specviz Documentation

Release 0.5.0

JDADF Developers

Jun 06, 2018

Contents

SpecViz is a tool for visualization and quick-look analysis of 1D astronomical spectra. It is written in the Python programming language, and therefore can be run anywhere Python is supported (see *Installation*). SpecViz is capable of reading data from FITS and ASCII tables (see *Custom Loaders*).

SpecViz allows spectra to be easily plotted and examined. It supports instrument-specific data quality handling, flexible spectral units conversions, custom plotting attributes, plot annotations, tiled plots, and other features.

SpecViz notably includes a measurement tool for spectral lines which enables the user, with a few mouse actions, to perform and record measurements. It has a model fitting capability that enables the user to create simple (e.g., single Gaussian) or multi-component models (e.g., multiple Gaussians for emission and absorption lines in addition to regions of flat continuua). SpecViz incorporates various methods for fitting such models to data. For more details, see *Model Fitting*.

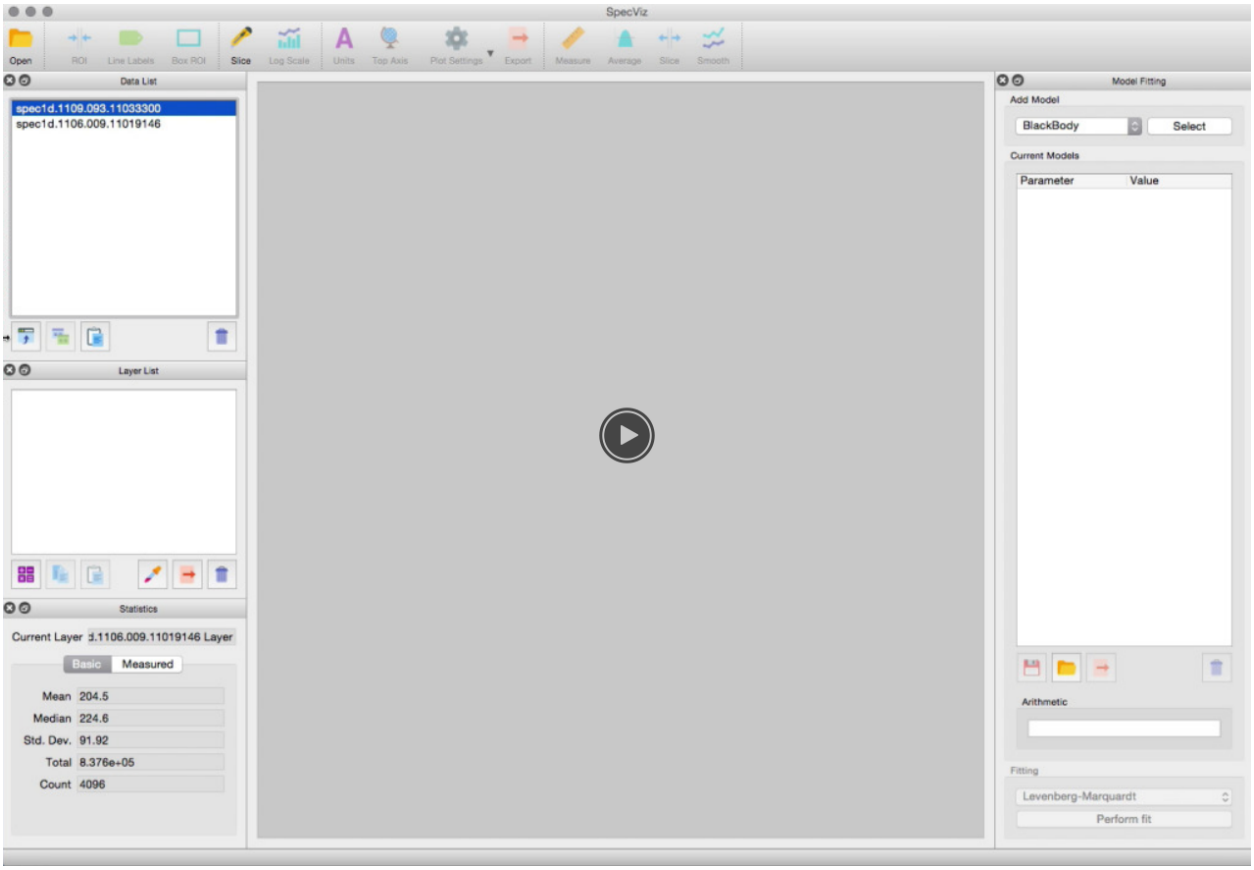
Furthermore, SpecViz allows for overplotting or combining of spectra.

SpecViz will soon include the ability to

- Measure the average of multiple spectra, detrending, and apply Fourier filters.
- Interactively renormalize data from spectral templates.
- Overplot spectral line lists.
- And more...

Part I

Demo


SpecViz
Open
ROI
Line Labels
Box ROI
Slice
Log Scale
Units
Top Axis
Plot Settings
Export
Measure
Average
Slice
Smooth
Data List
spec1d.1109.093.11033300
spec1d.1106.009.11019146
Layer List
Statistics
Current Layer 3.1106.009.11019146 Layer
Basic
Measured
Mean 204.5
Median 224.6
Std. Dev. 91.92
Total 8.376e+05
Count 4096
Model Fitting
Add Model
BlackBody
Select
Current Models
Parameter
Value
Arithmetic
Fitting
Levenberg-Marquardt
Perform fit

Part II

Installation and Setup

CHAPTER 1

Installation

SpecViz is distributed through the Anaconda package manager. Specifically, it lives within Space Telescope Science Institute's AstroConda channel.

If you do not have Anaconda, please follow the instructions here to install it, or scroll down for manual installation of SpecViz.

1.1 Install via Anaconda

If you have AstroConda setup, then all you have to do to install SpecViz is simply type the following at any Bash terminal prompt:

```
$ conda install specviz
```

If you do not have AstroConda setup, then you can install SpecViz by specifying the channel in your install command:

```
$ conda install --channel http://ssb.stsci.edu/astroconda specviz
```

At this point, you're done! You can launch SpecViz by typing the following at any terminal:

```
$ specviz
```

1.1.1 Uninstalling

To uninstall via Anaconda, simply type the following at a command line:

```
$ conda uninstall specviz
```

1.2 Install via source

SpecViz can also be installed manually using the source code and requires the following dependencies to be installed on your system. Most of these will be handled automatically by the setup functions, with the exception of PyQt/PySide.

- Python 3 (recommended) or Python 2
- PyQt5 (recommended), PyQt4, or PySide
- Astropy
- Numpy
- Scipy
- PyQtGraph
- qtpy

1.2.1 By using `pip`

Clone the SpecViz repository somewhere on your system, and install locally using `pip`. If you are using an Anaconda virtual environment, please be sure to activate it first before installing: `$ source activate <environment_name>`.

```
$ pip install git+http://github.com/spacetelescope/specviz.git@v0.4.4
```

This uses the `pip` installation system, so please note that

1. You need to have `pip` installed (included in most Python installations).
2. You do **not** need to run `python setup.py install`.
3. You do **not** need to install the dependencies by hand (except for PyQt).

Likewise, the `pip` command will use your default Python to install. You can specify by using `pip2` or `pip3`, if you're not using a virtual environment.

1.2.2 By cloning

You may also install by cloning the repository directly

```
$ git clone https://github.com/spacetelescope/specviz.git
$ cd specviz
$ git checkout tags/v0.3.0
$ python setup.py install
```

1.2.3 PyQt/PySide bindings

SpecViz requires PyQt. Currently, only python environments with 3.5 or higher installed can use `pip` to install PyQt5, in which case simply type:

```
$ pip install pyqt5
```

to install it on your system.

In any other case, PyQt can be installed via anaconda:

```
$ conda install pyqt
```

SpecViz works with with PyQt4 and PySide, but it is recommended that users use PyQt5 if available.

1.2.4 Uninstalling

To uninstall via `pip`, simply type the following at a command line:

```
$ pip uninstall specviz
```

1.3 Known Issues

On a Mac with Qt5, depending on exactly how you have set up Anaconda, you might see the following error after following the above instructions:

```
This application failed to start because it could not find or load the Qt platform plugin "cocoa".

Reinstalling the application may fix this problem.
```

If you see this message, you have encountered an incompatibility between Anaconda's packaging of Qt4 and Qt5. The workaround is to uninstall Qt4 with the following command:

```
$ conda uninstall pyqt qt
```

and SpecViz should now happily run.

Conversely, if you've had PyQt5 installed previously and you wish to run the PyQt4 version, you may run into a similar error:

```
$ RuntimeError: the PyQt4.QtCore and PyQt5.QtCore modules both wrap the
QObject class
```

This issue can be solved with the following command:

```
$ conda uninstall pyqt5 qt5
```

CHAPTER 2

Launching SpecViz

Once you've installed SpecViz, you can launch it via the command line:

```
$ specviz
```

If you only wish to inspect a single FITS or ASCII file using the default *Custom Loaders* file formatting, you can also pass in the filename as a command line argument, as follows:

```
$ specviz filename
```

You may also include the name of a custom loader as second optional argument:

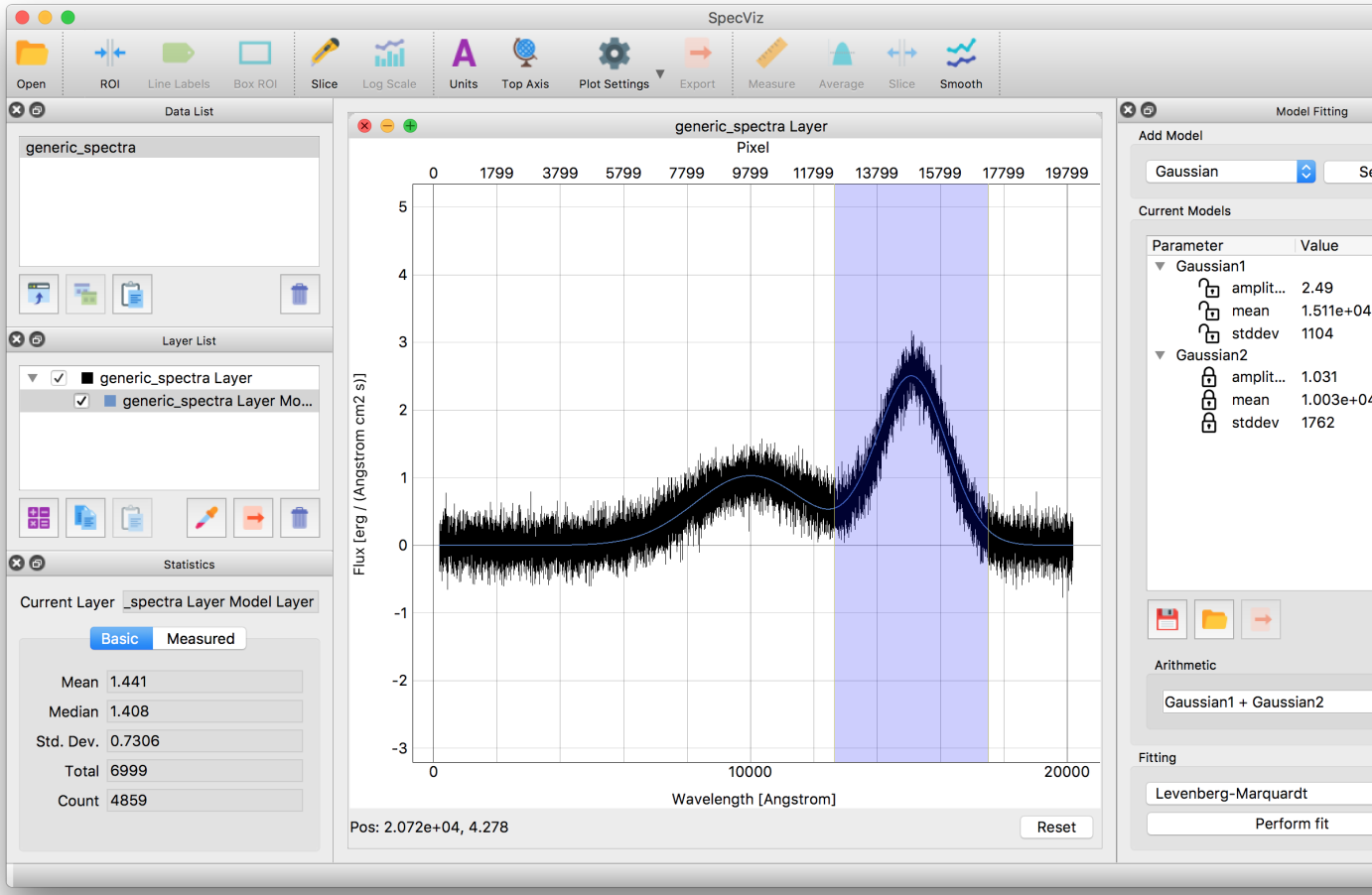
```
$ specviz filename --format="my-custom-format"
```

Part III

Using SpecViz

CHAPTER 3

Viewer



Note: Some buttons and menu options are not functional yet, as this is a work in progress.

To open a file:

1. Click on the folder icon on top left corner or select `File -> Open` from menu.
2. Select the desired file and data type in the file dialog and click "Open".
3. The file will be listed in the "Data List" window (top half of left panel).
4. To remove the file from "Data", click on "trash can" icon at lower right corner of the "Data List" window.

To plot the spectrum that is now loaded, **click on the plot icon** (lower left corner of the Data window). It will be plotted in the central window. Basic statistics for the displayed spectrum will be shown in the "Statistics" window at the lower left.

Each spectrum loaded will have its own layer, as shown the "Layer List" window at the lower left. New layers are created each time a spectrum is modified or a fit is performed, and the original spectrum is untouched. To work on the file of interest, click on its plot window to bring it in focus.

To adjust plot display:

1. If your mouse has center wheel, it can be used to zoom in and out while the cursor is over the plot.
2. Click on the "big wheel" icon on top of the plot to change top axis parameters or displayed units.
3. Left click on the plot to bring up a context menu.
4. Select "View All" to view all available data (if zoomed in).
5. Select "X Axis" or "Y Axis" to adjust respective axis behaviors.
6. Select "Mouse Mode" to toggle between 3-button and 1-button mouse.
7. More plot options are available under "Plot Options".
8. Select "Export…" to save plot out as an image.

To select a region of interest (ROI):

1. Select the layer from "Layers" list (if you have multiple).
2. Click on the "rectangle" icon on the top left of the plot. An adjustable rectangle will appear on plot display.
3. Click and drag the edges to adjust wavelength coverage. As the region changes, basic statistics in the "Statistics" window on bottom left of the viewer will also update accordingly.
4. Click inside the rectangle and drag to reposition it without resizing.
5. Click the left-most "knife" button under "Layers" (bottom left of the left panel) to create a new layer slice from the ROI.
6. To remove a layer slice, select the layer and click on "trash can" button under "Layers".
7. To remove ROI, left click while it is highlighted and select "Remove ROI" from the context menu.

Note: Layer arithmetic (the "calculator" button under "Layers") is work in progress.

To measure equivalent width and related properties:

1. Select the layer from "Layers" list (if you have multiple).
2. Add three ROIs.

3. Position the orange ROI over the emission or absorption line of interest and adjust accordingly.
4. Position each of the blue ROIs over two different continuum areas, one on each side of the emission/absorption line.
5. As you adjust the regions, values for equivalent width etc. will be updated under the "Measured" tab under "Statistics" on top right of the viewer.

Note: When multiple ROIs are present, the statistics in "Basic" tab reflect all data points contained in the ROIs.

To fit a model to the selected ROI, see *Model Fitting*.

To quit SpecViz, select `File -> Exit` from menu.

CHAPTER 4

Model Fitting

SpecViz utilizes Astropy Models and Fitting to fit models to its spectra. For example, you can fit one model to the continuum, another to an emission line of interest, and yet another to an absorption line.

Currently, the following models are available:

SpecViz Model Name	Astropy Model Class
BrokenPowerLaw	`BrokenPowerLaw1D`
Const	`Const1D`
ExponentialCutoffPowerLaw	`ExponentialCutoffPowerLaw1D`
Gaussian	`Gaussian1D`
GaussianAbsorption	`Gaussian1D` with negative amplitude
Linear	`Linear1D`
LogParabola	`LogParabola1D`
Lorentz	`Lorentz1D`
MexicanHat	`MexicanHat1D`
Trapezoid	`Trapezoid1D`
PowerLaw	`PowerLaw1D`
Scale	`Scale`
Shift	`Shift`
Sine	`Sine1D`
Spline	`UnivariateSpline`
Voigt	`Voigt1D`

The models can be fitted with the following fitters:

SpecViz Fitter Name	Astropy Fitter Class
Levenberg-Marquardt	`LevMarLSQFitter`
Simplex	`SimplexLSQFitter`
SLSQP	`SLSQPLSQFitter`

To use a model:

1. Select the layer you wish to operate on from the "Layer List" window in the bottom left. For example, you can choose the layer containing your emission or absorption line.
2. Create and position a region of interest (ROI) as described in the *Viewer* section of the documentation.
3. Select the desired model from the `Add Model` drop-down box and click `Select` to add it to `Current Models`.
4. If desired, repeat the above step to add additional models.
5. A new model layer will be created under your data layer in the "Layer List" window.

To edit model parameters or enter a better first estimate of the model parameters:

1. Select the model layer in the "Layer List" window that contains the desired model.
2. If desired, double-click on the model name to rename it. When you see a blinking cursor, enter its new name and press "Enter".
3. Expand the model listing under `Current Models` on the right of the viewer.
4. Double-click on the desired model parameter value in the listing. When you see a blinking cursor, enter the new value and press `Enter`.

To fit a model:

1. Select the model layer under `Layers` that contains the model(s) you wish to fit to your data.
2. Click the lock icon next to any parameter to choose whether it should be kept fixed (closed lock) or allowed to vary (open lock) during fitting.
3. Select the desired fitter from `Fitting` using its drop-down menu.
4. Click `Perform Fit`. This may take up to a few seconds, depending on the complexity of the fit.
5. The associated model parameters will be adjusted accordingly.

The `Arithmetic` text box is used to define the relationship between different models for the same layer. If nothing is defined, the default is to add all the models together. To describe a non-default model relationship, adjust the math operators, as shown in the examples below and then press `Enter` to produce the compound model:

```
Linear1 + Gaussian1
```

```
Linear1 * Gaussian1
```

```
Gaussian1 - Gaussian2
```

The entity that results from lumping together all the models, and combining them either using the arithmetic behavior expression, or just adding them all together, is called a "compound model".

4.1 Model names

When added to the `Current Models` list, a model will receive a default name that is generated from the model type (as listed in the drop down model selector) plus a running numerical suffix.

These names can be changed by clicking on the default name and entering a new name. Note that changing model names will require that any expression in the `Arithmetic` text box be edited accordingly.

For now, we are limited to only alphanumeric characters (and no white spaces) when re-naming models.

4.2 Spline model

Note that the Spline model is of an intrinsically different nature than the other models included in the drop down list of models. The Spline model, when added to a pre-existing list of models, or when added by itself to an empty list, will immediately be fit to the data within the currently defined Regions Of Interest (ROIs). That is, being a linear model, there is no need to iterate in search of a "best fit" spline. It is just computed once and for all, and kept as part of the compound model that is built from the models in the list and the arithmetic behavior expression.

This implies that, to change the regions of interest that define the spline, one must remove the spline from the list of models. Then, the user must redefine the ROIs and add a new spline to the list of models to be fit. To change a spline parameter, there is no need do discard the spline. Just do it in the same way as with other models: just type in the new value for the parameter.

Subsequently, when the fitter iterates the compound model in search of a best solution, the spline model will act as a constant. That is, it will be used to compute the global result of the compound model, but its parameters won't be accessed, and varied, by the fitter. Thus, the spline parameters are not fitted, they are just a convenient mechanism that enables user access to the parameter's values.

The documentation for the spline model can be found here:

http://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.interpolate.UnivariateSpline.html

Note that SpecViz provides access, at this point, to just two of the parameters in the scipy implementation of th spline function. Pay special attention to the `smooth` parameter. SpecViz initializes it to a 'best guess' (`len(wavelength)`). Too small of a value may cause the spline to enter an infinite loop. Change the `smooth` value with care, trying to stay close to the default value.

Note: Model arithmetic is a work in progress.

4.3 Saving and exporting models to a file

Selecting a model layer under "Layers" will enable the *Save* (the "floppy disk" icon) and *Export* (the "out the door" icon) buttons under "Current Models" on the right of the viewer. When a model is saved as a file on disk, it can later be read into SpecViz as described below. Exporting a model to a file wil create a Python script in a `.py` file. This file can be directly imported by Python in a command-line session.

Click on either button to get a file dialog window. Provide a name for the model file, if this file name does not end with the correct suffix, the suffix will automatically be appended. Click "Save", or just press the Return/Enter key. The correct suffix for saved and exported files are `.yaml` and `.py`, respectively.

4.4 Save and load

Saving the model to a file works in the same way as *Export*. The difference is that a saved model can be later read back into SpecViz via the "Load" button (the "folder" icon), also under "Current Models", whereas the exported model cannot.

For the "Load" button to be enabled, a data (spectrum) layer (not a model layer) must be selected in the "Layer List" window. The selected `.yaml` model file will generate a model that will be attached to a new model layer associated with the selected data layer.

The file is writen in the YAML format. Being a plain text file with a self-explanatory structure, it can be edited at will by the user, e.g., to add bounds, fixed flags, and ties to the model parameters. Note that these extra, user-defined

attributes, won't be accessible from SpecViz's user interface. They will however, be used by the fitter when a fit is run on the model. They will also be written out correctly, either when saving or exporting the model.

Note: YAML format for saved models and usage of advanced features like bounds and fixed flags are work in progress.

4.5 Export

This will save the model in the currently selected model layer to a file that can be directly imported by Python. The file is just a plain text file with the model given as a Python expression. The model is associated a variable named `'model1'`.

The following example uses the `'test3.py'` file name, and a model comprised of a constant and a gaussian:

```
>>> import test3
>>> test3
<module 'test3' from '/my/saved/models/test3.py'>
>>> test3.model1
<CompoundModel0(amplitude_0=0.297160787184, amplitude_1=2.25396100263, mean_1=15117.1710847, stddev_
↪1=948.493577186)>
>>> print(test3.model1)
Model: CompoundModel0
Inputs: ('x',)
Outputs: ('y',)
Model set size: 1
Parameters:
     amplitude_0    amplitude_1      mean_1       stddev_1
    -------------- ------------- ------------- -------------
    0.297160787184 2.25396100263 15117.1710847 948.493577186
```

The file can be edited by the user, e.g., to add bounds, fixed flags, and ties to the model parameters.

Warning: Security issues with importing model this way into Python and usage of advanced features like bounds and fixed flags are work in progress.

CHAPTER 5

Spectral line labels

SpecViz can display spectral line identifications ("Line labels") on top of spectral data being displayed. Line identifications are supplied by line lists. There is a set of line lists already included in the distribution, and the user can read his/hers own line lists as text files formatted in Astropy's ECSV format.

One a spectrum is displayed in the plot window, clicking on the green "Line Labels" button in the main menu bar will bring up a floating window which contains the functionality to manage the line lists.

5.1 Selecting line lists

Use the drop-down menu to select a line list. Alternatively, use the File menu button (the yellow folder) to read a line list from a user-supplied file.

Currently, the following line lists are supplied within the package and available via the drop-down menu:

Line list	Number of lines	Wavelength range
Atomic-Ionic	42	0.97 - 3.95 μ
CO	66	1.56 - 2.51 μ
H-He	145	0.72 - 3.74 μ
H2	226	1.09 - 3.99 μ
Common Stellar	95	1,215 - 10,938
Common Nebular	51	3,430 - 7,130
ILSS	25,800	2,950 - 13,160
Reader-Corliss	46,646	16 - 39,985
SDSS	48	1,034 - 8,660

Once a line list is selected, a dialog pop-up will ask what is the wavelength range one wants to read from the list. The dialog is populated by default with the wavelength range spanned by the spectrum being displayed. Typing in new values in the dialog text fields will retain then during subsequent uses of the dialog, until they are re-typed again.

The dialog will display the actual number of lines that will be read from the list. Large numbers of lines trigger an alert, in the form of red text.

The two large lists must be handled with care, because if one attempts to read or select the entire list, some functionality may be affected adversely and become very slow. It is recommended that small wavelength ranges be used with those lists, in a way that no more than about 2,000 lines be read.

5.2 Line list management

Each line list, once read, ends up in a table that is placed under a separate tab. The tab name is the list name. Each tabbed panel contains several sections, described below:

1. Header with descriptive information on the line list.
2. The table itself. The exact column names and contents are list-dependent. Column headers can be clicked; that way, the column is sorted in ascending/descending order upon successive clicks. Hovering the mouse on a column heading may bring additional information on the column, such as units.
3. Control section. This contains a number of controls to help configure the display of the selected lines in the table

Specific lines or groups of lines are selected in the table with the standard selection gestures provided by the underlying operating system. To select all lines in a table, click on the upper left corner of the table. To un-select all lines in a table, use the 'Deselect' button in the control section.

The 'Color', 'Height', and 'Redshift' controls allow the customization of the plot of the currently selected lines. 'Height' is interpreted as the fractional height on the plot window. 'Redshift' can be interpreted in either 'z' or 'km/s' units, according to the corresponding selector.

5.2.1 List sub-sets

When at least one line is selected in the table, clicking the 'Create set' button causes a new list to be built and displayed in a new tab after the 'Original' tab. The 'Original' tab always contains the entire original list. Successive use of table row selection gestures and the 'Create set' button, allows the creation of multiple sub sets. Sub-sets can be created from the 'Original' list, as well as from any other sub-set.

Each list sub-set carries its own group of display controls: color, height, and redshift. With these, one can customize the appearance of each sub-set on the plot.

The list sub-set capability can be combined with the column sorting and multiple row selection capabilities to implement hierarchical sorting.

Say, as an example, one wants to display all the Fe lines in blue color, and all the high intensity Fe lines (if intensity is provided by the line list) in red. One can sort the original table by species, select the subgroup of Fe lines, and create set #1. Next, on the #1 set, one sorts by intensity, selects all the high intensity lines, and creates set #2. One then de-selects everything in the Original set, and selects everything in sets #1 and #2. Finally, in set #1 one picks the blue color, and in set #2 the red color. Clicking 'Draw' will then plot everything.

5.3 Drawing the line labels

The 'Draw' button accomplishes the actual plotting. It works by finding all the lines that are selected in all tabs at once. Before performing the actual plotting, it will first erase all line labels left on screen by previous drawings. The 'Erase' button performs the same action, with no subsequent drawing.

At the left lower corner of the window, a counter keeps track of the total of lines selected at any time. The counter becomes red as a warning that a large number of lines is selected. Plotting a large number of line labels slows down

the plot and zoom functionalities. The user must be aware that the response may become slow when large numbers of lines are selected.

The drawing operation includes a de-cluttering step. This achieves the dual goal of making the plot more readable, and faster to zoom/pan/rescale. The de-cluttering algorithm trades speed for cleverness, and a side effect of that is that, when plotting sets of lines at different heights on screen, some line labels may disappear even though they shouldn't. Zooming in will eventually make all line labels visible.

Experience has shown that the perceived speed depends to a certain amount on the particular hardware and software platform the application is running on. Typically, a slow laptop can handle a couple of hundred lines with no problem. A faster, multi-core desktop can be pushed up to perhaps a thousand line labels before performance degrades significantly.

5.4 Results

The 'Plotted' tab will always contain the lines currently being displayed on the plot. The contents of this tab can be output to a ECSV file via the Export button on the top menu bar of the line lists window.

The file thus produced can be directly read by SpecViz via de File button (the yellow folder icon).

CHAPTER 6

Custom Loaders

SpecViz utilizes Astropy I/O registry and YAML data serialization language to enable flexible support for a variety of data formats both in FITS and ASCII.

When `specviz` is called with a filename as argument (see *Launching SpecViz*), the default formats below are assumed based on file extension:

- `.fits` or `.mits` – *Generic FITS Loader*
- `.txt` or `.dat` – *ASCII Loader*

By examining the YAML definitions in the following sub-sections and their associated example data files, you will be able to create your own YAML file to define most custom data formats (see *Creating a Custom Loader*). In addition, there are also other custom loaders that come with SpecViz that follow the same rules but are modified to load JWST DADF test data, which you can also use as a reference.

While using YAML is very flexible, it is also very sensitive to slight changes in your file format. For instance, if you have two files from the same instrument but processed differently (say, one was extracted using IRAF and another one using your own IDL program), they might have different formats and will need separate YAML files.

6.1 Generic FITS Loader

```
--- !CustomLoader
name: Generic Fits
extension: [fits, mits]
wcs:
  hdu: 0
data:
  hdu: 1
  col: 0
uncertainty:
  hdu: 1
  col: 1
  type: 'std'
```

(continues on next page)

(continued from previous page)

```
meta:
  author: Nicholas Earl
```

This is generic.yaml, which is a built-in YAML definition for a "generic" FITS file.

```
--- !CustomLoader
```

The first line states that this YAML file defines our custom loader. This is always the same no matter what kind of format you are defining.

```
name: Generic Fits
extension: [fits, mits]
```

These two lines define the format name and accepted extensions, respectively. In SpecViz GUI, this will translate to "Generic Fits (*.fits *.mits)" in the file type drop-down menu.

```
wcs:
  hdu: 0
```

This instructs the loader to look for WCS information in the `PRIMARY` (Extension 0) header. SpecViz also uses this header for other look-ups, e.g., flux unit from `BUNIT` (see below). Therefore, even if there are no WCS information in your file, you always define this block and point the HDU value to the `PRIMARY` header. If WCS information are available and supported by Astropy WCS, they will be used to establish dispersion values and unit.

In the absence of WCS or the presence of explicit dispersion column, an additional `dispersion:` block (not shown) can be defined similarly as `data:` (see below). Unlike flux, `TNULL` masking is ignored. If its column does not have `TUNIT` and `unit: 'unitname'` is defined, SpecViz will fall back to WCS unit. If all these unit look-ups failed, it defaults to null unit. In the presence of `unit:` definition, it overrides both `TUNIT` and WCS.

```
data:
  hdu: 1
  col: 0
```

This instructs the loader to look for flux values (data) in Extension 1, the first column (column index starts from 0). If the column complies to FITS standards (see Astropy FITS Table), flux unit (inferred from `TUNIT`) and data mask (inferred from `TNULL`) are also extracted from the same column.

If `TUNIT` is not defined, loader will look for `unit: 'unitname'` definition within this block, the unit name must be one that is accepted by Astropy Units (case sensitive). If that is undefined as well, flux unit is extracted from `BUNIT` keyword in the same header that contains WCS information (see `wcs:` block for details). If all unit look-ups failed, flux unit is assumed to be erg $^{-1}$ cm^{-2} s^{-1}.

Note that `TNULL` is not the same as DQ arrays, which can be similarly defined with `mask:` block (not shown). If both `TNULL` and DQ are defined, the masks will be combined.

```
uncertainty:
  hdu: 1
  col: 1
  type: 'std'
```

This instructs the loader to look for flux uncertainty values in Extension 1, the second column. Uncertainty type `'std'` states that the values are standard deviation (as opposed to inverse variance, `'ivar'`). Unlike flux data, its `TNULL` masking is ignored and `unit:` tag is not supported. If `TUNIT` is present, loader will attempt to convert the values to flux unit first. Otherwise, its unit is assumed to be the same as flux unit. If inverse variance is given, square-root is applied to the inversed values before being converted to `StdDevUncertainty`.

```
meta:
  author: Nicholas Earl
```

The `meta:` block can contain any metadata tags you wish to include. They do not affect how SpecViz works. In this example, the `author:` tag identifies Nicholas Earl as the origin author of this YAML file.

6.2 ASCII Loader

```
--- !CustomLoader
name: ASCII
extension: [txt, dat]
dispersion:
  col: 0
  unit: 'Angstrom'
data:
  col: 1
  unit: 'erg / (Angstrom cm2 s)'
uncertainty:
  col: 2
  type: 'std'
meta:
  author: STScI
```

This is ascii.yaml, which is a built-in YAML definition for a "generic" ASCII file.

```
--- !CustomLoader
```

The first line states that this YAML file defines our custom loader. This is always the same no matter what kind of format you are defining.

```
name: ASCII
extension: [txt, dat]
```

These two lines define the format name and accepted extensions, respectively. In SpecViz GUI, this will translate to "ASCII (*.txt *.dat)" in the file type drop-down menu. All ASCII files must comply to Astropy ASCII Table standards.

Any header comments with `KEY = VALUE` format will be extracted as header metadata information (currently not used by SpecViz).

```
dispersion:
  col: 0
  unit: 'Angstrom'
```

Unlike *Generic FITS Loader*, ASCII table does not contain WCS. Therefore, the `dispersion:` block is necessary to define the actual dispersion (e.g., wavelength) values. This instructs the loader to look for dispersion values in the first column (column index starts from 0). Its unit, if not defined in the table itself (e.g., via IPAC table format), will be taken from the `unit:` tag. The given unit name must be one that is accepted by Astropy Units (case sensitive). If unit is defined in both table and tag, the latter is ignore. If unit is not defined anywhere, it defaults to Angstrom.

```
data:
  col: 1
  unit: 'erg / (Angstrom cm2 s)'
```

This instructs the loader to look for flux values (data) in the second column. Flux unit handling is similar to dispersion unit (see above), except that the default unit would be erg $^{-1}$ cm^{-2} s^{-1} if undefined.

If there is an associated DQ column, it can be extracted in a similar fashion using a `mask:` block specifying the column index (unit is not applicable). Like *Generic FITS Loader*, zero mask values signify good data.

```
uncertainty:
  col: 2
  type: 'std'
```

This instructs the loader to look for flux uncertainty values in the third column. Uncertainty type `'std'` states that the values are standard deviation (as opposed to inverse variance, `'ivar'`). Its unit must be the same as flux unit. If inverse variance is given, square-root is applied to the inversed values before being converted to `StdDevUncertainty`.

```
meta:
  author: STScI
```

The `meta:` block can contain any metadata tags you wish to include. They do not affect how SpecViz works. In this example, the `author:` tag identifies STScI as the origin author of this YAML file.

6.3 Creating a Custom Loader

In addition to loaders that come pre-packaged with the software, SpecViz also looks for custom loaders that you created and saved in your `~/.specviz` directory, which can be created with the following Unix command:

```
$ mkdir ~/.specviz
```

To create your own loader, you can use either *Generic FITS Loader* or *ASCII Loader* as a template. Your YAML file can have any name of your choosing but must end with a `.yaml` extension.

In this section, we use a MOSFIRE spectrum named spec1d.gds1312_H0.003.emp26177.fits as an example of a custom FITS format for which we must create our own custom YAML definition file from the *Generic FITS Loader* template.

First, we inspect the file format that we have, as follow.

```
>>> from astropy.io import fits
>>> pf = fits.open('spec1d.gds1312_H0.003.emp26177.fits')
>>> pf.info()
Filename: spec1d.gds1312_H0.003.emp26177.fits
No.    Name         Type      Cards   Dimensions   Format
0    PRIMARY     PrimaryHDU       4   ()
1                BinTableHDU     23   1R x 3C      [2287E, 2287E, 2287E]
```

This opens the FITS file and prints out the overall file structure. From this, it is obvious that the table is in Extension 1.

```
>>> pf[0].header
SIMPLE  =                    T /Dummy Created by MWRFITS v1.4a
BITPIX  =                    8 /Dummy primary header created by MWRFITS
NAXIS   =                    0 /No data is associated with this header
EXTEND  =                    T /Extensions may (will!) be present
>>> pf[1].header
XTENSION= 'BINTABLE'           /Binary table written by MWRFITS v1.4a
BITPIX  =                    8 /Required value
...
TFORM3  = '2287E   '           /
```

This prints all the headers and we find no WCS information in either of the extensions.

```
>>> from astropy.table import Table
>>> tab = Table.read(pf[1], format='fits')
>>> print(tab)
FLUX [2287]      LAMBDA [2287]      IVAR [2287]
---------------- ------------------ ---------------
0.0 .. 0.0989667 14500.0 .. 18223.8 1e-06 .. 422.54
```

This shows that there are three columns in the table in Extension 1, namely flux, wavelength, and inverse variance. The table has 2287 rows. Knowing the wavelength regime that the instrument is sensitive to and looking at the wavelength values, we can safely assume that the wavelength unit is Angstrom.

```
>>> from astropy import units as u
>>> u.electron / u.s / u.pix
Unit("electron / (pix s)")
```

However, the flux unit is not defined anywhere and cannot be easily inferred. So, let's just say that we already know the unit to be electrons/s/pix. The code above shows us how Astropy can ingest the flux unit that we want.

```
--- !CustomLoader
name: Keck/MOSFIRE Fits
extension: [fits, mits]
wcs:
  hdu: 0
dispersion:
  hdu: 1
  col: 1
  unit: 'Angstrom'
data:
  hdu: 1
  col: 0
  unit: 'electron / (pix s)'
uncertainty:
  hdu: 1
  col: 2
  type: 'ivar'
meta:
  author: STScI
```

Now that we have the format figured out, it is time to write our own YAML file for it. We will name it keck_mosfire.yaml.

```
--- !CustomLoader
```

The first line states that this YAML file defines our custom loader.

```
name: Keck/MOSFIRE Fits
extension: [fits, mits]
```

These two lines define the format name and accepted extensions, respectively. We will keep the extensions from our FITS template but change the name to identify our new format. In SpecViz GUI, this will translate to "Keck/MOSFIRE Fits (*.fits *.mits)" in the file type drop-down menu.

```
wcs:
  hdu: 0
```

We do not have WCS nor `BUNIT` defined, so we will simply leave this the same as our template.

```
dispersion:
  hdu: 1
  col: 1
  unit: 'Angstrom'
```

This instructs the loader to look for our wavelength values in Extension 1, the second column. We explicitly set its unit to Angstrom.

```
data:
  hdu: 1
  col: 0
  unit: 'electron / (pix s)'
```

This instructs the loader to look for our flux values in Extension 1, the first column, like the template. However, we also explicitly set its unit to electrons/s/pix by providing the appropriate Astropy unit name.

```
uncertainty:
  hdu: 1
  col: 2
  type: 'ivar'
```

This instructs the loader to look for flux uncertainty values in Extension 1, the third column. Unlike the template, we define it as inverse variance.

```
meta:
  author: STScI
```

Since this does not affect how SpecViz works, we do the lazy thing here by leaving it the same as our template.

Once you are done writing your YAML file, be sure to save it in `~/.specviz`. Next, start SpecViz as usual. Now, in its open file dialog, you will see your new format listed in the file-type drop-down menu.

Part IV

References/API

CHAPTER 7

SpecViz API

7.1 Analysis Functions

7.1.1 Functions

centroid(line, avg_cont[, mask])	Compute centroid for the given spectrum.
eq_width(cont1_stats, cont2_stats, line[, mask])	Compute an equivalent width given stats for two continuum regions, and a specutils.core.generic.Spectrum1DRef instance with the extracted spectral line region.
extract(data, x_range)	Extract a region from a spectrum.
fwzi(cont1_stats, cont2_stats, line)	Compute full width at zero intensity (FWZI) for the given spectrum.
resample(data_in, x_in, x_out, y[, ...])	Resample the data of one spectrum using interpolation.
stats(data[, mask])	Compute basic statistics for a spectral region contained in a specutils.core.generic.Spectrum1DRef instance.

centroid

specviz.analysis.**centroid**(*line*, *avg_cont*, *mask=None*)

Compute centroid for the given spectrum.

```
w_cen = integral(wave*flux) / integral(flux)
```

Parameters

data : specutils.core.generic.Spectrum1DRef

Extracted spectrum data.

Returns

wcen : float

Centroid wavelength.

Examples

```
>>> d = Spectrum1DRef(...)
>>> line = extract(d, (15000, 20000))
>>> wcen_em = centroid(line)
```

eq_width

specviz.analysis.**eq_width**(*cont1_stats*, *cont2_stats*, *line*, *mask=None*)

Compute an equivalent width given stats for two continuum regions, and a `specutils.core.generic.Spectrum1DRef` instance with the extracted spectral line region.

This uses for now a very simple continuum subtraction method; i.e., it just subtracts a constant from the line spectrum, where the constant is `(continuum1[mean] + continuum2[mean]) / 2`.

Parameters

cont1_stats, cont2_stats : dict

This is returned by the `stats()` function.

line : `Spectrum1DRefLayer`

This is returned by the `extract()` function.

mask : ndarray

Boolean mask.

Returns

ew, flux, avg_cont : float

Flux and equivalent width values.

Examples

```
>>> d = Spectrum1DRefLayer(...)
>>> cont1 = extract(d, (100, 5000))
>>> cont2 = extract(d, (18000, 20000))
>>> cont1_stats = stats(cont1)
>>> cont2_stats = stats(cont2)
>>> line = extract(d, (15000, 18000))
>>> flux, ew = eq_width(cont1_stats, cont2_stats, line)
```

extract

specviz.analysis.**extract**(*data*, *x_range*)

Extract a region from a spectrum.

Parameters

data : specutils.core.generic.Spectrum1DRef

Contains the spectrum to be extracted.

x_range : tuple

A spectral coordinate range as in `(wave1, wave2)`.

Returns

result : specutils.core.generic.Spectrum1DRef

Spectrum data with extracted region.

Examples

```
>>> d = Spectrum1DRef(...)
>>> d2 = extract(d, (10000, 20000))
```

fwzi

specviz.analysis.**fwzi**(*cont1_stats*, *cont2_stats*, *line*)

Compute full width at zero intensity (FWZI) for the given spectrum. Continuum calculations are similar to `eq_width()`.

Parameters

cont1_stats, cont2_stats : dict

This is returned by the `stats()` function.

line : specutils.core.generic.Spectrum1DRef

This is returned by the `extract()` function.

Returns

fwzi_value : float

FWZI value.

w_range : tuple

Wavelengths used to calculate FWZI.

Examples

```
>>> d = Spectrum1DRef(...)
>>> cont1 = extract(d, (100, 5000))
>>> cont2 = extract(d, (18000, 20000))
>>> cont1_stats = stats(cont1)
>>> cont2_stats = stats(cont2)
>>> line = extract(d, (15000, 18000))
>>> fwzi_value, w_range = fwzi(cont1_stats, cont2_stats, line)
```

resample

specviz.analysis.**resample**(*data_in*, *x_in*, *x_out*, *y*, *data_out=None*, *kind='linear'*)

Resample the data of one spectrum using interpolation.

Dependent variables y1, y2, ... in the input data are resampled in the independent variable x using interpolation models y1(x), y2(x), ... evaluated on a new grid of x values. The independent variable will typically be a wavelength or frequency and the independent variables can be fluxes, inverse variances, etc.

Interpolation is intended for cases where the input and output grids have comparable densities. When neighboring samples are correlated, the resampling process should be essentially lossless. When the output grid is sparser than the input grid, it may be more appropriate to "downsample", i.e., average dependent variables over consecutive ranges of input samples.

The basic usage of this function is:

```
>>> data = np.ones((5,),
... [('wlen', float), ('flux', float), ('ivar', float)])
>>> data['wlen'] = np.arange(4000, 5000, 200)
>>> wlen_out = np.arange(4100, 4700, 200)
>>> resample(data, 'wlen', wlen_out, ('flux', 'ivar'))
array([(4100, 1.0, 1.0), (4300, 1.0, 1.0), (4500, 1.0, 1.0)],
      dtype=[('wlen', '<i8'), ('flux', '<f8'), ('ivar', '<f8')])
```

The input grid can also be third_party to the structured array of spectral data, for example:

```
>>> data = np.ones((5,), [('flux', float), ('ivar', float)])
>>> wlen_in = np.arange(4000, 5000, 200)
>>> wlen_out = np.arange(4100, 4900, 200)
>>> resample(data, wlen_in, wlen_out, ('flux', 'ivar'))
array([(1.0, 1.0), (1.0, 1.0), (1.0, 1.0), (1.0, 1.0)],
      dtype=[('flux', '<f8'), ('ivar', '<f8')])
```

If the output grid extends beyond the input grid, a masked array will be returned with any values requiring extrapolation masked:

```
>>> wlen_out = np.arange(3500, 5500, 500)
>>> resample(data, wlen_in, wlen_out, 'flux')
masked_array(data = [(--,) (1.0,) (1.0,) (--,)],
             mask = [(True,) (False,) (False,) (True,)],
       fill_value = (1e+20,),
            dtype = [('flux', '<f8')])
```

If the input data is masked, any output interpolated values that depend on an input masked value will be masked in the output:

```
>>> data = ma.ones((5,), [('flux', float), ('ivar', float)])
>>> data['flux'][2] = ma.masked
>>> wlen_out = np.arange(4100, 4900, 200)
>>> resample(data, wlen_in, wlen_out, 'flux')
masked_array(data = [(1.0,) (--,) (--,) (1.0,)],
             mask = [(False,) (True,) (True,) (False,)],
       fill_value = (1e+20,),
            dtype = [('flux', '<f8')])
```

Interpolation is performed using `scipy.interpolate.inter1pd`.

Parameters

data_in : numpy.ndarray or numpy.ma.MaskedArray

Structured numpy array of input spectral data to resample. The input array must be one-dimensional.

x_in : string or numpy.ndarray

A field name in data_in containing the independent variable to use for interpolation, or else an array of values with the same shape as the input data.

x_out : numpy.ndarray

An array of values for the independent variable where interpolation models should be evaluated to calculate the output values.

y : string or iterable of strings.

A field name or a list of field names present in the input data that should be resampled by interpolation and included in the output.

data_out : numpy.ndarray or None

Structured numpy array where the output result should be written. If None is specified, then an appropriately sized array will be allocated and returned. Use this method to take control of the memory allocation and, for example, re-use the same array when resampling many spectra.

kind : string or integer

Specify the kind of interpolation models to build using any of the forms allowed by `scipy.interpolate.inter1pd`. If any input dependent values are masked, only the `nearest` and ``linear` values are allowed.

Returns

numpy.ndarray or numpy.ma.MaskedArray

Structured numpy array of the resampled result containing all `y` fields and (if `x_in` is specified as a string) the output `x` field. The output will be a `numpy.ma.MaskedArray` if `x_out` extends beyond `x_in` or if `data_in` is masked.

stats

`specviz.analysis.`**`stats`**(*data*, *mask=None*)

Compute basic statistics for a spectral region contained in a `specutils.core.generic.Spectrum1DRef` instance.

Parameters

data : specutils.core.generic.Spectrum1DRef

Typically this is returned by the `extract()` function.

Returns

statistics : dict

Statistics results.

Examples

```
>>> d = Spectrum1DRef(...)
>>> d_stats = stats(d)
```

7.1.2 Classes

`BlackBody`	Produce a blackbody flux spectrum.
`Spline1D`	Perform a spline fit

BlackBody

class specviz.analysis.**BlackBody**

Bases: astropy.modeling.Fittable1DModel

Produce a blackbody flux spectrum.

Note that the wave and flux arrays used to Quantity

Notes

See Fittable1DModel for further details on modeling and all possible parameters that can be passed in.

Description of the blackbody function itself is described in blackbody

Attributes Summary

norm
param_names
temp

Methods Summary

evaluate(x, temp, norm)	Evaluate the blackbody for a given temperature over a wavelength range.

Attributes Documentation

norm

param_names = ('temp', 'norm')

temp

Methods Documentation

evaluate(*x*, *temp*, *norm*)

Evaluate the blackbody for a given temperature over a wavelength range.

Parameters

x: numpy.ndarray

The wavelengths to evaulate over.

temp: float

The temperature to evualate at.

norm: float

The normalization factor.

Returns
blackbody_flux: numpy.ndarray

The blackbody flux.

Spline1D

class specviz.analysis.**Spline1D**
Bases: astropy.modeling.Fittable1DModel

Perform a spline fit

Notes

See Fittable1DModel for further details.

The spline function is based on UnivariateSpline

Attributes Summary

degree
param_names
smooth

Methods Summary

evaluate(x, degree, smooth)	Evaluate the spline

Attributes Documentation

degree

param_names = ('degree', 'smooth')

smooth

Methods Documentation

evaluate(*x*, *degree*, *smooth*)
Evaluate the spline

Parameters
x: numpy.ndarray

The wavelengths to evaluate over.

degree: int

The degree of spline to evaluate.

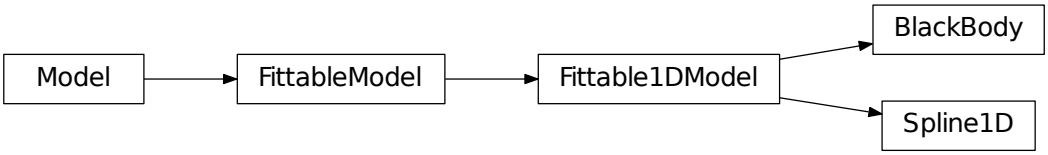
smooth: float or None

The smoothing factor used to choose the number of knots.

Returns

The evaluated spline

7.1.3 Class Inheritance Diagram



7.2 SpecViz core

7.2.1 Data Objects

Data Objects

7.2.2 Classes

Spectrum1DRefLayer(data[, wcs, parent, ...])	Class to handle layers in SpecViz.
Spectrum1DRefModelLayer(data[, model])	A layer for spectrum with a model applied.

Spectrum1DRefLayer

class specviz.core.data.**Spectrum1DRefLayer**(*data*, *wcs=None*, *parent=None*, *layer_mask=None*, *uncertainty=None*, *unit=None*, *mask=None*, **args*, ***kwargs*)

Bases: specutils.core.generic.Spectrum1DRef

Class to handle layers in SpecViz.

Parameters

data: numpy.ndarray

The flux.

wcs: '~astropy.wcs.WCS'

If specified, the WCS relating pixel to wavelength.

parent: layer

If specified, the parent layer.

layer_mask: layer

The layer defining the valid data mask.

args, kwargs:

Arguments passed to the `Spectrum1DRef` object.

Attributes Summary

full_mask	Mask for spectrum data.
layer_mask	Mask applied from an ROI.
masked_data	Flux quantity with mask applied.
masked_dispersion	Dispersion quantity with mask applied.
raw_uncertainty	Flux uncertainty with mask applied.
shape	
unit	
unmasked_data	Flux quantity with no layer mask applied.
unmasked_dispersion	Dispersion quantity with no layer mask applied.
unmasked_raw_uncertainty	Flux uncertainty with mask applied.

Methods Summary

from_formula(formula, layers)	Create a layer from an operation performed on other layers
from_parent(parent[, layer_mask, name, copy])	Create a duplicate child layer from a parent layer
from_self([name, layer_mask])	Create a new, parentless, layer based on this layer
set_units([disp_unit, data_unit])	Set the dispersion and flux units

Attributes Documentation

full_mask
Mask for spectrum data.

layer_mask
Mask applied from an ROI.

masked_data
Flux quantity with mask applied. Returns a masked array containing a Quantity object.

masked_dispersion
Dispersion quantity with mask applied. Returns a masked array containing a Quantity object.

raw_uncertainty
Flux uncertainty with mask applied. Returns a masked array containing a Quantity object.

shape

unit

unmasked_data
Flux quantity with no layer mask applied.

unmasked_dispersion
Dispersion quantity with no layer mask applied.

unmasked_raw_uncertainty
Flux uncertainty with mask applied. Returns a masked array containing a Quantity object.

Methods Documentation

classmethod from_formula(*formula*, *layers*)
Create a layer from an operation performed on other layers

Parameters
formula: str

The operation to perform on the given layers.

layers: [layer, . . .]

The layers which are arguments to the given formula.

Returns
new_layer:

Result of the operation

classmethod from_parent(*parent*, *layer_mask=None*, *name=None*, *copy=True*)
Create a duplicate child layer from a parent layer

Parameters
parent: layer

The layer to duplicate.

layer_mask: layer

The layer defining the valid data mask.

name: str

Layer's name. If `None`, a name based on the parent layer is used.

Returns
child_layer:

The new layer.

from_self(*name=u"*, *layer_mask=None*)
Create a new, parentless, layer based on this layer

Parameters
name: str

Name of the new layer

layer_mask: layer

The layer defining the valid data mask.

Returns
new_layer:

The new, parentless, layer.

set_units(*disp_unit=None*, *data_unit=None*)
Set the dispersion and flux units

Parameters

disp_unit: '~astropy.units'

The dispersion units.

data_unit: '~astropy.units'

The flux units.

Spectrum1DRefModelLayer

class specviz.core.data.**Spectrum1DRefModelLayer**(*data*, *model=None*, **args*, ***kwargs*)

Bases: specviz.core.data.Spectrum1DRefLayer

A layer for spectrum with a model applied.

Parameters

data: numpy.ndarray

The flux.

model: '~astropy.modeling'

The model

args, kwargs:

Arguments passed to the Spectrum1DRef object.

Attributes Summary

model	Spectrum model.
parent_mask	A bitwise combination of the data mask and the Spectrum1DRefModelLayer's parent's layer mask.
unmasked_data	Flux quantity with no layer mask applied.
unmasked_dispersion	Dispersion quantity with no layer mask applied.
unmasked_raw_uncertainty	Flux uncertainty with mask applied.

Methods Summary

from_formula(models, formula)	Create a layer from an operation performed on other models
from_parent(parent[, model, layer_mask, copy])	Create a duplicate child layer from a parent layer

Attributes Documentation

model

Spectrum model.

parent_mask

A bitwise combination of the data mask and the Spectrum1DRefModelLayer's parent's layer mask. This is useful when dealing with slices of spectra in which you want the model layer to be the visible size of the parent *in all cases* (whereas full_mask will always be the selected region)*[]:

unmasked_data

Flux quantity with no layer mask applied. Use the parent layer mask for cases wherein a slice of the spectrum is being used.

`unmasked_dispersion`
Dispersion quantity with no layer mask applied. Use the parent layer mask for cases wherein a slice of the spectrum is being used.

`unmasked_raw_uncertainty`
Flux uncertainty with mask applied. Returns a masked array containing a Quantity object. Use the parent layer mask for cases wherein a slice of the spectrum is being used.

Methods Documentation

`classmethod from_formula`(*models*, *formula*)
Create a layer from an operation performed on other models

Parameters
formula: str

The operation to perform on the given layers.

models: [model, . . .]

The models which are arguments to the given formula.

Returns
result_model:

Result of the operation

`classmethod from_parent`(*parent*, *model=None*, *layer_mask=None*, *copy=False*)
Create a duplicate child layer from a parent layer

Parameters
parent: layer

The layer to duplicate.

model: '~astropy.modeling'

The model.

layer_mask: layer

The layer defining the valid data mask.

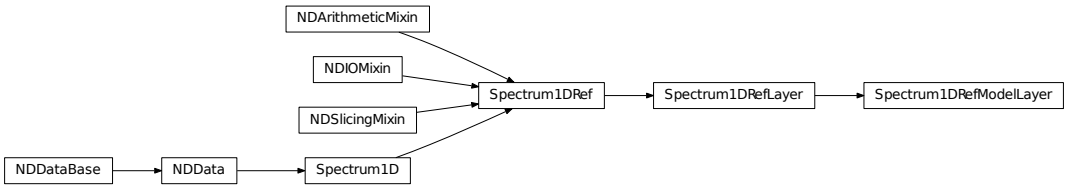
copy : bool

Copy the model if one is provided.

Returns
child_layer:

The new layer.

7.2.3 Class Inheritance Diagram



7.2.4 Object Event Handling

7.2.5 Object Dispatch Machinery

Object Event Handling

The singleton `Dispatch` object manages the set of `EventNode` events. Handlers or **listeners** are attached to `EventNode`s. The `DispatchHandle` decorator is used to decorate classes that handle events.

7.2.6 Classes

`Dispatch`	Central communications object for all events.
`EventNode`(*args)	An event

Dispatch

class specviz.core.dispatch.**Dispatch**

Bases: `object`

Central communications object for all events.

Methods Summary

`register_event`(name[, args])	Add an `EventNode` to the list of possible events
`register_listener`(*args)	Decorate event listeners
`setup`(inst)	Register all methods decorated by `register_listener`
`tear_down`(inst)	Remove all registered methods from their events
`unregister_listener`(name, func)	Remove a listener from an event

Methods Documentation

register_event(*name*, *args=None*)

Add an `EventNode` to the list of possible events

Parameters
name: str

The name of the event.

args: [arg, ...]

The list of keyword arguments this event will pass to its handlers.

register_listener(**args*)
Decorate event listeners

setup(*inst*)
Register all methods decorated by `register_listener`

tear_down(*inst*)
Remove all registered methods from their events

unregister_listener(*name*, *func*)
Remove a listener from an event

Parameters
name: str

The event from wich the listener should be removed.

func: function

The function to be removed

EventNode

class specviz.core.dispatch.**EventNode**(**args*)
Bases: `object`

An event

An event is defined by the arguments the listeners of the event expect to be given.

Parameters
args: [arg, ...]

The list of keyword arguments that the event will provide to its listeners

Methods Summary

`clear`()	Removes all handlers from object.
`emit`(*args, **kwargs)	Call the hanlders of this event

Methods Documentation

clear()
Removes all handlers from object.

emit(**args*, ***kwargs*)
Call the hanlders of this event

Parameters
args: [arg, ...]

The keyword arguments being provided to the event.

kwargs: {arg: value, ...}

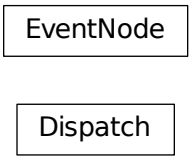
The keyword/value pairs passed to the listeners.

Raises

ValueError

An keyword is being passed that does not belong to this event.

7.2.7 Class Inheritance Diagram



7.2.8 Spectrum Layer Plotting

Spectrum Layer Plotting

7.2.9 Classes

LinePlot(layer[, plot, visible, style, pen, ...])	Plot representation of a layer

LinePlot

class specviz.core.plots.**LinePlot**(*layer*, *plot=None*, *visible=True*, *style=u'line'*, *pen=None*, *err_pen=None*, *mask_pen=None*, *color=(0, 0, 0)*, *line_width=1*)

Bases: object

Plot representation of a layer

Parameters

layer: 'Spectrum1DRefLayer'

The layer to plot

plot: LinePlot

LinePlot instance to reuse.

visible: bool

If True, the plot will be visible

style: str

The plotting style

pen: str

If defined, the pen style to use.

err_pen: str

If defined, the pen style to use for the error/uncertainty.

Attributes Summary

error_pen
layer
mask_pen
pen
plot

Methods Summary

change_units(x[, y, z])	Change the displayed units.
from_layer(layer, **kwargs)	Create a LinePlot from a layer
set_error_visibility([show])	Show the error/uncertainty
set_line_width(width)	Set the line plot width
set_mask_visibility([show])	Show masked data
set_mode(mode)	Set the line plotting mode
set_plot_visibility([show, inactive])	Set visibility and active state
update([autoscale])	Refresh the plot

Attributes Documentation

error_pen

layer

mask_pen

pen

plot

Methods Documentation

change_units(*x*, *y=None*, *z=None*)

Change the displayed units. Note that if an axis is defined as unit- less, providing a new unit will defined that axis as being that unit.

Parameters
x: '~astropy.units'

The new units for the dispersion

y: '~astropy.units'

The new units for the flux

z: '~astropy.units'

The new units for the multi-spectral dimension.

`static from_layer`(*layer*, ***kwargs*)
Create a LinePlot from a layer

Parameters
layer: 'Spectrum1DRefLayer'

The layer to create from.

kwargs: dict

Other arguments for `LinePlot` class.

Returns
plot_container:

The new LinePlot

`set_error_visibility`(*show=None*)
Show the error/uncertainty

Parameters
show: bool

If True, show the error/uncertainty info.

`set_line_width`(*width*)
Set the line plot width

Parameters
width: float

The width of the line

`set_mask_visibility`(*show=None*)
Show masked data

Parameters
show: bool

If True, display data points with mask value True.

`set_mode`(*mode*)
Set the line plotting mode

Parameters
mode: 'line' | 'scatter | 'histogram'

The plot mode

`set_plot_visibility`(*show=None*, *inactive=None*)
Set visibility and active state

Parameters
show: bool

If True, show the plot

inactive: bool

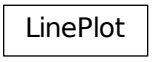
If True, set plot style to indicate this is not the active plot.

update(*autoscale=False*)
Refresh the plot

Parameters
autoscale: bool

If True, rescale the plot to match the data.

7.2.10 Class Inheritance Diagram



7.2.11 Emission/Absorption Line list utilities

Line list utilities

7.2.12 Functions

get_from_file(linelist_path, filename)	
get_from_cache(index)	
ingest(range)	Returns a list with LineList instances.
populate_linelists_cache()	
descriptions()	Returns a python list with strings containing a description of each line list.

get_from_file

specviz.core.linelist.**get_from_file**(*linelist_path*, *filename*)

get_from_cache

specviz.core.linelist.**get_from_cache**(*index*)

ingest

specviz.core.linelist.**ingest**(*range*)
Returns a list with LineList instances.

Each original list is stripped out of lines that lie outside the wavelength range.

Parameters
range:

The wavelength range of interest.

Returns
[LineList, . . .]

The list of linelists found.

populate_linelists_cache

specviz.core.linelist.**populate_linelists_cache**()

descriptions

specviz.core.linelist.**descriptions**()
Returns a python list with strings containing a description of each line list.

Returns
list

The list of strings.

7.2.13 Classes

LineList([table, tooltips, name, masked])	A list of emission/absorption lines

LineList

class specviz.core.linelist.**LineList**(*table=None*, *tooltips=None*, *name=None*, *masked=None*)
Bases: astropy.table.Table

A list of emission/absorption lines

Parameters
table: '~astropy.table.Table'

If specified, a table to initialize from.

name: str

The name of the list.

masked: bool

If true, a masked table is used.

Attributes Summary

table	

Methods Summary

extract_range(wrange)	Builds a LineList instance out of self, with the subset of lines that fall within the wavelength range defined by 'wmin' and 'wmax'.
extract_rows(indices)	Builds a LineList instance out of self, with the subset of lines pointed by 'indices'
merge(lists, target_units)	Executes a 'vstack' of all input lists, and then sorts the result by the wavelength column.
read_list(filename, yaml_loader)	
setColor(color)	
setHeight(height)	
setRedshift(redshift, z_units)	

Attributes Documentation

table

Methods Documentation

extract_range(*wrange*)

Builds a LineList instance out of self, with the subset of lines that fall within the wavelength range defined by 'wmin' and 'wmax'.

REMOVED FOR NOW: The actual range is somewhat wider, to allow for radial velocity and redshift effects. The actual handling of this must wait until we get more detailed specs for the redshift functionality.

Parameters

wrange: (Quantity, Quantity)

minimum and maximum wavelength of the data (spectrum) wavelength range

Returns

LineList

line list with subset of lines

extract_rows(*indices*)

Builds a LineList instance out of self, with the subset of lines pointed by 'indices'

Parameters

indices: [QModelIndex, . . .]

List of QModelIndex instances to extract from.

Returns

LineList

line list with subset of lines

classmethod merge(*lists*, *target_units*)
Executes a 'vstack' of all input lists, and then sorts the result by the wavelength column.

Parameters
lists: [LineList, . . .]
list of LineList instances
target_units: Units
units to which all lines from all tables must be converted to.

Returns
LineList
merged line list

classmethod read_list(*filename*, *yaml_loader*)

setColor(*color*)

setHeight(*height*)

setRedshift(*redshift*, *z_units*)

7.2.14 Class Inheritance Diagram

7.2.15 Thread Helpers

Thread Helpers

7.2.16 Classes

FileLoadThread([parent])	Asynchronously read in a file
LineListLoadThread(linelist, metadata[, parent])	Asynchronously read in a line list.
FitModelThread([parent])	Asynchronously fit a model to a layer

FileLoadThread

class specviz.core.threads.**FileLoadThread**(*parent=None*)
Bases: PyQt5.QtCore.QThread

Asynchronously read in a file

Parameters
parent: QtWidget

The parent widget or None

Attributes

file_name: str	Name of the file to read.
file_filter: str	Type of file to read. If `Auto`, try all known formats.

Attributes Summary

`result`
`status`

Methods Summary

`__call__`(file_name, file_filter)	Initialize the thread
`read_file`(file_name, file_filter)	Convenience method that directly reads a spectrum from a file.
`run`()	Start thread to read the file.

Attributes Documentation

result

status

Methods Documentation

__call__(*file_name*, *file_filter*)
Initialize the thread

read_file(*file_name*, *file_filter*)
Convenience method that directly reads a spectrum from a file.

Parameters
file_name: str

Name of the file to read.

file_filter: str

Type of file to read. If `Auto`, try all known formats.

Returns
data: Spectrum1DRef

The file's data or None if no known formats are found.

Notes

This exists mostly to facilitate development workflow. In time it could be augmented to support fancier features such as wildcards, file lists, mixed file types, and the like. Note that the filter string is hard coded here; its details might depend on the intrincacies of the registries, loaders, and data classes. In other words, this is brittle code.

run()
Start thread to read the file.

LineListLoadThread

class specviz.core.threads.**LineListLoadThread**(*linelist*, *metadata*, *parent=None*)
Bases: specviz.core.threads.FileLoadThread

Asynchronously read in a line list.

Extends functionality in the main file load thread to handle the specifics of a line list.

Parameters
parent: QtWidget

The parent widget or None

Attributes

file_name: str	Name of the file to read.
file_filter: str	Type of file to read. If Auto, try all known formats.

Attributes Summary

result	
status	

Methods Summary

read_file(file_name, file_filter)	Directly reads a line list from a file.

Attributes Documentation

result

status

Methods Documentation

read_file(*file_name*, *file_filter*)
Directly reads a line list from a file.

Parameters
file_name: str

Name of the file to read.

file_filter: str

Type of file to read. If `Auto`, try all known formats.

Returns
data: LineList

The file's data or None if no known formats are found.

FitModelThread

class specviz.core.threads.**FitModelThread**(*parent=None*)
Bases: `PyQt5.QtCore.QThread`

Asynchronously fit a model to a layer

Parameters
parent: QtWidget

The parent widget or None

Attributes

model_layer: Spectrum1DRefLayer	The layer to fit to.
fitter_name: An '~astropy.modeling' fitter	The fitter to use
mask: numpy.ndarray	The mask to apply

Attributes Summary

result	
status	

Methods Summary

__call__(...) <==> x(...)	
fit_model(model_layer, fitter_name[, mask, ...])	Fit the model
run()	Start thread to fit the model

Attributes Documentation

result

status

Methods Documentation

__call__(*...*) <==> *x(...)*

fit_model(*model_layer*, *fitter_name*, *mask=None*, *kwargs=None*)
Fit the model

Parameters
model_layer: Spectrum1DRefLayer

The layer to fit to.

fitter_name: An '~astropy.modeling' fitter

The fitter to use

mask: numpy.ndarray

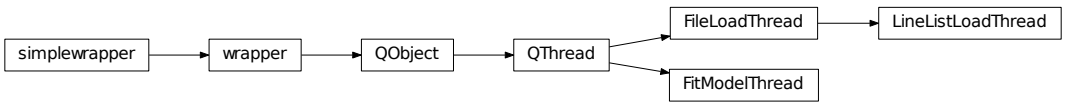
The mask to apply

Returns
(model_layer, fitter_message): Spectrum1DRefLayer, str

The model_layer.model is updated with the fit paramters. The message is from the fitter itself.

run()
Start thread to fit the model

7.2.17 Class Inheritance Diagram



7.3 Interfaces

7.3.1 Functions

data_loader(label, identifier[, priority, ...])	A decorator that registers a function and identifies with an Astropy io registry object.

data_loader

specviz.interfaces.**data_loader**(*label*, *identifier*, *priority=-1*, *extensions=None*, *writer=None*, ***kwargs*)
A decorator that registers a function and identifies with an Astropy io registry object.

Priority will be assigned as an attribute to the wrapped function for use in the auto loader. Loaders with high numerical value will be tried first, followed by low numerical value in order.

Parameters

label : str

user-fiendly name for the data loader

identifier : function

function used to determine if the loader should be used on the Input

priority : int

absolute priority to determine which loader to attempt first

7.3.2 Model Factories

App-wide factories

7.3.3 Classes

Factory	Responsible for creation of objects.
FitterFactory	Create a fitter
ModelFactory	Create a model

Factory

class specviz.interfaces.factories.**Factory**
Bases: object

Responsible for creation of objects.

FitterFactory

class specviz.interfaces.factories.**FitterFactory**
Bases: specviz.interfaces.factories.Factory

Create a fitter

Attributes Summary

all_fitters

Methods Summary

create_fitter(name)	Create a fitter

Attributes Documentation

all_fitters = {u'Levenberg-Marquardt': <class 'astropy.modeling.fitting.LevMarLSQFitter'>, u'Simplex': <

Methods Documentation

classmethod create_fitter(*name*)
Create a fitter

Parameters
name: str

The name of the fitter desired.

Returns
fitter: Fitter

The fitter class requested. None if the requested fitter does not exist.

ModelFactory

class specviz.interfaces.factories.**ModelFactory**
Bases: specviz.interfaces.factories.Factory

Create a model

Notes

Ideally we should be getting these classes from astropy directly and transparently, instead of explicitly naming them here. This is basically a maintenance issue: at each new release of astropy we should check if new models became available, and existing models got deprecated.

This might not be possible unless astropy itself somehow manages to further subclass its Fittable1DModel class into spectrum-specific and other types. Right now we have a mix of spectral models, galaxy surface brightness models, and others, all lumped into a single type. Thus we have to keep picking the spectral relevant types by hand for the time being.

Attributes Summary

all_models

Methods Summary

create_model(name)	Create a model

Attributes Documentation

all_models = {u'BlackBody': <class 'specviz.analysis.models.blackbody.BlackBody'> Name: BlackBody Inputs

Methods Documentation

classmethod create_model(*name*)
Create a model

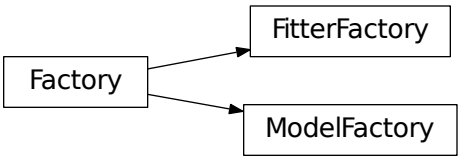
Parameters
name: str

The name of the model desired.

Returns
model: `models`

The requested model. None if the requested model does not exist.

7.3.4 Class Inheritance Diagram



7.3.5 Model Initialization

This module is used to initialize spectral models to the data at hand.

This is used by model-fitting code that has to create spectral model instances with sensible parameter values such that they can be used as first guesses by the fitting algorithms.

7.3.6 Functions

initialize(instance, x, y)	Initialize given model.

initialize

specviz.interfaces.initializers.**initialize**(*instance*, *x*, *y*)
Initialize given model.

X and Y are for now Quantity arrays with the independent and dependent variables. It's assumed X values are

stored in increasing order in the array.

Parameters

instance: '~astropy.modeling.models'

The model to initialize.

x, y: numpy.ndarray

The data to use to initialize from.

Returns

instance: `models`

The initialized model. If there are any errors, the instance is returned uninitialized.

7.4 I/O module

7.4.1 Functions

apStar_identify(*args, **kwargs)	Check whether given filename is FITS.
apStar_loader(*args, **kwargs)	Loader for APOGEE apStar files.
apVisit_identify(*args, **kwargs)	Check whether given filename is FITS.
apVisit_loader(*args, **kwargs)	Loader for APOGEE apVisit files.
aspcapStar_identify(*args, **kwargs)	Check whether given filename is FITS.
aspcapStar_loader(*args, **kwargs)	Loader for APOGEE aspcapStar files.
cos_identify(*args, **kwargs)	Check whether given file contains HST/COS spectral data.
cos_spectrum_loader(*args, **kwargs)	Load file from COS spectral data into a spectrum object
ecsv_identify(*args, **kwargs)	Check if it's an ECSV file.
ecsv_spectrum_loader(*args, **kwargs)	Load spectrum from ECSV file
fits_identify(*args, **kwargs)	Check whether given filename is FITS.
simple_generic_loader(*args, **kwargs)	Basic FITS file loader
simple_generic_writer(data, file_name, **kwargs)	Basic `Spectrum1DRef` FITS writer.
spSpec_identify(*args, **kwargs)	Check whether given filename is FITS.
spSpec_loader(*args, **kwargs)	Loader for SDSS-I/II spSpec files.
spec_identify(*args, **kwargs)	Check whether given filename is FITS.
spec_loader(*args, **kwargs)	Loader for SDSS-III/IV spec files.
stis_identify(*args, **kwargs)	Check whether given file contains HST/STIS spectral data.
stis_spectrum_loader(*args, **kwargs)	Load file from STIS spectral data into a spectrum object

apStar_identify

specviz.io.**apStar_identify**(**args*, ***kwargs*)

Check whether given filename is FITS. This is used for Astropy I/O Registry.

apStar_loader

specviz.io.**apStar_loader**(**args*, ***kwargs*)

Loader for APOGEE apStar files.

Parameters
file_name: str

The path to the FITS file

Returns
data: Spectrum1DRef

The data.

apVisit_identify

specviz.io.**apVisit_identify**(**args*, ***kwargs*)
Check whether given filename is FITS. This is used for Astropy I/O Registry.

apVisit_loader

specviz.io.**apVisit_loader**(**args*, ***kwargs*)
Loader for APOGEE apVisit files.

Parameters
file_name: str

The path to the FITS file

Returns
data: Spectrum1DRef

The data.

aspcapStar_identify

specviz.io.**aspcapStar_identify**(**args*, ***kwargs*)
Check whether given filename is FITS. This is used for Astropy I/O Registry.

aspcapStar_loader

specviz.io.**aspcapStar_loader**(**args*, ***kwargs*)
Loader for APOGEE aspcapStar files.

Parameters
file_name: str

The path to the FITS file

Returns
data: Spectrum1DRef

The data.

cos_identify

specviz.io.**cos_identify**(**args*, ***kwargs*)
Check whether given file contains HST/COS spectral data.

cos_spectrum_loader

specviz.io.**cos_spectrum_loader**(**args*, ***kwargs*)
Load file from COS spectral data into a spectrum object

Parameters
file_name: str

The path to the FITS file

Returns
data: Spectrum1DRef

The data.

ecsv_identify

specviz.io.**ecsv_identify**(**args*, ***kwargs*)
Check if it's an ECSV file.

ecsv_spectrum_loader

specviz.io.**ecsv_spectrum_loader**(**args*, ***kwargs*)
Load spectrum from ECSV file

Parameters
file_name: str

The path to the ECSV file

Returns
data: Spectrum1DRef

The data.

fits_identify

specviz.io.**fits_identify**(**args*, ***kwargs*)
Check whether given filename is FITS. This is used for Astropy I/O Registry.

simple_generic_loader

specviz.io.**simple_generic_loader**(**args*, ***kwargs*)
Basic FITS file loader

Presumption is the primary data is a table with columns 'flux' and 'err'. The dispersion information is encoded in the FITS header keywords.

Parameters
file_name: str

The path to the FITS file

Returns
data: Spectrum1DRef

The data.

simple_generic_writer

specviz.io.**simple_generic_writer**(*data*, *file_name*, ***kwargs*)
Basic Spectrum1DRef FITS writer.

spSpec_identify

specviz.io.**spSpec_identify**(**args*, ***kwargs*)
Check whether given filename is FITS. This is used for Astropy I/O Registry.

spSpec_loader

specviz.io.**spSpec_loader**(**args*, ***kwargs*)
Loader for SDSS-I/II spSpec files.

Parameters
file_name: str

The path to the FITS file

Returns
data: Spectrum1DRef

The data.

spec_identify

specviz.io.**spec_identify**(**args*, ***kwargs*)
Check whether given filename is FITS. This is used for Astropy I/O Registry.

spec_loader

specviz.io.**spec_loader**(**args*, ***kwargs*)
Loader for SDSS-III/IV spec files.

Parameters
file_name: str

The path to the FITS file

Returns
data: Spectrum1DRef

The data.

stis_identify

specviz.io.**stis_identify**(**args*, ***kwargs*)
Check whether given file contains HST/STIS spectral data.

stis_spectrum_loader

specviz.io.**stis_spectrum_loader**(**args*, ***kwargs*)

Load file from STIS spectral data into a spectrum object

Parameters

file_name: str

The path to the FITS file

Returns

data: Spectrum1DRef

The data.

This module contains functions that perform the actual data parsing.

7.4.2 Classes

AsciiYamlRegister(reference)	Defines the generation of Spectrum1DRef objects by parsing ASCII files with information from YAML files.
EcsvYamlRegister(reference)	Defines the generation of Spectrum1DRef objects by parsing ECSV files with information from YAML files.
FitsYamlRegister(reference)	Defines the generation of Spectrum1DRef objects by parsing FITS files with information from YAML files.
LineListYamlRegister(reference)	Defines the generation of LineList objects by parsing line list files with information from YAML files.
YamlRegister(reference)	Class to encapsulate the IO registry information for a set of yaml-loaded attributes.

AsciiYamlRegister

class specviz.io.yaml_loader.**AsciiYamlRegister**(*reference*)

Bases: specviz.io.yaml_loader.YamlRegister

Defines the generation of Spectrum1DRef objects by parsing ASCII files with information from YAML files.

Initialize this particular YamlRegister.

Parameters

reference : dict-like

The yaml reference object created by loading a yaml file.

Methods Summary

get_table(filename)	
reader(filename, **kwargs)	

Methods Documentation

get_table(*filename*)

reader(*filename*, ***kwargs*)

EcsvYamlRegister

class specviz.io.yaml_loader.**EcsvYamlRegister**(*reference*)
Bases: specviz.io.yaml_loader.AsciiYamlRegister

Defines the generation of Spectrum1DRef objects by parsing ECSV files with information from YAML files.

Initialize this particular YamlRegister.

Parameters
reference : dict-like

The yaml reference object created by loading a yaml file.

Methods Summary

get_table(filename)	

Methods Documentation

get_table(*filename*)

FitsYamlRegister

class specviz.io.yaml_loader.**FitsYamlRegister**(*reference*)
Bases: specviz.io.yaml_loader.YamlRegister

Defines the generation of Spectrum1DRef objects by parsing FITS files with information from YAML files.

Initialize this particular YamlRegister.

Parameters
reference : dict-like

The yaml reference object created by loading a yaml file.

Methods Summary

reader(filename, **kwargs)	This generic function will query the loader factory, which has already loaded the YAML configuration files, in an attempt to parse the associated FITS file.

Methods Documentation

reader(*filename*, ***kwargs*)
This generic function will query the loader factory, which has already loaded the YAML configuration files, in an attempt to parse the associated FITS file.

Parameters
filename : str

Input filename.

kwargs : dict

Keywords for Astropy reader.

LineListYamlRegister

class specviz.io.yaml_loader.**LineListYamlRegister**(*reference*)
Bases: specviz.io.yaml_loader.YamlRegister

Defines the generation of LineList objects by parsing line list files with information from YAML files.

Initialize this particular YamlRegister.

Parameters
reference : dict-like

The yaml reference object created by loading a yaml file.

Methods Summary

reader(filename, **kwargs)

Methods Documentation

reader(*filename*, ***kwargs*)

YamlRegister

class specviz.io.yaml_loader.**YamlRegister**(*reference*)
Bases: object

Class to encapsulate the IO registry information for a set of yaml-loaded attributes.

Initialize this particular YamlRegister.

Parameters
reference : dict-like

The yaml reference object created by loading a yaml file.

Methods Summary

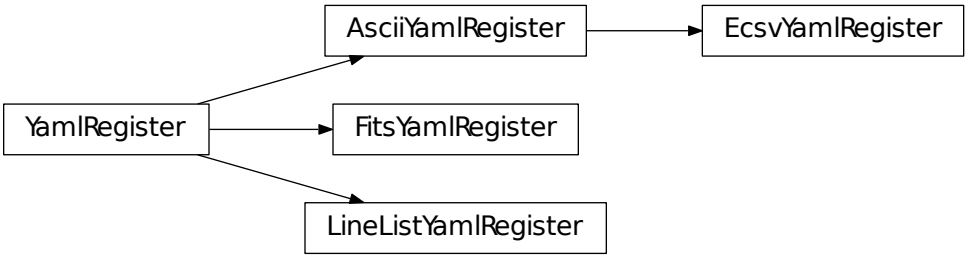
identify(*args, **kwargs)
reader(filename, **kwargs)

Methods Documentation

identify(**args*, ***kwargs*)

reader(*filename*, ***kwargs*)

7.4.3 Class Inheritance Diagram



Part V

Indices and tables

- genindex
- modindex

Python Module Index

S

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

P

R

S

T

U

Y