ExopyPulses Documentation

Release 0.2.0.dev

Exopy team

Aug 13, 2020

CONTENTS

ExopyPulses adds pulse sequences synthesis capabilities

CHAPTER

ONE

USER GUIDE FOR EXOPYPULSES

Exopy high quality graphical user interface should make it quite intuitive to work with. However it can take some time to discover all its capabilities (who wants to search all the menus and tabs !). This guide should help you get started and master the different capabilities of the software.

1.1 Installation

1.1.1 Installing using Conda

The easiest way to install numba and get updates is by using Conda, a cross-platform package manager and software distribution maintained by Continuum Analytics. You can either use Anaconda to get the full stack in one download, or Miniconda which will install the minimum packages needed to get started.

Once you have conda installed, just type:

```
$ conda install -c exopy exopy_pulses
```

or:

```
$ conda update -c exopy exopy_pulses
```

Note: The -c option select the exopy channel on <http://anaconda.org> as Exopy is not part of the standard Python stack.

1.1.2 Installing from source

Exopy_pulses itself is a pure python package and as such is quite easy to install from source, to do so just use :

$ pip install https://github.com/Exopy/exopy_pulses/tarball/master

ExopyPulses adds numpy over Exopy dependencies.

Note: On python 2, you can use the development version of enaml which can be found at <https://github.com/nucleic/enaml>. On python 3 however, you should use the fork located in the Exopy organization <https://github.com/Exopy/enaml> as long as the changes present in that fork have not been merged back into the main repository.

1.1.3 Checking your installation

When starting Exopy you should now be able to select the Pulses workspace to create and edit pulses sequences. The details of the edition, compilation and transfer of pulse sequences is discussed in the next sections.

1.2 Sequence edition

Once the Exopy application started, select the Pulses workspace to create a new sequence. The following section will guide through the required steps and explain the principle of pulse synthesis in Exopy.

Contents

1.2.1 Creating a sequence

When selecting the Pulses workspace, the.measurement edition panels are replaced by three panels :

- one is dedicated to the edition of a sequence
- the second, tabbed beind the first, can be used to edit the variables of the sequence (more on their use will be said later on).
- the thrid one is nothing else than a log panel similar to the one present in the.measurement edition workspace.

Note: Currently the workspace supports only editing a single sequence at a time.

When starting the editor contains a blank sequence. You can either start from there or load an existing sequence using the File menu. From the same menu you can save the sequence (Ctrl+S can also be used).

A pulse sequence is made of two things :

- the sequence of pulses itself which can contain subsequences.
- a compilation context responsible of compiling the pulse sequence and transferring it to a waveform geenrator.

For simplicity sakes, we will first decribe the role of the pulses and sequences but keep in mind that some settings can only be performed once a context is selected.

Note: The edition of the pulse sequence is concentrated in a single scrollable area. To avoid undesired modification of some settings it is advised to only scroll when the mouse is over the scrool bar.

Adding a pulse

When a sequence is empty, it displays two buttons : one dedicated to the addition of a pulse the other to the addition of a sequence. Clicking on the one dedicated to the pulse will immediately insert a new pulse in the sequence and the two previously diplayed buttons will disappear.

Note: Just like when editing the tasks composing a.measurement, the button of the left of each item (pulse/sequence) can be used to add new items, move items or delete items.

The inserted pulse is blank. First you should determine its kind. A pulse can be either **Logical** or **Analogical**. A logical pulse as its name indicates can only take two values 0 or 1. By convention during the pulse duration its take the value 1. By contrast an analogical pulse is arbitrary and hence requires more settings that will be detailed later on.

Once the kind chosen, if you have selected a context, you can then specify on which channel should the pulse be played. Depending on its kind, only the matching channels will be proposed.

Note: The contexts provide the possibility to invert the meaning of a logical channel.

One setting common to all kind of pulse is the timing of the pulse, which can be specified in three different ways : start/stop, start/duration, or duration/stop. In each case, the two required parameters can be specified using formulas with variables in between {}.

The accessible variables, listed by the autocompletion, are the following :

- the global variables declared on the root sequence (discussed later)
- the local variables declared in a subsequence (discussed later)
- the start/stop/duration of any pulse or sequence (in the form i_start, ...) where i is the index of the pulse or sequence. This include the ones of pulses occuring after the edited pulse.

To avoid editing sequences more than necessary, it is advised to avoid hardcoding durations but rather use variables and express the constraints between the pulses using their timing information.

Note: The pulses being indexed rather than named, one should be careful when inserting, moving or deleting a pulse as this will lead to a re-indexing of the pulses.

As previously mentionned, the above information are sufficient to specify a logical pulse but not an analogical one. Analogical pulses need additional parameters provided by their shape and an optional modulation.

The shape of a pulse can be chosen from an existing list of shapes providing different parameters. As it is very common in some experiments to use such pulses to drives mixers, an additional sinusoïdal modulation can be added. In such a case the output pulse will be a sinuoïdal signal whose amplidtude is given by the shape. As can be expected one can specify the frequency and phase of the modulation.

For simple sequences using only pulses is sufficient, however if a more complex/versatile structure is required subsequence can be used. For example, the `ConditionalSequence` can be used to include its child pulses on a given condition making it easy to compile a witness sequence missing the active part.

Adding a sequence

Adding a sequence is very similar to a adding a task to a task hierarchy. When clicking the button, you will be prompted to choose the kind of sequence to add and provide some informations. Once this is done the sequence will be inserted.

Depending on the sequence you will have to adjust specific parameters but some of them are common (and also apply to a point to the root sequence):

- should the sequence have a fixed duration ? This is particularly useful is a specific duty cycle should be respected. When activated you will have to specify the timing of the sequence just like for a pulse and teh context will take care to fill in any necessary blank. Note that the timing infos of a sequence are only accessible in the formula fields if the sequence as a fixed duration.
- the local variables of the sequence. Local variables of a sequence are only visible to its child items and can be used to avoid writing multiple times the same formula.

Your sequence is now ready (up to the selection of the channels) and it is time to select a context.

Choosing a context

The context is responsible for compiling the sequence and transferrring it to the instrument. During compilation, all the formulas are evaluated and the sequence is simplified before being transferred to the instrument.

As a context is linked to a particular instrument, it knwows the available channels.

The context can offer different parametrization, the default ones being :

- the unit used for representing the time in the sequence. By default the unit is the mus.
- the logical channels whose meaning should be inverted (an absence of pulse becoming a logical 1, a presence a 0)

Warning: Currently the context does not support using arbitrary formulas.

Note: Because some optimizations of the compilation may be specific to the waveform generator used the context can work only with specific drivers.

Using the external variables

It is often desirable to parametrize a pulse sequence, for example performing a Rabi oscillation measurement requires to vary duration of the driving pulse. Such a parametrization is possible through the use of the root sequence external variables. Contrary to the local variables, the externals variables will be accessible from the task used to transfer the sequence to the instrument.

These variables can be edited from the second tab of the workspace. Their value will be specified mainly through the transfer task but are needed here too to allow to check that the sequence can be compiled as explained in the next section.

1.2.2 Checking that the sequence compile

Once your sequence written and saved, it may be a good idea to check that it can be compiled. To check it, click on the compile button at the bottom right corner. This will open a dialog,

From this dialog, you can edit the external variables and attempt to compile the sequence. If the compilation succeed the compilation duration will appear in green, if not in red and the compilations errors will be visible in the errors tabs.

Note: At the moment it is not possible to transfer a pulse sequence directly from the pulse sequence edition workspace.

1.3 Sequence transfer

Once you have created a sequence, you are ready to use it in a.measurement. To do so you need to transfer it to the waveform generator. The following section will describe the interaction between the sequence and the transfer task.

1.3.1 Transfer task

The transfer task (`TransferPulseSequenceTask`) allow to load a sequence from a file and to specify the value of the external variables as functions of the task database values. It also allows to parametrize the context.

Note: The context only has access to the sequence variables not to the task tree variables. An easy workaround is to define an external variables matching the task variable.

The chosen instrument must match the selected context. If it is not so, a dialog will notify you that you should choose a new driver.

As it may happen that you edit the sequence after building a.measurement, each time you reopen the.measurement the system will check whether you have loaded the last version of the pulse sequence. If the loaded version is outdated the refresh button will be red. In the same way, during enqueuing a warning will be emitted if the sequence is oudated.

After a successful transfer the task write into the database the informations concerning the sequence returned by the context (this can be the name under which it is stored on the instrument for example).

1.3.2 Sequence re-edition

No matter the care you take in writing the sequence it may well happen that you need to quickly re-edit it while modifying a.measurement. If it is opened in the Pulses workspace you can easily switch and simply reload, however it may prove cumbersome if it is not so.

The easiest way then consist in selecting the transfer task in tree on left of the measure edition panel, and then selecting the pulses editor that appeared in the tabs above the task edition panel. When selecting it, the tree will disappear and you will have access to the sequence. You can then perform any operation you may in the Pulses workspace (note that you can only edit the name of the external variables, as their values should be specified in the task).

Once you are done, go back to the task edition (using the tabs on top). If you have changed of context you may have to choose a new instrument. Or similarly provide the values of any new external variable.

You can also save the sequence and override the old file or save it to a new file.

CHAPTER

TWO

EXTENDING EXOPYPULSES

ExopyPulses instroduce a new plugin dedicated to the handling of pulse sequences. As can be expected one can provide additional shapes, sequences and contexts. The responsability of the context being numerous its case will be treated in its own section.

2.1 Pulse sequences and shapes

Synthesis of pulse sequences in Exopy is centered around the notion of sequence and pulse. Those two elements are used describe the sequence to synthetyse, while the actual synthesis is actually carried out by the context.

The following sections will present how to contribute new shapes for analogical pulses, new sequences and new configuration objects for sequences. The case of context will be treated in detail in another setion.

Contents

2.1.1 Object with evaluable fields

Sequence, pulses, shapes and contexts can all have formula fileds that can be either formatted or formatted and evaluated. In those fields, variables such as the start, stop, and duration of any pulse can be referenced in between curly brackets.

The handling of the evaluation is automated by their common base class `HasEvaluableFields`. All members tagged with either 'fmt' or 'feval' will be automatically formatted/formatted and evaluated when the eval_entries method is called. In case of success, the result will be stored in the *_cache* dictionary of the object. For formatted values, the value of the 'fmt' metadata should be either **True** or **False** depending on whether the formatted value should be stored in the global variables avalaible to other object when evaluating their entries or not. For feval values, the value of the 'feval' metadata should be a `Feval` instance (which is reminiscent of the system used for tasks). `Feval` can be subclassed to customize when to perform the evaluation and how to check the resulting value. By default, a tuple of types can be specified. For values which should be stored as globals, the **store_global** member should be set to **True**.

Note: Shapes and contexts cannot store values in the global namespace.

Note: Even if the name is identical to the objects used for tasks, the Feval object sused in both cases are different. For pulses related object, it should be imported from *exopy_pulses.pulses.api*.

2.1.2 Creating a new shape

Creating a new shape is a three step process :

- first the shape itself which holds the logic must be created.
- to allow a user to correctly parametrize the shape a dedicated widget or view should also be created.
- finally the shape must be declared in the manifest of the plugin contributing it.

Implementing the logic

The shape itself should be a subclass of `AbstractShape`.

The shape parameters should be declared using the appropriate member and tagged with 'pref' in order to be correctly saved. If the default way of saving/restoring (repr/literal_eval) is enough simply use True as a value otherwise you can specify the function to use to serialize/desarialize should be passed as a tuple/list.

```
from numbers import Real

from atom.api import Str, Int
from exopy_pulses.pulses.api import Feval

class MyShape(AbstractShape):
    """MyShape description.

    Use Numpy style docstrings.

    """
    #: my_int description
    my_int = Int(1).tag(pref=True)  # Integer with a default value of 1
```

(continues on next page)

(continued from previous page)

```
    #: my_formula description
    my_formula = Str().tag(pref=True, feval=Feval(types=Real))
```

You will also need to implement one method :

- **compute** : this method should evaluate and return the shape of the pulse at the provided times (taking into account the unit in which the time is expressed). It should return an array. As mentioned before values computed by **eval_entries** can be found in the **_cache** dictionary.

Creating the view

All shape views should inherit from *AbstractShapeView* which is nothing more than a customized SplitItem (Which it should have a single Container child). The view will always have a reference to the shape it is used to edit under *shape* and to the pulse using the shape *item*. From there you are free to design your UI the way you want.

To edit member corresponding to formulas with access to the sequence variables, note that the `QtLineCompleter` and `QtTextCompleter` widgets give auto-completion for the sequence variables after a '{'. You need to set the entries_updater attribute to *item.parent.get_accessible_vars*. If you do so you may also want to use `EVALUATER_TOOLTIP` as a tool tip (tool_tip member) so that your user get a nice explanation about what he can and cannot write in this field. From a general point of view it is a good idea to provide meaningful tool tips.

```
enamldef MyShapeView(AbstractShapeView):

    QtLineCompleter:
        text := shape.my_formula
        entries_updater = item.parent.get_accessible_vars
        tool_tip = EVALUATER_TOOLTIP
```

For more informations about the Enaml syntax please give a look at the relevant section in the Exopy documentation.

At this point your shape is ready to be registered in Exopy, however writing a bunch of unit tests for your shape making sure it works as expected and will go on doing so is good idea. Give a look at *ExopyPulses tests* for more details about writing tests and checking that your tests do cover all the possible cases.

Registering your shape

The last thing you need to do is to declare your shape in a plugin manifest so that the main application can find it. To do so your plugin should contribute an extension to 'exopy.pulses.shapes' providing `Shapes` and/or `Shape` objects.

Let's say we need to declare a single shape named 'MyShape'. The name of our extension package (see the glossary section in Exopy documentation) is named 'my_exopy_plugin'. Let's look at the example below:

```
enamldef MyPluginManifest(PluginManifest):

    id = 'my_plugin_id'

    Extension:
        point = 'exopy.pulses.shapes'

        Shapes:
            path = 'my_exopy_plugin'

            Shape:
```

(continues on next page)

(continued from previous page)

```
                shape = 'my_shape:MyShape'
                view = 'views.my_shape:MyView'
```

We declare a single child for the extension a `Shapes` object. `Shapes` does nothing by themselves they are simply container for grouping shapes declarations. They have a single attribute:

- 'path': when declaring a shape you must specify in which module it is defined as a '.' sperated path. When declaring a path in a `Shapes` it will be prepended to any path-like declaration in all children.

We then declare our shape using a `Shape` object. A `Shape` has three attributes but only two of them must be given non-default values :

- 'shape': this is the path ('.' separated) to the module defining the shape. The actual name of the shape is specified after a colon (':'). As mentioned above the path of all parent `Shapes` is preprended to this path.
- 'view': this identic to the shape attribute but used for the view definition. Once again the path of all parent `Shapes` is preprended to this path.
- 'metadata': Any additional informations about the shape. Those should be specified as a dictionary.

This is it. Now when starting Exopy your new shape should be listed.

2.1.3 Creating a new sequence

Creating a new sequence is very similar to creating a new shape and the same three steps exists :

- first the sequence itself which holds the logic must be created.
- to allow a user to correctly parametrize the sequence one view should also be created.
- finally the sequence must be declared in the manifest of the plugin contributing it.

Minimal methods to implement

The sequence should be a subclass either of `BaseSequence` or `AbstractSequence`. The second case is reserved to cases where a higher level of control over the children item is required which should be quite uncommon.

The declaration of parameters is similar to the one used for shape. If the sequence wish to share a computed value with the other items, it should set the value of the **linkable_vars** member.

```
from numbers import Real

from atom.api import Str, Int
from exopy_pulses.pulses.api import Feval

class MySequence(BaseSequence):
    """MySequence description.

    Use Numpy style docstrings.

    """
    #: my_int description
    my_int = Int(1).tag(pref=True)  # Integer with a default value of 1

    #: my_text description
    my_val = Str().tag(pref=True, feval=Feval(store_global=True))

    linkable_vars = set_default(['my_val'])
```

Sequences should at least implement the following method :

- **simplify_sequence**: this method should a return a list of items that the context declares it can handle as declared in its **supported_sequences** member (pulses are implicitely handled).

Creating the view(s)

Just like for a shape, you need to provide a widget to edit the sequence parameters. The view should subclass either `BaseSequence` or `AbstractSequenceView`. `AbstractSequenceView` is pretty muchh bare, while `BaseSequence` handle the display of the child items, the edition of the local variables and the possibility to fix the duration of the sequence. In both cases, the view is a custom Groupbox which has a reference to the edited sequence and to the root view which itself provide access to the core plugin and several useful methods.

enamldef MySequenceView(BaseSequenceView):

constraints << ([vbox(hbox(t_bool, cond_lab, cond_val),

hbox(*t_def.items), nb)]

if t_def.condition else [vbox(hbox(t_bool, cond_lab, cond_val), nb)])

Label: cond_lab: text = 'Condition'

QtLineCompleter: cond_val: text := item.condition entries_updater = item.get_accessible_vars tool_tip = EVALUATER_TOOLTIP

Registering your sequence

Registering a sequence is quite similar to registering a shape.

Let's say we need to declare a sequence named *MySequence*. The name of our extension package (see the glossary section in Exopy documentation) is 'my_exopy_plugin'. Let's look at the example below:

```
enamldef MyPluginManifest(PluginManifest):

    id = 'my_plugin_id'

    Extension:
        point = 'exopy.pulses.sequences'

        Sequences:
            path = 'my_exopy_plugin'

            Sequence:
                sequence = 'my_sequence:MySequence'
                view = 'views.my_sequence:MyView'
```

We declare a single child for the extension a `Sequences` object. `Sequences` does nothing by themselves they are simply container for grouping sequences declarations. They have a single attribute:

- 'path': when declaring a sequence you must specify in which module it is defined as a '.' sperated path. When declaring a path in a `Sequences` it will be prepended to any path-like declaration in all children.

We then declare our sequence using a `Sequence` object. A `Sequence` has three attributes but only two of them must be given non-default values :

- 'sequence': this is the path ('.' separated) to the module defining the sequence. The actual name of the sequence is specified after a colon (':'). As mentioned above the path of all parent `Sequences` is preprended to this path.
- 'view': this identic to the sequence attribute but used for the view definition. Once again the path of all parent `Sequences` is preprended to this path.
- 'metadata': Any additional informations about the sequence. Those should be specified as a dictionary.

This is it. Now when starting Exopy your new sequence should be listed.

2.1.4 Creating your own sequence filter

As the number of sequences available in Exopy grows, finding the sequence you need might become a bit tedious. To make searching through tasks easier Exopy can filter the sequences from which to choose from. A number a basic filters are built-in but one can easily add more.

To add a new filter you simply need to contribute a `SequenceFilter` to the 'exopy.pulses.filters' extension point, as in the following example :

```
enamldef MyPluginManifest(PluginManifest):

    id = 'my_plugin_id'

    Extension:
        point = 'exopy.pulses.filters'

        SequenceFilter:
            id = 'MySequenceFilter'
            filter_tasks => (sequences, templates):
                return sorted(sequences)[::2]
```

A filter need a unique *id* (basically its name) and a method to filter through sequences. This method receives two dictionaries: the first ones contains the known sequences and their associated infos, the second the templates names and their path. Here we override the *filter_tasks* method, we could also have used one of the following specialized filters:

- `SubclassSequenceFilter`: filter the sequences (exclude the templates) looking for a common subclass (declared in the *subclass* attribute)
- `MetadataSequenceFilter`: filter the sequences (exclude the templates) based on the value of a metadata (*meta_key* is the metadata entry to look for, *meta_value* the value looked for).

2.1.5 Creating your own sequence configurer

In some cases, the default way to configure a sequence before inserting it in a sequence hierarchy (ie simply specifying its name) is not enough. The sequence configurers exist to make possible to customize the creation of a new sequence. Creating one is once again similar to creating a new shape.

Note: Sequence configurers are not meant to fully parametrize a sequence, the sequence view is already there for that purpose. It is rather meant to provide essential informations necessary before including the sequence in a hierarchy or parameters not meant to change afterwards.

Note: When a sequence configurer is specified for a sequence it is by default used form all its subclasses too.

Minimal methods to implement

All sequence configurers need to inherit from `AbstractConfig`, which defines the expected interface of all configurers. When creating a new configurer two methods need to be overwritten :

- build_sequence : this method is supposed to return when called a new instance of the sequence being configured correctly initialized. The configurer holds a refrence to the class of the sequence it is configuring.
- check_parameters : this method should set the *ready* flag to *True* if all the parameters required by the configurer have been provided and *False* otherwise. It should be called each time the value of a parameter change (using a *_post_settattr_** method).

```
class MySequenceConfig(AbstractConfig):
    """Config for MySequence.

    """
    #: My parameter description
    parameter = Int()

    def check_parameters(self):
        """Ensure that parameter is positive and task has a name.

        """
        self.ready = self.parameter

    def build_task(self):
        """Build an instance of MySequence.

        """
        return self.sequence_class(parameter=self.parameter)

    def _post_setattr_parameter(self, old, new):
        """Check parameters each time parameter is updated.

        """
        self.check_parameters()
```

Creating the view

Just like for tasks and interfaces, you need to create a custom widget to allow the user to parametrize the configurer. Your widget should inherit from `AbstractConfigView`. This widget is simple container with a reference to the configurer being edited (**model**)

Declaring the configurer

Finally you must declare the config in a manifest by contributing an extension to the 'exopy.pulses.configs' extension point. This is identical to how shapes are declared but relies on the `SequenceConfigs` (instead of `Shapes`) and `SequenceConfig` (instead of `Shape`) objects. The base sequence class for which the configurer is meant should be returned by the get_sequence_class method.

2.2 Context and pulse sequence compilation

Once a pulse sequence created it need to be uploaded on the waveform generator. The associated compilation (ie translation) and transfer is handled by a context object specific to each instrument (and driver). The following sections will describe in details the responsability of the context and how to implement new ones.

Note: As all instrument specific actions are handled by the driver, the transfer sequence task does not rely on interfaces, implementing the context matching your instrument is sufficient.

Contents

2.2.1 Compilation process

The compilation of a pulse sequence is a three step process :

- first the sequence is evaluated. In this process, all the formulas are evaluated which defines explicitely all the pulses parameters. As pulses can refer to pulses occuring later in the sequence this process can happen in multiple passes till all the formulas can be evaluated (or till a dead end is detected).
- then the sequence is simplified. At this step, all the sequences not explicitely supported by the context are inlined. For example, if the context cannot do anything special with any sequence this leads to the production to a flat list of pulses. A context may choose to special case a specific type of sequence if the hardware provide an effficient support for it, one example could be repeating a subsequence multiple times without explicitely creating a repetitive byte sequence.

- finally the list of simplified items is passed to the context along with the driver of the instrument on which to transfer the sequence. It is then up to the context to turn the list of simple items into an hardware compatible representation and to proceed to the transfer.

2.2.2 Creating a new context

Creating a new context is as usual a three step process :

- first the context itself which holds the logic must be created.
- to allow a user to correctly parametrize the context a dedicated widget or view should also be created.
- finally the context must be declared in the manifest of the plugin contributing it.

Implementing the logic

The context itself should be a subclass of `BaseContext`.

The context parameters should be declared using the appropriate member and tagged with 'pref' in order to be correctly saved. If the default way of saving/restoring (repr/literal_eval) is enough simply use True as a value otherwise you can specify the function to use to serialize/desarialize should be passed as a tuple/list.

```
from numbers import Real

from atom.api import Str, Int
from exopy_pulses.pulses.api import Feval

class MyContext(BaseContext):
    """MyContext description.

    Use Numpy style docstrings.

    """
    #: my_int description
    my_int = Int(1).tag(pref=True)  # Integer with a default value of 1

    #: my_formula description
    my_formula = Str().tag(pref=True, feval=Feval(types=Real))
```

You will also need to implement two methods :

- **compile_and_transfer_sequence**: this method does the heavy work of conversion and transfer. The passed sequence should be considered unevaluated. If the context does not need any control over the evaluation and simplification steps it can simply call the **preprocess_sequence** method to get a list of simplified items. During this step, the entries of the context are evaluated and are afterwards available in the *_cache* member. The returned list of items is guaranteed to be composed only of object the context can handle. Note however that if a context declare it supports a specific sequence, the sequence items are not simplified. If the driver is None, the context should do its best to validate that the sequence can be compiled.

Note: If the sequence declare a fixed duration the context should honor it.

- **list_sequence_infos**: return a dict matching the infos returned are a successful compilation. Those infos can for example contain the names under which the sequence is stored for each channel. The keys should not depend on the sequence.

Creating the view

All context views should inherit from *BaseContextView* which is nothing more than a customized Container. The view will always have a reference to the context it is used to edit under *context*, to the root sequence and to the application core plugin. From there you are free to design your UI the way you want.

To edit member corresponding to formulas with access to the sequence variables, note that the QtLineCompleter and QtTextCompleter widgets give auto-completion for the sequence variables after a '{'. You need to set the entries_updater attribute to *sequence.get_accessible_vars*. If you do so you may also want to use EVALUATER_TOOLTIP as a tool tip (*tool_tip* member) so that your user get a nice explanation about what he can and cannot write in this field. From a general point of view it is a good idea to provide meaningful tool tips.

```
enamldef MyContextView(BaseContextView):

    QtLineCompleter:
        text := context.my_formula
        entries_updater = seuence.get_accessible_vars
        tool_tip = EVALUATER_TOOLTIP
```

For more informations about the Enaml syntax please give a look at the relevant section in the Exopy documentation.

At this point your context is ready to be registered in Exopy, however writing a bunch of unit tests for your context making sure it works as expected and will go on doing so is good idea. Give a look at *ExopyPulses tests* for more details about writing tests and checking that your tests do cover all the possible cases.

Registering your context

The last thing you need to do is to declare your shape in a plugin manifest so that the main application can find it. To do so your plugin should contribute an extension to 'exopy.pulses.contexts' providing Contexts and/or Context objects.

Let's say we need to declare a single context named 'MyContext'. The name of our extension package (see the glossary in Exopy documentation) is named 'my_exopy_plugin'. Let's look at the example below:

```
enamldef MyPluginManifest(PluginManifest):

    id = 'my_plugin_id'

    Extension:
        point = 'exopy.pulses.contexts'

        Contexts:
            path = 'my_exopy_plugin'

            Context:
                context = 'my_context:MyContext'
                view = 'views.my_context:MyContextView'
```

We declare a single child for the extension a Contexts object. Contexts does nothing by themselves they are simply container for grouping shapes declarations. They have a single attribute:

- 'path': when declaring a shape you must specify in which module it is defined as a '.' sperated path. When declaring a path in a Contexts it will be prepended to any path-like declaration in all children.

We then declare our shape using a Context object. A Context has four attributes but only three of them must be given non-default values :

- 'context': this is the path ('.' separated) to the module defining the context. The actual name of the context is specified after a colon (':'). As mentioned above the path of all parent `Contexts` is preprended to this path.
- 'view': this identic to the context attribute but used for the view definition. Once again the path of all parent `Contexts` is preprended to this path.
- 'instruments': list of all the driver ids supported by this context. For memory a driver id is of the form 'definition_package.architecture.class_name'
- 'metadata': Any additional informations about the context. Those should be specified as a dictionary.

This is it. Now when starting Exopy your new context should be listed and work with the specified drivers. Note that just for tasks, you can extend the list of supported drivers by redeclaring a context but specifying only its id (package_name.class_name) and the additionally supported drivers.

2.3 ExopyPulses tests

Test in exopy_pulses uses the same architecture as the ones found in exopy. And just like exopy, exopy_pulses provides some useful fixtures that can be found in the testing package.

Note: Please refer to the Exopy documentation for generic informations about the extension mechanisms.

CHAPTER

THREE

FAQS

CHAPTER

FOUR

API DOCS

Extension package for the Exopy application.

exopy_pulses.**list_manifests**()
List the manifest that should be regsitered when the main Exopy app is started.

4.1 Subpackages

4.1.1 exopy_pulses.pulses package

Extension package for the Exopy application.

Subpackages

exopy_pulses.pulses.configs package

Submodules

exopy_pulses.pulses.configs.base_config module

exopy_pulses.pulses.configs.base_config_views module

Base classes for Sequence configuration views.

class exopy_pulses.pulses.configs.base_config_views.**AbstractConfigView**(*parent=None*, ***kwargs*)
Bases: enaml.widgets.container.Container

Base View for a config from which any other config must inherit

model
A value which allows objects of a given type or types.

Values will be tested using the *PyObject_IsInstance* C API call. This call is equivalent to *isinstance(value, kind)* and all the same rules apply.

The value of an Instance may be set to None.

class exopy_pulses.pulses.configs.base_config_views.**SequenceConfigView**(*parent=None*, ***kwargs*)
Bases: *exopy_pulses.pulses.configs.base_config_views.AbstractConfigView*

Config View of a simple config

exopy_pulses.pulses.configs.template_config module

exopy_pulses.pulses.configs.template_config_view module

Declaration for the view used by TemplateConfig.

class exopy_pulses.pulses.configs.template_config_view.**TemplateConfigView**(*parent=None*, ***kwargs*)

Bases: *exopy_pulses.pulses.configs.base_config_views.AbstractConfigView*

View allowing the user to select how to insert the template.

exopy_pulses.pulses.contexts package

Subpackages

exopy_pulses.pulses.contexts.views package

Submodules

exopy_pulses.pulses.contexts.views.base_context_view module

Base view for all contexts.

class exopy_pulses.pulses.contexts.views.base_context_view.**BaseContextView**(*parent=None*, ***kwargs*)

Bases: enaml.widgets.container.Container

Base class for all Context views.

context

A value which allows objects of a given type or types.

Values will be tested using the *PyObject_IsInstance* C API call. This call is equivalent to *isinstance(value, kind)* and all the same rules apply.

The value of an Instance may be set to None.

core

A value which allows objects of a given type or types.

Values will be tested using the *PyObject_IsInstance* C API call. This call is equivalent to *isinstance(value, kind)* and all the same rules apply.

The value of an Instance may be set to None.

sequence

A value which allows objects of a given type or types.

Values will be tested using the *PyObject_IsInstance* C API call. This call is equivalent to *isinstance(value, kind)* and all the same rules apply.

The value of an Instance may be set to None.

exopy_pulses.pulses.contexts.views.template_context_view module

class exopy_pulses.pulses.contexts.views.template_context_view.**Mapping**(*parent=None*, ***kwargs*)

Bases: enaml.widgets.flow_item.FlowItem

key
A value which allows objects of a given type or types.

Values will be tested using the *PyObject_IsInstance* C API call. This call is equivalent to *isinstance(value, kind)* and all the same rules apply.

The value of an Instance may be set to None.

mapping
A value which allows objects of a given type or types.

Values will be tested using the *PyObject_IsInstance* C API call. This call is equivalent to *isinstance(value, kind)* and all the same rules apply.

The value of an Instance may be set to None.

values
A value which allows objects of a given type or types.

Values will be tested using the *PyObject_IsInstance* C API call. This call is equivalent to *isinstance(value, kind)* and all the same rules apply.

The value of an Instance may be set to None.

class exopy_pulses.pulses.contexts.views.template_context_view.**MappingEditor**(*parent=None*, ***kwargs*)

Bases: enaml.widgets.container.Container

context
A value which allows objects of a given type or types.

Values will be tested using the *PyObject_IsInstance* C API call. This call is equivalent to *isinstance(value, kind)* and all the same rules apply.

The value of an Instance may be set to None.

root
A value which allows objects of a given type or types.

Values will be tested using the *PyObject_IsInstance* C API call. This call is equivalent to *isinstance(value, kind)* and all the same rules apply.

The value of an Instance may be set to None.

class exopy_pulses.pulses.contexts.views.template_context_view.**TemplateContextView**(*parent=No*
***kwargs*)

Bases: *exopy_pulses.pulses.contexts.views.base_context_view.BaseContextView*

View for a template context.

edition_mode
A value which allows objects of a given type or types.

Values will be tested using the *PyObject_IsInstance* C API call. This call is equivalent to *isinstance(value, kind)* and all the same rules apply.

The value of an Instance may be set to None.

Submodules

exopy_pulses.pulses.contexts.base_context module

exopy_pulses.pulses.contexts.template_context module

exopy_pulses.pulses.sequences package

Subpackages

exopy_pulses.pulses.sequences.views package

Submodules

exopy_pulses.pulses.sequences.views.abstract_sequence_view module

exopy_pulses.pulses.sequences.views.base_sequences_views module

exopy_pulses.pulses.sequences.views.conditional_view module

exopy_pulses.pulses.sequences.views.sequence_editor_view module

exopy_pulses.pulses.sequences.views.template_view module

Submodules

exopy_pulses.pulses.sequences.base_sequences module

exopy_pulses.pulses.sequences.conditional_sequence module

exopy_pulses.pulses.sequences.repeat_sequence module

exopy_pulses.pulses.sequences.template_sequence module

exopy_pulses.pulses.shapes package

Subpackages

exopy_pulses.pulses.shapes.views package

Submodules

exopy_pulses.pulses.shapes.views.base_shape_view module

Base classe for shape views.

class exopy_pulses.pulses.shapes.views.base_shape_view.**AbstractShapeView**(*parent=None*, ***kwargs*)

Bases: enaml.widgets.split_item.SplitItem

Abstract class from which all ShapeViews must inherit.

item

A value which allows objects of a given type or types.

Values will be tested using the *PyObject_IsInstance* C API call. This call is equivalent to *isinstance(value, kind)* and all the same rules apply.

The value of an Instance may be set to None.

shape

A value which allows objects of a given type or types.

Values will be tested using the *PyObject_IsInstance* C API call. This call is equivalent to *isinstance(value, kind)* and all the same rules apply.

The value of an Instance may be set to None.

exopy_pulses.pulses.shapes.views.square_shape_view module

exopy_pulses.pulses.shapes.views.modulation_view module

Submodules

exopy_pulses.pulses.shapes.base_shape module

exopy_pulses.pulses.shapes.modulation module

exopy_pulses.pulses.shapes.square_shape module

exopy_pulses.pulses.utils package

Subpackages

exopy_pulses.pulses.utils.widgets package

Submodules

exopy_pulses.pulses.utils.widgets.browsing module

exopy_pulses.pulses.utils.widgets.building module

Submodules

exopy_pulses.pulses.utils.entry_eval module

exopy_pulses.pulses.utils.validators module

exopy_pulses.pulses.utils.normalizers module

Utility functions to build user-friendly names from ids.

Those should be use as formatting function argument to the ids_to_unique_names function found in exopy.utils.transformers

exopy_pulses.pulses.utils.normalizers.**normalize_sequence_name**(*name*)
Normalize sequence names.

For sequences, strip the terminal Sequence if present, replaces '_' by spaces and add spaces between 'aA' sequences. For templates, only the extension file is removed.

exopy_pulses.pulses.utils.normalizers.**normalize_context_name**(*name*)
Normalize context names.

Strip terminal Context and replaces '_' by spaces and add spaces between 'aA' sequences.

exopy_pulses.pulses.utils.normalizers.**normalize_shape_name**(*name*)
Normalize shape names.

Strip terminal Shape and replaces '_' by spaces and add spaces between 'aA' sequences.

exopy_pulses.pulses.utils.sequences_io module

Helper functions for sequences IO.

exopy_pulses.pulses.utils.sequences_io.**load_sequence_prefs**(*path*)
Load the preferences of a sequence stored in a file.

Parameters **path** (unicode) – Location of the template file.

Returns

- **prefs** (ConfigObj) – The data needed to rebuild the tasks.
- **doc** (str) – The doc of the template.

exopy_pulses.pulses.utils.sequences_io.**save_sequence_prefs**(*path*, *prefs*, *doc=''*)
Save a sequence to a file

Parameters

- **path** (unicode) – Path of the file to which save the template
- **prefs** (dict(str : str)) – Dictionnary containing the template parameters
- **doc** (str) – The template doc

exopy_pulses.pulses.workspace package

Submodules

exopy_pulses.pulses.workspace.content module

exopy_pulses.pulses.workspace.dialogs module

exopy_pulses.pulses.workspace.w_manifest module

exopy_pulses.pulses.workspace.workspace module

Submodules

exopy_pulses.pulses.declarations module

exopy_pulses.pulses.dependencies_analysis module

exopy_pulses.pulses.filters module

exopy_pulses.pulses.infos module

exopy_pulses.pulses.item module

exopy_pulses.pulses.manifest module

exopy_pulses.pulses.plugin module

exopy_pulses.pulses.pulse module

exopy_pulses.pulses.pulse_view module

4.1.2 exopy_pulses.tasks package

Extension for the tasks.

Subpackages

exopy_pulses.tasks.tasks package

Subpackages

exopy_pulses.tasks.tasks.instrs package

Subpackages

exopy_pulses.tasks.tasks.instrs.views package

Submodules

exopy_pulses.tasks.tasks.instrs.views.transfer_sequence_task_view module

Submodules

exopy_pulses.tasks.tasks.instrs.transfer_sequence_task module

Submodules

exopy_pulses.tasks.manifest module

4.1.3 exopy_pulses.testing package

Subpackages

exopy_pulses.testing.workspace package

Testing tools for the pulses workspace.

Submodules

exopy_pulses.testing.workspace.fixtures module

Submodules

exopy_pulses.testing.context module

exopy_pulses.testing.context_view module

View for the DummyContext.

class exopy_pulses.testing.context_view.**DummyContextView**(*parent=None*, ***kwargs*)
Bases: *exopy_pulses.pulses.contexts.views.base_context_view.BaseContextView*

View for the base context.

exopy_pulses.testing.fixtures module

4.2 Submodules

4.2.1 exopy_pulses.version module

The version information for this release of Exopy.

exopy_pulses.version.**version_info = version_info(major=0, minor=2, micro=0, status='dev')**
A namedtuple of the version info for the current release.

- *User guide for ExopyPulses*

  How to set up ExopyPulses and use it in your lab.

- *Extending ExopyPulses*

  You need to extend ExopyPulses functionalities, you should definitively start here.

- *FAQS*

  Some questions that might have occurred to others too.

- *API docs*

When all else fails, consult the API docs to find the answer you need. The API docs also include convenient links to the most definitive Exopy documentation: the source.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX