py3o.template Documentation

Release 0.9.10

Florent Aide

Apr 12, 2017

Contents

Contents:

CHAPTER 1

The Python part

Here is an example program that you can find in our source code. It shows how you can use a templated odt and create a final odt with your dataset from python code:

```
from py3o.template import Template

t = Template("py3o_example_template.odt", "py3o_example_output.odt")

t.set_image_path('staticimage.logo', 'images/new_logo.png')


class Item(object):
    pass

items = list()

item1 = Item()
item1.val1 = 'Item1 Value1'
item1.val2 = 'Item1 Value2'
item1.val3 = 'Item1 Value3'
item1.Currency = 'EUR'
item1.Amount = '12345.35'
item1.InvoiceRef = '#1234'
items.append(item1)

for i in xrange(1000):
    item = Item()
    item.val1 = 'Item%s Value1' % i
    item.val2 = 'Item%s Value2' % i
    item.val3 = 'Item%s Value3' % i
    item.Currency = 'EUR'
    item.Amount = '6666.77'
    item.InvoiceRef = 'Reference #%04d' % i
    items.append(item)

document = Item()
document.total = '9999999999999.999'

data = dict(items=items, document=document)
t.render(data)
```

Templating with LibreOffice

If you have read the Python code above you have seen that we pushed a dictionary to our template.render() method.

We must now declare the attributes you want to use from those variables in LibreOffice.

Use control structures

At the moment “for” and “if” controls are available.

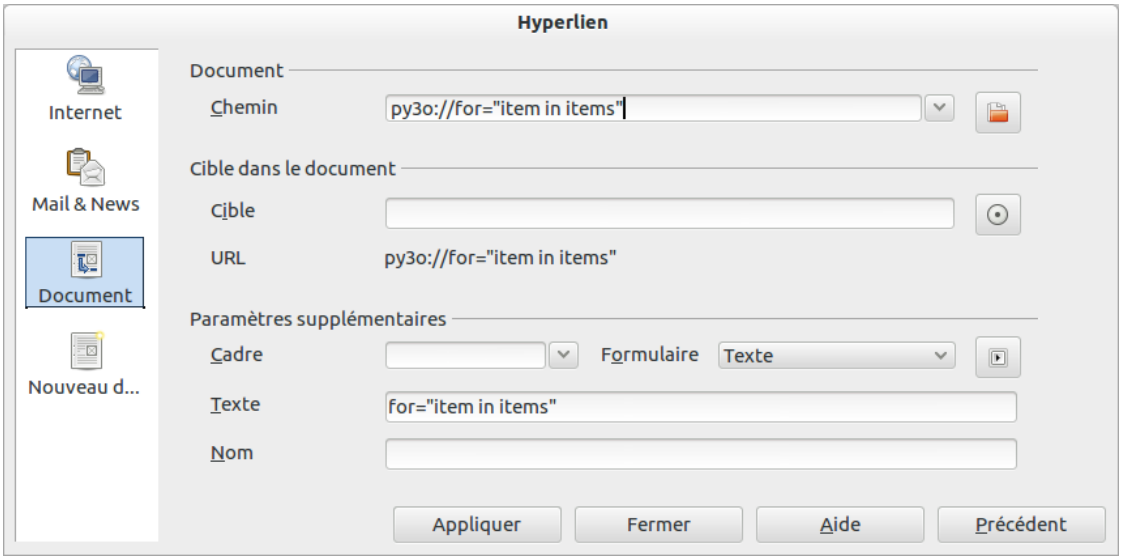
In our example python code we have a dataset that contains a list of items. This list itself is named “items” and we want to iterate on all the items.

We should add a for loop using an hyperlink.

Every control structure must be added to you document using a specially formatted hyperlink:

```
link = py3o://for="item in items"
text = for="item in items"
```

Here is an example setup:



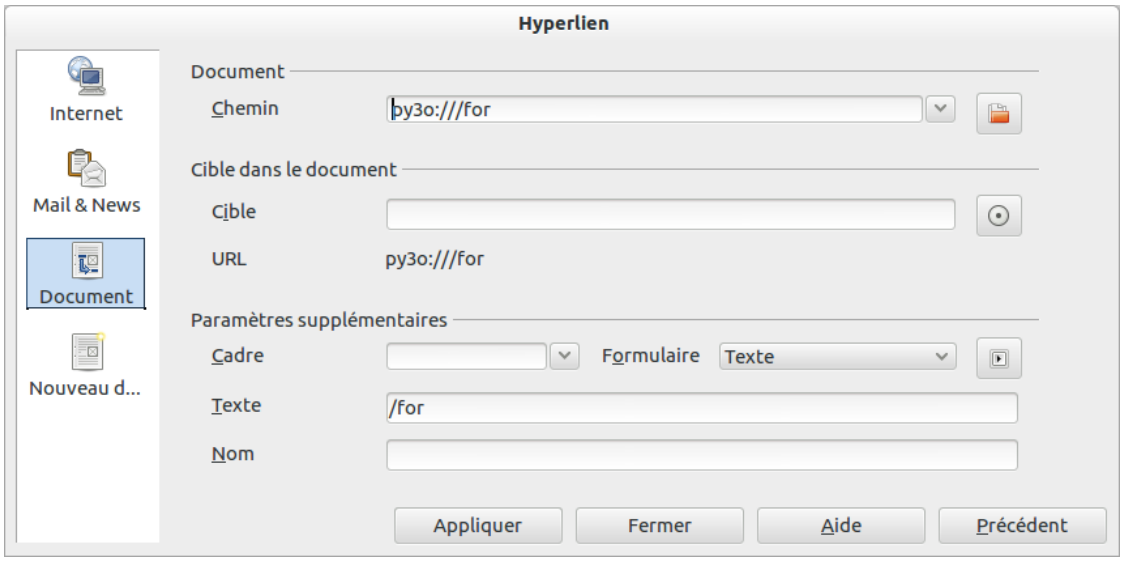
It is especially important to have the link value equivalent to the text value as in the example above.

Once you save your hyperlink, your py3o:// URL will become URL escaped which is fine.

Every control structure must be closed by a corresponding closing tag. In our case we must insert a "/for" hyperlink:

```
link = py3o:///for
text = /for
```

Defined in the user interface as such:



Define variables

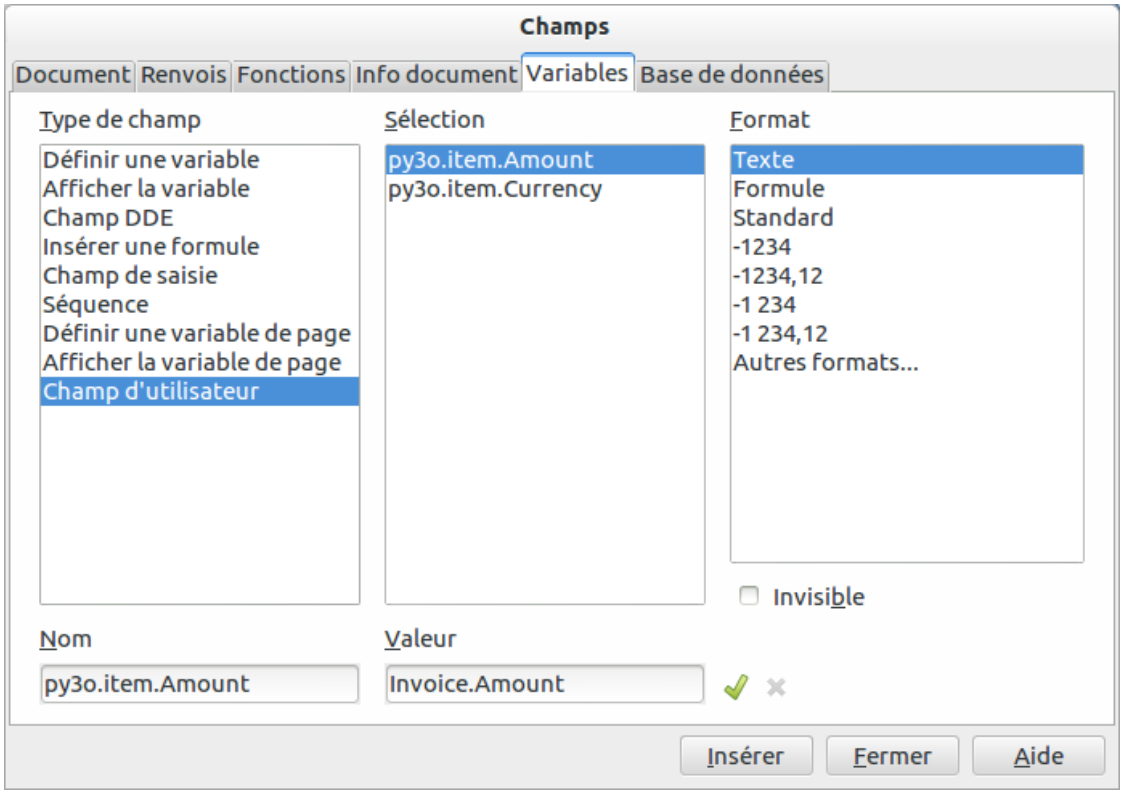
This is done by creating user fields (CTRL-F2) with specific names. The naming scheme is important because it permits differentiate real user fields, which have their own purpose we won't discuss in this document, from the ones we define in our templates.

Since we are inside a for loop that defines a variable names "items" we want to create a user variable in LibreOffice that is named like this:

```
py3o.item.Amount
```

The "Amount" is not something we invent. This is because the item variable is an object coming from your python code. And we defined the Amount attribute back then.

In LibreOffice, user fields can be defined by pressing CTRL-F2 then choosing variables and user-fields:



You must enter a value in name and value then press the green arrow to the right.

the "py3o." prefix is mandatory. Without this prefix the templating system will not be able to find your variables.

The value (in our screenshot: Invoice.Reference) is only some sugar that helps read the template in OpenOffice.

You should take care to pick a nice and meaningfull value so that your end-users know what they will get just by looking at the document without being forced to open the variable definition.

Data Dictionnary

If you are a developper and want to provide some kind of raw document for your users, it is a good idea to create all the relevent user variables yourself. This is what we call in our jargon creating the data dictionary.

This is especially important because the variable names (eg: py3o.variable.attribute) are linked to your code. And remember that your users do not have access to the code.

You should put them in a position where they can easily pick from a list instead of being forced to ask you what are the available variables.

Insert variables

Once you have setup variables and defined some optional control structures you can start inserting variables inside the document.

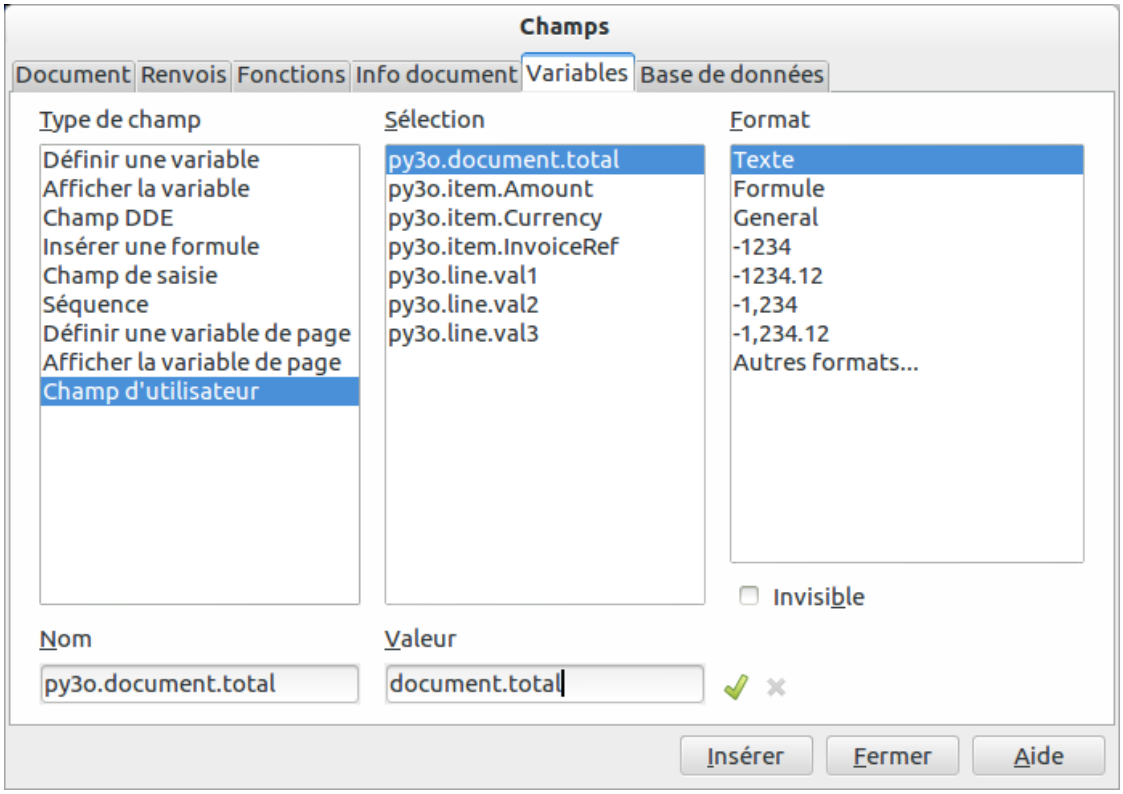
The best way it to use the menu:

```
Insert > Field > Other
```

or just press:

```
CTRL-F2
```

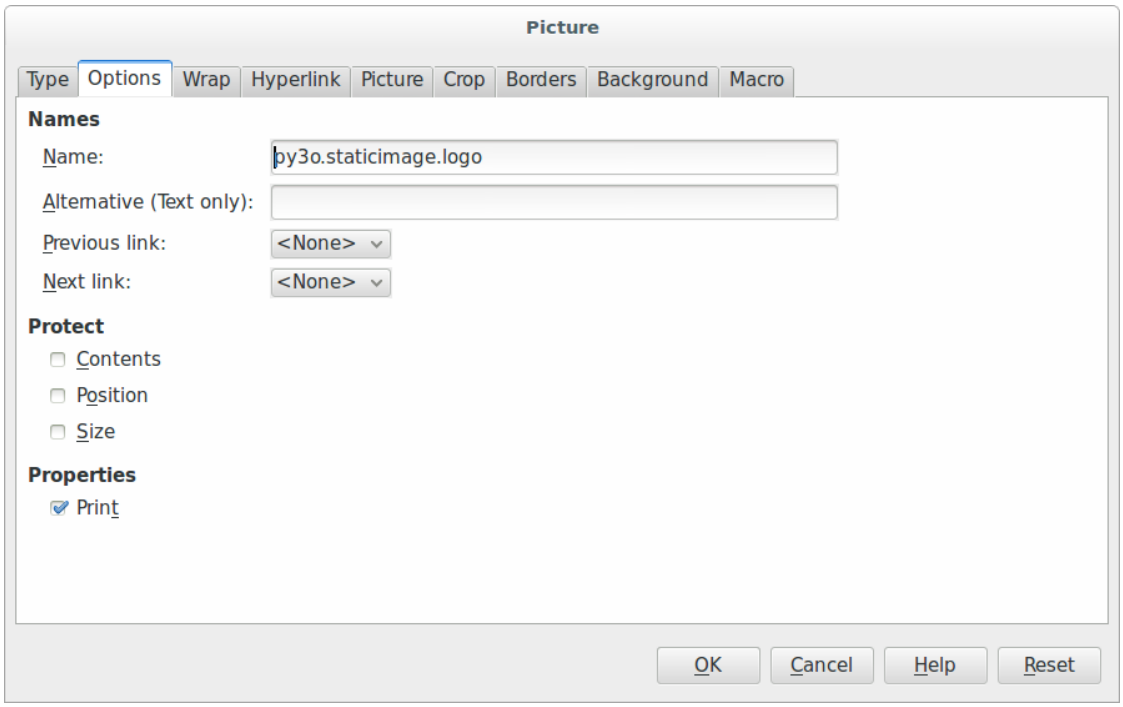
then choose User fields in the field type selection, then choose your desired variable in the second column and then finally click insert at the bottom:



This operation will insert your user field near your cursor. This field will be replaced at template.render() time by the real value coming from the dataset (see above python code)

Insert placeholder images

py3o.template can replace images on-the-fly. To add an image field, add a regular image as a placeholder, open its properties and prefix its name with "py3o.staticimage."; the rest of the image name is then its identifier:

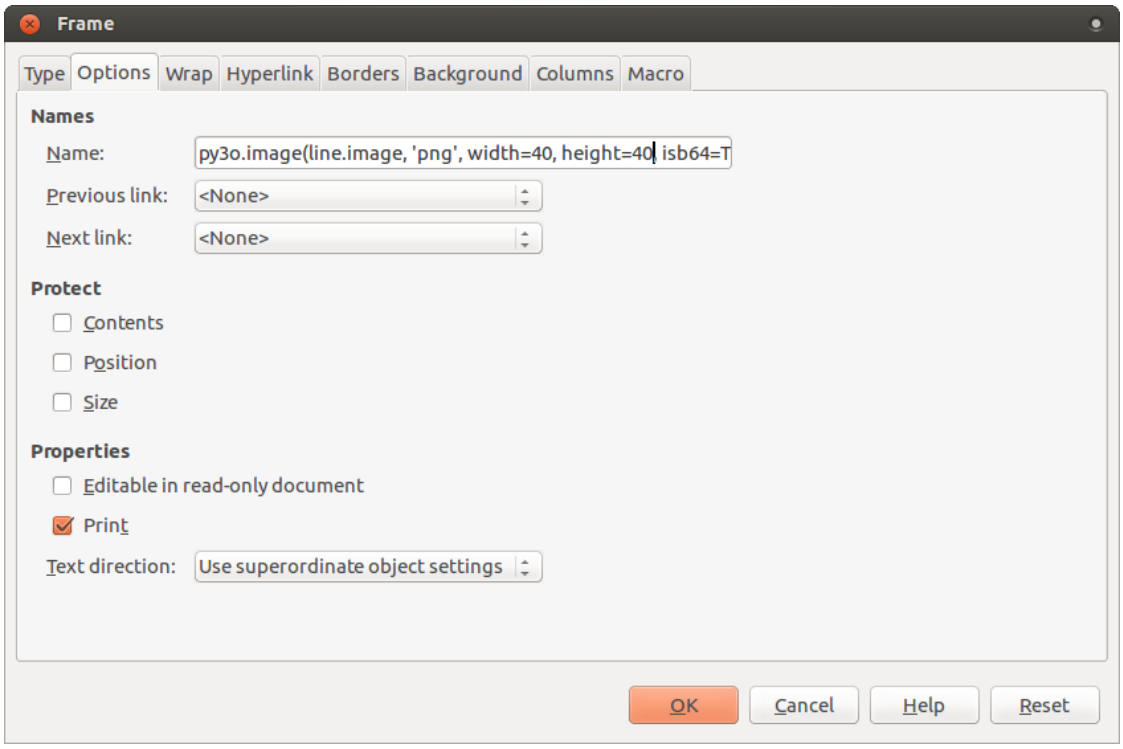


The Python code has to call set_image_path or set_image_data to let py3o know about the image; check our example code:

```
from py3o.template import Template
t = Template("py3o_example_template.odt", "py3o_example_output.odt")
t.set_image_path('staticimage.logo', 'images/new_logo.png')
```

Insert images from the data dictionary

Images can also be injected into the template from the data dictionary. This method should be used in case you have to deal with multiple objects in a for loop, each with their own image. Insert an empty frame as a placeholder (Insert > Frame). Then open its properties and call the py3o.image function in the Name field.



data (required) the variable name for the image in the data dictionary.

mime_type (required) the image's file type.

height (optional) the desired height for the image.

width (optional) The desired width for the image.

isb64 (optional) Whether the image data should be interpreted as base64-encoded bytes instead of raw bytes.

Use format functions

> **Warning:** Format functions are considered to be deprecated. They are meant to be replaced by py3o.types and native ODF formatting capabilities.

Some functions can be called from inside the template in order to format the data. To use a format function, insert a hyperlink as you would to start a loop or condition block:

```
Target:     py3o://function="format_function_name(data, format_arguments)"
Text:       function="format_function_name(data, format_arguments)"
```

Number Formatting

```
format_amount(amount, format)
```

amount A float or Decimal value.

format (optional) The desired number format. See the Python format documentation. Periods in the result are always replace with commas.

Date Formatting

```
format_date(date, format)
```

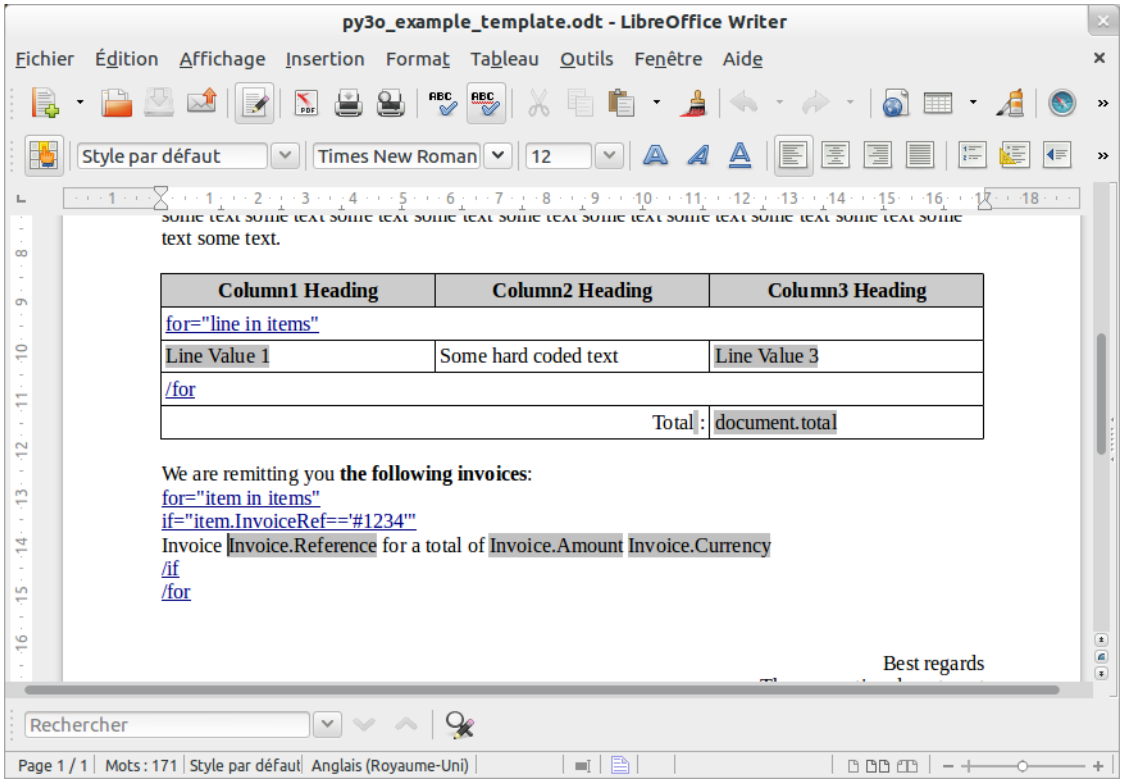
date A date or datetime object, or the ISO-8601 string representation of a date.

format The desired output format

Example document

You can find several example templates (ODT and ODS) in our source tree

Here is a screenshot to show you some control structures (for and if) in action. As you can see you can use these control structures even inside tables:



CHAPTER 3

Source code documentation

Templating

class py3o.template.main.**Template**(*template*, *outfile*, *ignore_undefined_variables=False*, *escape_false=False*)

The default template to be used to output ODF content.

static convert_py3o_to_python_ast(*expressions*)

Convert py3o expressions to parsable Python code.

The py3o expressions can be extracted from a Template object using the *get_all_user_python_expression()* method. The result can then be parsed by Py3oConvertor in order to obtain the data structure expected by the template.

Parameters **expressions** (*list*) – A list of strings that represent expressions in the template.

Returns The expressions in the form of Python code that can be parsed by AST.

Return type str

static find_image_frames(*content_trees*, *namespaces*)

find all frames that must be converted to images

get_all_user_python_expression()

Public method to get all python expression

get_user_instructions()

Public method to help report engine to find all instructions This method will be removed in a future version. Please use Py3oTemplate.get_all_user_python_expression().

get_user_variables()

a public method to help report engines to introspect a template and find what data it needs and how it will be used returns a list of user variable names without the leading 'py3o.' This method will be removed in a future version. Please use Py3oTemplate.get_all_user_python_expression().

handle_draw_frame(*frame*, *py3o_base*)

remove a draw:frame content and inject a draw:image with py:attrs="__py3o_image(image_name)" this __py3o_image method will be injected inside the template dictionary to be called back from inside the template

handle_link(*link*, *py3o_base*, *closing_link*)

transform a py3o link into a proper Genshi statement rebase a py3o link at a proper place in the tree to be ready for Genshi replacement

render(*data*)
render the OpenDocument with the user data

@param data: the input stream of userdata. This should be a dictionary mapping, keys being the values accessible to your report. @type data: dictionary

render_flow(*data*)
render the OpenDocument with the user data

@param data: the input stream of user data. This should be a dictionary mapping, keys being the values accessible to your report. @type data: dictionary

render_tree(*data*)
prepare the flows without saving to file this method has been decoupled from render_flow to allow better unit testing

set_image_data(*identifier*, *data*, *mime_type=None*)
Set data for an image mentioned in the template.

@param identifier: Identifier of the image; refer to the image in the template by setting "py3o.staticimage.[identifier]" as the name of that image. @type identifier: string

@param data: Contents of the image. @type data: binary

set_image_path(*identifier*, *path*)
Set data for an image mentioned in the template.

@param identifier: Identifier of the image; refer to the image in the template by setting "py3o.[identifier]" as the name of that image. @type identifier: string

@param path: Image path on the file system @type path: string

static **validate_link**(*link*, *py3o_base*)
this method will ensure a link is valide or raise a TemplateException

Parameters **link** (*lxml.Element*) – a link node found in the tree

Returns nothing

Raises TemplateException

exception py3o.template.main.**TemplateException**(*message*)
some client code is used to catching ValueErrors, let's keep the old codebase happy

class py3o.template.main.**TextTemplate**(*template*, *outfile*, *encoding='utf-8'*, *ignore_undefined_variables=False*)
A specific template that can be used to output textual content.

It works as the ODT or ODS templates, minus the fact that is does not support images.

render(*data*)
Render the template with the provided data.

Parameters **data** – a dictionnary containing your data (preferably

a iterators) :return: Nothing

py3o.template.main.**detect_keep_boundary**(*start*, *end*, *namespaces*)
a helper to inspect a link and see if we should keep the link boundary

py3o.template.main.**format_amount**(*amount*, *format='%f'*)
Replace the thousands separator from '.' to ','

py3o.template.main.**format_date**(*date*, *format='%Y-%m-%d'*)
Format the date according to format string :param date: datetime.datetime object or ISO formatted string

('%Y-%m-%d' or '%Y-%m-%d %H:%M:%S')

py3o.template.main.**get_all_python_expression**(*content_trees*, *namespaces*)
Return all the python expressions found in the whole document

py3o.template.main.**get_image_frames**(*content_tree*, *namespaces*)
find all draw frames that must be converted to draw:image

py3o.template.main.**get_instructions**(*content_tree*, *namespaces*)
find all text links that have a py3o

py3o.template.main.**get_list_transformer**(*namespaces*)

this function returns a transformer to find all list elements and recompute their xml:id.

Because if we duplicate lists we create invalid XML. Each list must have its own xml:id

This is important if you want to be able to reopen the produced document wih an XML parser. LibreOffice will fix those ids itself silently, but lxml.etree.parse will bork on such duplicated lists

py3o.template.main.**move_siblings**(*start*, *end*, *new_*, *keep_start_boundary=False*, *keep_end_boundary=False*)
a helper function that will replace a start/end node pair by a new containing element, effectively moving all in-between siblings This is particularly helpful to replace for /for loops in tables with the content resulting from the iteration

This function call returns None. The parent xml tree is modified in place

@param start: the starting xml node @type start: lxml.etree.Element

@param end: the ending xml node @type end: lxml.etree.Element

@param **new_**: the new xml element that will replace the start/end pair @type **new_**: lxlm.etree.Element

@param keep_start_boundary: Flag to let the function know if it copies your start tag to the **new_** node or not, Default value is False @type keep_start_boundary: bool

@param keep_end_boundary: Flag to let the function know if it copies your end tag to the **new_** node or not, Default value is False @type keep_end_boundary: bool

@returns: None @raises: ValueError

Data extraction

AST Conversion

class py3o.template.helpers.**Py3oConvertor**
Provide the data extraction functionality.

bind_target(*iterable*, *target*, *context*, *iterated=True*)
Helper function to the For node. This function fill the context according to the iterable and

target and return a new_context to pass through the for body

The new context should contain the for loop declared variable as main key so our children can update their content without knowing where they come from.

Example: python_code = 'for i in list' context = {

'i': Py3oArray({}), '__py3o_module__': Py3oModule({'list': Py3oArray({})}),

}

In the above example, the two Py3oArray are the same instance. So if we later modify the context['i'] Py3oArray,

we also modify the context['__py3o_module__']['list'] one.

static set_last_item(*py3o_obj*, *inst*)

Helper function that take a Py3oObject and set the first leaf found with inst.

This should not be called with a leaf directly.

visit(*node*, *local_context=None*)
Call the node-class specific visit function, and propagate the context

visit_attribute(*node*, *local_context*)
Visit our children and return a Py3oDummy equivalent Example:

i.egg.foo -> Py3oDummy({

'i': Py3oName({ 'egg': Py3oName({'foo': Py3oName()})

}

}

visit_call(*node*, *local_context*)
Visit a function call.

visit_expr(*node*, *local_context*)

An Expr is the way to express the will of printing a variable in a Py3oTemplate. So here we must update the context to map all attribute access.

We only handle attribute access and simple name (i.foo or i)

visit_for(*node*, *local_context*)
Update the context so our chidren have access to the newly declared variable.

visit_module(*node*, *local_context*)
The main node, should be alone. Here we initialize the context and loop for all our children

visit_name(*node*, *local_context*)
Simply return Py3oDummy equivalent

visit_str(*node*, *local_context*)
Do nothing

Data structure

This file contains all the data structures used by Py3oConvertor See the docstring of Py3oConvertor.__call__() for further information

class py3o.template.data_struct.**Py3oArray**
A class representing an iterable value in the data structure. The attribute direct_access will tell if this class should be considered

as a list of dict or a list of values.

render(*data*)
This function will render the datastruct according to the user's data

class py3o.template.data_struct.**Py3oBuiltin**
This class holds information about builtins

classmethod **from_name**(*name=None*)
Return the Py3oObject subclass for the given built-in name Return None if the name does not correspond to a builtin.

Parameters **name** – A Py3oObject instance that represent a name/attribute path

Returns A Py3oObject subclass or None

class py3o.template.data_struct.**Py3oCall**(*name*, *dict*)
This class holds information of function call. 'name' holds the name of function as a Py3oName The keys are the arguments as:

- numeric keys are positional arguments ordered ascendently
- string keys are keywords arguments

class py3o.template.data_struct.**Py3oContainer**(*values*)
Represent a container defined in the template. This container can be: _ A literal list, tuple, set or dict definition _ A tuple of variables that are the target of an unpack assignment

get_tuple()
Return the container's values in a tuple

class py3o.template.data_struct.**Py3oDummy**
This class holds temporary dict, or unused attribute such as counters from enumerate()

class py3o.template.data_struct.**Py3oEnumerate**(*name*, *dict*)
Represent an enumerate call

class py3o.template.data_struct.**Py3oName**
This class holds information of variables. Keys are attributes and values the type of this attribute

(another Py3o class or a simple value)

i.e.: i.egg -> Py3oName({'i': Py3oName({'egg': Py3oName({})})})

render(*data*)
This function will render the datastruct according to the user's data

class py3o.template.data_struct.**Py3oObject**
Base class to be inherited.

get_key()
Return the first key

get_size()
Return the max depth of the object

get_tuple()
Return the value of the Py3oObject as a tuple. As a default behavior, the object returns None.

rget(*other*)
Get the value for the path described by the other Py3oObject.

Recursively checks that the values in other can be found in self.

The method returns the values of self and other at the point where the search stopped. If other is a leaf, the search stops sucessfully. The method returns True, the value that corresponds to the path described by other, and the leaf in question. If other cannot be found in self, the search stops unsuccessfully. The method returns False, the value that corresponds to the deepest point reached in self, and the rest of the path.

Example: self = Py3oObject({

'a': Py3oObject({}), 'b': Py3oObject({

'c': Py3oObject({}),

}),

}) other = Py3oObject({

'b': Py3oObject({ 'd': Py3oObject({}),

}),

}) res = (

False, Py3oObject({'c': Py3oObject({})}), # is self['b'] Py3oObject({'d': Py3oObject({})}), # is other['b']

) if other['b'] was a leaf, res[0] would be True and res[2] the leaf.

Returns A triple: - True if the search was successful, False otherwise - The active sub-element of self when the search stopped - The active sub-element of other when the search stopped

rupdate(*other*)

Update recursively the Py3oObject self with the Py3oObject other. Example: self = Py3oObject({

'a': Py3oObject({}), 'b': Py3oObject({

'c': Py3oObject({}),

}),

}) other = Py3oObject({

'b': Py3oObject({ 'd': Py3oObject({}),

}),

}) res = Py3oObject({

'a': Py3oObject({}), 'b': Py3oObject({

'c': Py3oObject({}), 'd': Py3oObject({}),

}),

})

p

B

C

D

F

G

H

M

P

R

S

T

V