can be toggled in any combination to specify whether the fixture's channel, circuit and dimmer numbers should be displayed next to the fixture icon in the USITT format. Refer to section 6.14.1 of the standard to see what this is. The additional information will only be displayed if it appears in the fixture in `circuit` or `dimmer` tags. The fixture reference is always assumed to be the channel number so will always be printed if this option is enabled. Default `True`.

channel-notation-radius Each of the channel, circuit and dimmer numbers are printed in a box as given by the standard. Use this option to change the nominal size of the boxes. Measured in millimetres. Default `3.1`.

notation-connectors If disabled, will prevent the connector lines between the fixture body and external notation numbers (channel, circuit, dimmer) from being draw. Default `True`.

show-beams If enabled, a line will be printed from the centre of the fixture to it's focus position. A fixture must have both `focusX` and `focusY` tags for this to display. In the event that the focus point is outside of the drawing area, beam lines will extend beyond the border into the margins. Default `False`.

beam-dasharray An SVG dasharray specification for the aforementioned fixture beams. Default `1, 1`.

beam-source-colour If enabled, the beam lines will be printed in the colour matching the source fixture's gel tag. Inconvertible gel names or fixtures without gels will continue to have their beams rendered in black. Default `False`.

show-focus-point Draws a circle at the focus position of each fixture. Similar to the beams option. These will only work on fixtures with focus values and will print in the margins. Default `False`.

focus-point-radius Adjust the radius of the drawn focus point circle. Measured in millimeters. Default `1`.

focus-point-source-colour Similar to the `beam-source-colour` option, if enabled, focus points will be rendered according to the colour of the gel in the source fixture. Default `False`.

7.1.5 Title Block Format

title-block What format of title block to use. Currently supported formats are `None` and `sidebar`. `None` will omit the title block entirely. `sidebar` will draw the title block down the full height on the right hand side of the page.

sidebar-title-width-pc, sidebar-title-min-width, sidebar-title-max-width The width of the sidebar title is calculated as a percentage of the page width, defined by `sidebar-title-width-pc`. Minimum and maximum widths, in millimetres can be provided to ensure that sidebar titles remain sensible widths when changing the paper size. Defaults `0.1`, `50`, `100`.

sidebar-title-padding The amount of space to leave inside the title block, to prevent titles and other items rendering right against the sidebar boundaries. Measured in millimetres. Default `2`.

titles A list of metadata tags to include in the title section of the title block. These are added to an HTML foreignObject element for external styling with the included stylesheet. Only the tag values are added, headings should be added using the ::before CSS selector. Class names given to the text paragraph will be `title-meta_tag_name`. Format as a literal list of strings. Default `['company', 'production', 'venue', 'lighting_designer']`

legend-text-margin The space to leave between the fixture symbol in the legend and its corresponding text label. Measured as a percentage of the overall title bar width. Default `2`.

7.1.6 Scale Rule Settings

show-scale-rule Show a scale rule in the bottom left corner of the page when enabled. Default `True`.

scale-rule-major-increment, scale-rule-minor-increment The scale rule gives you a minor scale (drawn to the left-hand-side) and a major scale (drawn to the right-hand-side). Both can have their increment defined independently. This is the unscaled length to draw each increment at. Measured in metres. Defaults `1`, `0.5`.

scale-rule-major-length, scale-rule-minor-length The overall unscaled length to draw the corresponding side of the rule to. Will only draw complete increments, so any length defined over a whole number of increments will be ignored. For example an increment of 1 and a length of 3.4 will result in a rule of length 3. Measured in metres. Defaults `3`, `2`.

scale-rule-thickness The height of scale rule to draw. This is the height excluding the border line (which is drawn according to `line-weight-light`). Measured in millimetres. Default `1`.

scale-rule-padding The distance to leave between the scale rule and the lower left hand corner of the plot area boundary. The same distance is left on both the x and y axis and this is measured to the lower left hand corner of the rule itself, not any associated text. Measured in millimetres. Default `3`.

scale-rule-label-padding The distance to leave between the top of the rule itself and the labels marking the distances on the rule. Measured in millimetres. Default `0.5`.

scale-text-padding The distance to leave between the top of the rule itself and the associated text labelling the scale of the plot (in the form SCALE 1:x). This will likely only require changing if you change the marking labels font size in the stylesheet. Measured in millimetres. Default `3.5`.

scale-rule-units The name to give to the units on the scale rule, as printed to the right of it. Default `metres`.

CHAPTER 8

Base Reference

8.1 Cue Commands

8.1.1 Cue About

Usage `Cue refs About`

Synopsis Show the intensities of all fixtures recorded in `refs`.

8.1.2 Cue Create

Usage `Cue refs Create`

Synopsis Create empty cues at `refs`.

8.1.3 Cue Display

Usage `Cue refs Display`

Synopsis Show a single-line summary of `refs`.

8.1.4 Cue Query

Usage `Cue refs Query`

Synopsis Show the levels of all intensities and non-intensity parameters in `refs`.

8.1.5 Cue Remove

Usage `Cue refs Remove`

Synopsis Remove `refs` from the show file entirely.

8.1.6 Cue Set

Usage `Cue refs k v`

Synopsis Set arbitrary data tag `k` to `v` in `refs`.

8.1.7 Cue SetIntens

Usage `Cue refs SetIntens fixs level`

Synopsis Set the intensity of `fixs` to `level` in `refs`.

8.2 File Commands

8.2.1 File Write

Usage `File Write path`

Synopsis Save the current working file to `path`.

8.3 Filter Commands

8.3.1 Filter Create

Usage `Filter refs Create k v`

Synopsis Create a filter at `refs` with requirement that arbitrary data tag `k` has value `v`.

8.3.2 Filter Remove

Usage `Filter refs Remove`

Synopsis Remove `refs` from the show file entirely.

8.4 Fixture Commands

8.4.1 Fixture About

Usage `Fixture refs About`

Synopsis Show all additional data tags and DMX functions of `refs`.

8.4.2 Fixture Create

Usage `Fixture refs Create`

Synopsis Create empty fixtures at `refs`.

8.4.3 Fixture CreateFrom

Usage `Fixture refs CreateFrom template`

Synopsis Create fixtures at `refs`, using additional data tags and DMX functions from `template`.

8.4.4 Fixture CompleteFrom

Usage `Fixture refs CompleteFrom template`

Synopsis For any additional data tags which exist in `template` but not `refs`, copy the tag and value from `template` to `ref`. Also copy the entire DMX personality if there is no personality in `refs`.

8.4.5 Fixture CopyTo

Usage `Fixture ref CopyTo dests`

Synopsis Make a copy of `ref` at `dests`.

8.4.6 Fixture Display

Usage `Fixture refs Display`

Synopsis Show a single-line summary of `refs`.

8.4.7 Fixture Patch

Usage `Fixture refs Patch universe address`

Synopsis Patch `refs`, beginning at `address`, in `universe`.

8.4.8 Fixture Remove

Usage `Fixture refs Remove`

Synopsis Remove `refs` entirely from the show file.

8.4.9 Fixture Set

Usage `Fixture refs Set k v`

Synopsis Set arbitrary data tag `k` to `v` in `refs`.

8.4.10 Fixture Unpatch

Usage `Fixture refs Unpatch`

Synopsis Remove all entries in all universes of `refs`.

8.5 Group Commands

8.5.1 Group About

Usage `Group refs About`

Synopsis Show the constituent fixture references of `refs`.

8.5.2 Group Append

Usage `Group refs Append fixs``

Synopsis Add fixtures `fixs` to the end of `refs`.

8.5.3 Group Create

Usage `Group refs Create`

Synopsis Create empty groups at `refs`.

8.5.4 Group Display

Usage `Group refs Display`

Synopsis Show a single-line summary of `refs`.

8.5.5 Group Query

Usage `Group refs Query`

Synopsis Show a single-line summary of each fixture in `refs`.

8.5.6 Group Remove

Usage `Group refs Remove`

Synopsis Remove `refs` entirely from the show file.

8.5.7 Group Set

Usage `Group refs Set k v`

Synopsis Set arbitrary data tag `k` to `v` in `refs`.

8.6 Metadata Commands

8.6.1 Metadata Set

Usage `Metadata Set k v`

Synopsis Set the value of `k` to `v`. Omit `v` to delete an existing entry under `k`.

8.7 Registry Commands

8.7.1 Registry About

Usage `Registry refs About`

Synopsis Show a table-style overview of used addresses in `refs`.

8.7.2 Registry Create

Usage `Registry refs Create`

Synopsis Create empty registries at `refs`.

8.7.3 Registry Display

Usage `Registry refs Display`

Synopsis Show a single-line summary of `refs`.

8.7.4 Registry Query

Usage `Registry refs Query`

Synopsis Show a single-line summary of every patched function in `refs`.

8.7.5 Registry Remove

Usage `Registry refs Remove`

Synopsis Remove `refs` entirely from the show file.

CHAPTER 9

Developer Introduction

This program is badly written.

The developer documentation here is purely so I don't forget how it works.

9.1 General Structure

A front-facing *interface* sends raw commands to an *interpreter* and receives output via a message bus.

An interpreter may be extended through modules called extensions.

All document interaction takes place in `document.py`.

CHAPTER 10

Interface Specification

The interface of the program is the point at which commands are issued to the interpreter, and the results and outputs of commands relayed back to the end user. There is no requirement for how the interface itself operates, or indeed how many of the potential features it incorporates. However, it must communicate with the interpreter in a specific way.

For code-based reference, see `cli.py` which is the included interface. It should be relatively easy to understand that file with the information in this page.

10.1 Launching

The main process of the interface must be located in a function called `main`, which has one argument. This argument is the initialisation globals which are passed on launch of the program. These will contain the file to load, parsed configuration, and potentially further globals in the future.

10.2 Structure

Every interface must initialise an instance of `interpreter.Interpreter`, to which it must pass on construction the working show file, a message bus object, and the configuration file.

The working show file is the deserialised JSON document which has been parsed into a Python list. It is *not* the load location of the file. The configuration file will have already been parsed into a dict when the main process was launched, so this can safely be passed straight to the interpreter instance.

You must extend the interpreter manually with any extensions you wish to use, even the `base` extension. If you do not add any extensions, the interpreter will not respond to any commands. To extend the interpreter, just pass the string of the extension name to the `register_extension` function of your interpreter object.

10.3 Sending Commands

Commands are sent to the interpreter by sending the raw input string to the `process_command` function of your interpreter instance. There is no need to process the string in anyway before sending it. If your interface relies on the command line to perform interface-specific functions (for example, the CLI interface uses special unused commands to change context), then you can process the command separately.

10.4 Receiving Feedback and Output

Feedback (whether a command was successful or not) and output (data the command returns) are received through the message bus object you passed to the interpreter instance on initialisation. This message bus object is a class which must have two functions: `post_feedback` and `post_output`. These are functions the interpreter will use to communicate output back to your interface.

Both of these will receive text feedback in the format described below.

10.5 Format of Text Output

As line breaks are very important in the interpretation of the data the interpreter returns, all text output is returned as a list of lines, even if it is only one line long. Each of these lines (list items) could themselves be:

- a string
- a tuple
- a list of strings and tuples

Any tuple returned will be of the form (FORMAT, STRING). This is to flag to your interface that there is specific formatting to be added to STRING which will enhance its legibility. For example, it could be a colour to indicate a specefic object type or status. The actual formatting to apply will not be specified, FORMAT is simply a string indicating the *type* of formatting to add. For example `fixture` or `function`. How you interpret these format strings is entirely up to you. You may just completely ignore them if you wish, although this could make the output more difficult for the end user to interpret.

A list of strings and tuples should be interpreted as a concatenation of the constituent strings.

CHAPTER 11

Adding Object Types

Adding object types is now substantially quicker due to new streamlined universal commands.

To create a new object type:

Add an object definition to `lib.constant`, which will define the internal name for the object and the structure of a blank such object in the file. Here should be defined any fields which are required for the object to work, for example the `levels` field in a cue object.

Bootstrap off base commands in base. Available are create, display, remove and set. For each of these for your new object, all you need is:

```
def new_object_create(self, refs):
    return self._base_create(refs, constant.NEW_OBJECT_TYPE)
```

And then the same for display, remove and set.

You also need to register each of these commands by adding to `register_commands`:

```
self.commands.append(RegularCommand(('NewObject', 'Create'), self.new_object_create,␣
↪check_refs=False))
```

The `check_refs=False` flag is only required for the create command.

Finally, add any other specialist commands to base and define any keymaps in the config file.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search