oosheet Documentation

Release 1.3

Luis Fagundes

Apr 04, 2018

Contents

OOSheet is a Python module for manipulating LibreOffice spreadsheet documents and creating macros.

Using Python, you interact with an LibreOffice instance to develop and test your code. When you're finished you can insert your python script inside the document to run it as macro, if this is what you desire.

OOSheet API was inspired by JQuery. It uses selectors in same way that you would use in LibreOffice, with cascading method calls for quick development. Part of the API was also inspired by Django's object-relational mapper.

CHAPTER 1

Why OOSheet?

When you use a lot of spreadsheets and have some programming skills, writing scripts to make your life easier are something that will come to your mind. If you start to look for your options with LibreOffice, you'll see that OO.org Basic is ugly and weird, while Python support is powerful but very tricky. And finally, your routines are likely to be coupled to the structure of your spreadsheet, so the best place to have your macros would be inside your documents, but OO.org provides no way for you to do that.

If you see yourself in need of automating routines in a spreadsheet and like Python, OOSheet is surely for you.

If your situation is not really like this but you're considering using PyUno for anything, it's very likely that you'll find OOSheet useful in some way, even if your document is not a spreadsheet. The base class OODoc may be a good general wrapper for PyUno, and OOPacker class can be used to insert your python script in any OO.org document.

For using OOSheet you need a running instance of LibreOffice. If you just want to generate a document, for example, as a result of a web system in which user will download some automatically generated spreadsheet, then this module is probably not what you're looking for. It could be used though, if you're willing to manage a running LibreOffice process.

CHAPTER 2

Download / Install

Just type:

```
$ pip3 install oosheet
```

You can get the pip command here.

You'll need git and python uno. If you use a Debian-based GNU/Linux distribution (like Ubuntu), you can do this with:

```
$ sudo aptitude install python3-uno
```

OOSheet was developed and tested on Python 3.5 and LibreOffice 5.1. It should work in other versions, though. If you try it in other environments, please report results to author.

CHAPTER 3

Source

The OOSheet source can be downloaded as a tar.gz file from http://pypi.python.org/pypi/oosheet

Using git you can clone the source from http://github.com/lfagundes/oosheet.git

OOSheet is free and open for usage under the MIT license.

CHAPTER 4

Documentation

Contents:

4.1 Using OOSheet

4.1.1 Developing with OOSheet

There are two ways of using OOSheet. You can either make a python script that manipulates an instance of LibreOffice, or you can make python macros that will be executed directly by OO. Macros can be installed in a global path for all users and documents, in user's home directory for a single user or inside a document.

No matter what your choice is, the python code is the same and works in any of those environments. So, it's always best to start your development by manipulating an instance of Openoffice.org, so that you don't have to restart it to test your routines.

You must launch LibreOffice Spreadsheet allowing socket connections. To do so, use the following command line (in GNU/Linux):

```
$ libreoffice --calc --accept="socket,host=localhost,port=2002;urp;StarOffice.
↪ServiceManager"
```

Since this command is very complicated to remember, a reminder command is included:

```
$ oosheet-launch
```

4.2 OOSheet with Spreadsheets

4.2.1 Hello world

After launching libreoffice, in a python shell:

```
>>> from oosheet import OOSheet as S
>>> S('a1').string = 'Hello world'
```

4.2.2 Cells selectors

OOSheet uses selectors to determine which cells you are working with. Each instance of OOSheet class receives a selector as parameter.

Examples of possible selectors:

```
>>> S('b2') # one single cell at first sheet
>>> S('a1:10') # one column from a1 to A10 at first sheet
>>> S('a1:a10') # same thing
>>> S('a1:g1') # one row with six columns
>>> S('a1:g7') # a 7x7 square with 49 cells
>>> S('Sheet2.a2:3') # cells A2 and A3 in sheet named "Sheet2"
```

and so on. Finally, if no selector is passed, OOSheet object will be initialized with user's selection:

```
>>> S() # gets the user selection
```

4.2.3 Data manipulation

Each cell has three relevant attributes concerning its data: value, string and formula. Besides that, OOSheet adds the "date" attribute, that is a wrapper around value to work with datetime.datetime objects and properly format date cells. Setting one of these 4 attributes will modify all of them accordinly.

Value is always a float and is not seen by the user. String is the representation you see inside the cell. Formula is the one you see at the formula box on top of the spreadsheet and generates a value and string. Dates are represented as values internally, counting the number of days since 30/12/1899, but that is transparent to developers using OOSheet.

The following code illustrates how to deal with those types:

```
>>> S('a1').value = 1
>>> S('a1').value
1.0
>>> S('a1').string
u'1'
>>> S('a1').formula
u'1'
>>> S('a1').date
datetime.datetime(1899, 12, 31, 0, 0)
```

```
>>> S('a1').string = 'Hello world'
>>> S('a1').value
0.0
>>> S('a1').formula
u'Hello world'
>>> S('a1').date
datetime.datetime(1899, 12, 30, 0, 0)
```

```
>>> S('a1').value = 1
>>> S('a2').formula = '=a1+10'
>>> S('a2').value
```

```
11.0
>>> S('a2').string
u'11'
>>> S('a2').formula
u'=A1+10'
>>> S('a2').date
datetime.datetime(1900, 1, 10, 0, 0)
```

```
>>> S('a1').date = datetime.datetime(2011, 01, 19)
>>> S('a1').value
40562.0
>>> S('a1').string
u'01/19/2011'
>>> S('a1').formula
u'40562'
```

Alternatively, you can use set_value(), set_string(), set_formula() and set_date() methods:

```
>>> S('a1').set_value(1)
>>> S('a2').set_string('Hello')
and so on
```

This is useful for cascading calls.

You can use flatten() to consolidate values and strings generated by formulas to be independent of the formulas:

```
>>> S('a1').value = 1
>>> S('a2').formula = '=a1+3'
>>> S('a2').formula
u'=A1+3'
>>> S('a2').flatten()
>>> S('a2').formula
u'4'
>>> S('a2').value
4.0
```

It's also possible to access value of cells as a 2d-tuple:

```
>>> S('a1').set_value(2).drag_to('a3').drag_to('b3')
>>> S('a1:b3').data_array
((2.0, 3.0), (3.0, 4.0), (4.0, 5.0))
```

4.2.4 Acessing Cells

A subgroup of cells can be acessed as arrays:

```
>>> S('a1:g10')[4]
Sheet1.A5:G5
>>> S('a1:g10')[4][2]
Sheet1.C5
>>> S('a1:g10')[4]['F']
Sheet1.F5
>>> S('a1:g10')[1:2][3:4]
Sheet1.D2:E3
>>> S('a1:g10')[1:2]['B':'F']
Sheet1.B2:F3
```

As you see, you can access columns either by index number or string. By default, if you first access rows, then columns, but if you access columns using strings, the order does not matter:

```
>>> S('a1:g10')[1]['F']
Sheet1.F2
>>> S('a1:g10')['F'][1]
Sheet1.F2
```

Selections can also be iterated:

```
>>> for cell in S('Sheet1.a1:b10'):
>>>     print str(cell) # will print something like Sheet1.A3,
```

the example above will iterate over 20 cells, in each iteration cell will hold an OOSheet object with one cell. Exactly the same thing can be obtained with:

```
>>> for cell in S('Sheet1.a1:b10').cells:
>>>     print str(cell)
```

You can also iterate over rows or columns:

```
>>> for row in S('Sheet1.a1:b10').rows:
>>>     print "This loop will be iterated 10 times"
```

```
>>> for col in S('Sheet1.a1:b10').columns:
>>>     print "This loop will be iterated twice"
```

4.2.5 Finding Cells

A selection can be searched for cells matching some criteria:

```
>>> S('a1:g10').find(lambda cell: cell.string.startswith(u'...'))
```

The find() method returns an iterator:

```
>>> for cell in S('a1:g10').find(u'word'):
>>>    # do something with cell
```

You can also pass a string, integer or float as parameter. Internally, it will be converted to a lambda function depending on type:

```
>>> S('a1:g10').find(u'word') # same as find(lambda cell: cell.string == u'word')
>>> S('a1:g10').find(17)      # same as find(lambda cell: cell.value == 17)
```

4.2.6 Simulating user events

Several user events can be simulated: dragging, inserting and deleting rows, cutting and pasting, formatting, undo and redo, saving and quitting.

Dragging does an autofill, as when you drag that little square in the bottom right corner of you selection:

```
>>> S('a1').value = 1
>>> S('a1').drag_to('a10')
>>> S('a1:a10').drag_to('g10')
```

Rows can be inserted and deleted. Note that when you insert rows or columns, the selection of the object will grow to include the cells just inserted:

```
>>> S('a4').insert_row() #insert one row
Sheet1.A4:A5
>>> S('a4').insert_rows(7) #inserts seven rows
Sheet1.A4:A11
>>> S('d1').insert_column()
Sheet1.D1:E1
>>> S('a7').delete_rows()
>>> S('g1').delete_columns()
```

Cut & paste:

```
>>> S('a8:b8').cut()
>>> S('a1:4').copy()
>>> S('j5').paste()
```

The format of a cell can be used to format another cell. Internally, this is done with a "paste special" that copies data from other cell and pastes the format on the current selection:

```
>>> S('j4').format_as('a2')
(you won't see anything, unless you have previously formatted a2 manually. Try␣
↪setting its background first)
```

Undo, redo, save_as and quit:

```
>>> S().undo()
>>> S().redo()
>>> S().save_as('/tmp/oosheet_sandbox.ods')
>>> S().quit() # this will close LibreOffice
```

Any LibreOffice event can be generated, not only the ones above. See *Recording macros* for instructions on how to discover events.

4.2.7 Cascading calls

Most methods can be cascaded. For example:

```
>>> S('a1').set_value(1).drag_to('a10').drag_to('g10')
```

This is because these methods returns OOSheet objects. Note that the selection is not necessarily preserved, sometimes it is modified. In the above example, set_value() does not change the selection, but drag_to('a10') expands the selection to 'a1:a10', so the whole column is dragged to G10.

The cascading logic is so that the resulting selection should always be as you expect.

4.2.8 Moving, growing and shrinking selections

Sometimes you don't know exactly where your group of cells is, but know its position relative to a selector you have. In this situation, the selection modificators are handful. With them, you can move, grow or shrink a selection.

Selectors can be moved. For example:

```
>>> S('sheet1.a1:a10').shift_right()
Sheet1.B1:B10
```

The result is an OOSheet object with selector Sheet1.B1:B10. The shift_* methods are useful for cascading calls:

```
>>> S('a1').set_value(1).drag_to('a10').drag_to('g10') #just to setup
>>> S('c1:c10').insert_column().shift_right(2).copy().shift_left(3).paste()
```

It's also possible to shift a selector up and down:

```
>>> S('a1').shift_down(2)
Sheet1.A3
>>> S('a3:c5').shift_up()
Sheet1.A2:C4
```

You can also shift the selector until a condition is satisfied. The shift_DIRECTION_until() methods are used for this:

```
>>> S('f1').value = 15
>>> S('a1').shift_right_until(15)
Sheet1.F1
```

The above example will only work for single cell selectors. For other selectors, you have to specify where to look for a value:

```
>>> S('g5').string = 'total'
>>> S('a1:10).shift_right_until(row_5 = 'total')
Sheet1.G1:G10
>>> S('a1:z1').shift_down_until(column_g = 'total')
Sheet.A5:Z5
```

(Note that only one parameter is accepted)

For more complex conditions, you can use lambda functions:

```
>>> S('g5').string = 'hello world'
>>> S('a1:10').shift_down_until(column_g_satisfies = lambda s: s.string.endswith(
↪'world'))
Sheet1.G1:G10
```

The "s" parameter in lambda function will be a 1 cell OOSheet object.

When looking for cells, you must specify a column if you're shifting up or down, and a row if right or left. If you specify a column, the row considered will be the last one if you're going down and the first one if you're going up, and vice-versa.

Selectors can also be expanded or reduced:

```
>>> S('a1:10').grow_right()
Sheet1.A1:B10
>>> S('a1:g1').grow_down(2)
Sheet1.A1:G3
>>> S('c3:d4').grow_left()
Sheet1.B3:D4
>>> S('a1:g10').shrink_down()
Sheet1.A1:G9
>>> S('a1:g10').shrink_left()
Sheet1.B1:G10
```

There are also grow_DIRECTION_until() and shrink_DIRECTION_until() methods, that works similar to shift_until conditions:

```
>>> S('a1').set_value(1).drag_to('a10').drag_to('g10') #setup
```

```
>>> S('a1:b2').grow_right_until(row_2 = 6)
Sheet1.A1:E2
>>> S('a1:e2').shrink_right_until(row_1 = 3)
Sheet1.A1:C2
>>> S('a1:b2').grow_down_until(column_c_satisfies = lambda s: s.value > 10)
Sheet1.A1:B9
>>> S('a1:b9').shrink_down_until(column_c_satisfies = lambda s: s.value < 5)
Sheet1.A1:B2
```

(Note that the reverse of grow_up is shrink_up and not shrink_down. Authors are not sure which way would be best, but currently shrink_down will remove lines from bottom resulting in an upward moving sensation.)

Moving selections can also be done by arithmetical operations. You can add or subtract tupples of (column, row) to make a shift:

```
>>> S('a1')
Caixa.A1
>>> S('a1')
Sheet1.A1
>>> S('a1') + (1, 0)
Sheet1.B1
>>> S('a1') + (0, 1)
Sheet1.A2
>>> S('a1') + (2, 3)
Sheet1.C4
>>> S('b5:d7') - (1, 2)
Sheet1.A3:C5
```

Subtraction can also be used to calculate the shift between two selections. This may be useful after you do a shift_until:

```
>>> S('b5:d7') - S('a1:c3')
(1, 4)
>>> total_row = S('a1:c10').shift_down_until(col_b = 'Total: ')
>>> cols, rows = total_row - S('a1:c10')
```

4.2.9 Getting the borders

After shift, grow and shrink operations you may need to get the first or last row or column or your selection. This can be done with first_row, last_row, first_column and last_column properties:

```
>>> S('a1:g10').first_row
Sheet1.A1:G1
>>> S('a1:g10').last_row
Sheet1.A10:G10
>>> S('a1:g10').first_column
Sheet1.A1:A10
>>> S('a1:g10').last_column
Sheet1.G1:G10
```

4.2.10 Cells protection

Sheets and cells can be protected and unprotected against editions. When sheet is protected, only unprotected cells can be edited, while if sheet is unprotected, all cells can be modified no matter its protection. Sheet can be protected

with a password, so that same password is required to unprotect it.

NOTE: Depending on LibreOffice version, protected cells can be edited by scripts by changing value directly.

To protect and unprotect sheets and cells:

```
>>> S('Sheet1.a1').protect_sheet()
>>> S('Sheet1.a1').unprotect_sheet()
>>> S('Sheet1.a1').protect_sheet("secretpassword")
>>> S('Sheet1.a1').unprotect_sheet("secretpassword")
>>> S('Sheet1.a1').protect()
>>> S('Sheet1.a1').unprotect()
```

4.3 Working with documents types other than spreadsheet

Although OOSheet high-level API is developed for spreadsheets, it's base class **OODoc** will make macro development job much easier for other types of document. The facility of testing your code via socket and then packing with oosheet's packing tool is the same. And OODoc's API for dispatching events is much simpler than OpenOffice.org Basic code.

4.3.1 Recording macros

First thing you want is to know what kind of OpenOffice.org events will do what you need to do. For that, you can record a macro in OpenOffice.org Basic.

For recording a macro, go to **Tools -> Macros -> Record Macro**. A recording dialog will open, and every action will do will be recorded. Do some actions you'd like to reproduce later, then click the "stop recording" button in the recording dialog. You'll need to select a name for saving this macro. After saving, go to **Tools -> Macros -> Organize Macros... -> OpenOffice.org Basic**, find the macro you just saved and click in "Edit". Check the code you just recorded.

The following code is an example of OpenOffice.org Basic macro for typing "Hello World" and then centering it in Writer:

```
document   = ThisComponent.CurrentController.Frame
dispatcher = createUnoService("com.sun.star.frame.DispatchHelper")

rem ----------------------------------------------------------------------
dim args1(0) as new com.sun.star.beans.PropertyValue
args1(0).Name = "Text"
args1(0).Value = "Hello World"
dispatcher.executeDispatch(document, ".uno:InsertText", "", 0, args1())

rem ----------------------------------------------------------------------
dim args2(0) as new com.sun.star.beans.PropertyValue
args2(0).Name = "CenterPara"
args2(0).Value = true
dispatcher.executeDispatch(document, ".uno:CenterPara", "", 0, args2())
```

The following Python code would do exactly the same thing, with OODoc:

```
>>> from oosheet import OODoc
>>> doc = OODoc()
>>> doc.dispatch('InsertText', ('Text', 'Hello World'))
>>> doc.dispatch('CenterPara', True)
```

Note that the args1, passed in the first dispatcher.executeDispatch() call, is substituted for a tupple (Name, Value) in Python code, while the args2 value can be represented by just a boolean. The tuple can be substituted by a single value when the Name of the property is the same as the command being executed.

The same will work with OOSheet:

```
>>> from oosheet import OOSheet as S
>>> S().dispatch('AutomaticCalculation', False)
```

4.4 Python Macros

To make a macro, create a python script and create a function in global scope for each routine you want to call from your document. Since the same code can run outside OpenOffice.org, you're advised to make a script that will work in standalone context. The template below might be a good start:

> #!/usr/bin/ipython
>
> from oosheet import OOSheet as S
>
> **def my_macro():** # do something pass
>
> **def my_other_macro():** pass

Using ipython will give you a python shell when you run you script from command line. Do this to test your script. When you're done, you have three options to run the macro from OpenOffice.org:

- Put your script in the global python scripts path. This is some directory like /usr/lib/openoffice/basis3.2/share/Scripts/python/. The macro you be available to all users in this computer.
- Put your script in the local python scripts path of your user. The path is something like ~/.openoffice.org/3/user/Scripts/python
- Pack the macro in the document. Details below.

If you choose one of the first two methods, which are simpler, you won't need any security configurations and will be able to run the macro for several documents. The third method makes more sense if your script logic is tied to an specific document structure and/or if you want to have stuff like buttons that trigger your macros.

In any of these methods, you can run you macro from Tools -> Macros -> Run macro menu.

4.4.1 Packing your script in document

If you go to Tools -> Macros -> Organize macros -> Python, you'll notice that the "Create", "Edit", "Rename" and "Remove" options are disabled. This is because OO.org does not support managing your macros yet. To solve this, OOSheet comes with a command-line tool to pack your script in document. Just type:

> oosheet-pack my_document.ods my_script.py

When you open the document, you'll be warned that the document contains macros and that this is a security issue. So, you have to go to Tools -> Options -> Security -> Macro Security and configure it properly. It's a smarty thing to leave the security level at least "High".

4.5 Issues

4.5.1 Breakpoint issue

It's worth noticing that *ipdb.set_trace() does not work* when you use OOSheet. This is not an issue from this module, it happens in deeper and darker layers of python-uno. If you see an error like this:

> SystemError: 'pyuno runtime is not initialized, (the pyuno.bootstrap needs to be called before using any uno classes)'

it's probably because you have an ipdb breakpoint. Use *pdb* instead.

4.5.2 Connection issues

In version 1.2 performance got much better when using sockets, by caching the conection. The drawback is that when connection is broken, script must be restarted.

Sometimes the initial connection takes a long time. This was not reported in OpenOffice.org, but with LibreOffice this is a bit common. Any clue on why this helps is very welcome.

CHAPTER 5

API Reference

5.1 OODoc

5.2 OOSheet

5.3 OOPacker

CHAPTER 6

Contributing

Please submit bugs and patches, preferably with tests. All contributors will be acknowledged. Thanks!

CHAPTER 7

Credits

OOSheet was created by Luis Fagundes and was sponsored by hacklab/ until version 1.2.

Fudge project also take credits for a good documentation structure, on which this one was based.

Oscar Garcia suggested the user selection feature, implemented in v0.9.4.

Thomas Lundqvist sent implementation of data_array property, implemented in v0.9.6.

Luc Jean sent patch that allows OOSheet to run on Windows Vista with default Python interpreter, v0.9.7

CHAPTER 8

Changelog

- 1.3
  - Works with Python 3 and LibreOffice 5
  - Probably better support for Windows
  - Removed dynamic column constant (ex: from oosheet.colums import C)
- 1.2.1
  - Move OOSheet.Quit() to OODoc.quit()
  - Correct bug in OOPacker.script_name() for Windows 7
  - Change fetching InstallPath winreg key for LibreOffice 3.4 on Windows 7
- 1.2
  - Much better performance when acessing via sockets
  - [NEW] Objects can be accessed as arrays
  - [NEW] oosheet-launch reminds that complicated launching command line
- 1.1
  - [NEW] support for LibreOffice
  - [NEW] find() methods searches selection for matching cells
- 1.0
  - [NEW] Iterators for cells, rows and columns
  - [MOD] Refactor for working with all types of documents
- 0.9.7
  - [FIX] works with default Python interpreter in Windows Vista
- 0.9.6
  - [NEW] data_array property returns selection's values as 2d-tuple

- 0.9.5
  - [FIX] flatten() breaking fields with formulas based on zero values formatted as strings (ex: R$ 0,00)
- 0.9.4
  - [NEW] S() now gets user's selection
  - [FIX] Date format is not automatically set if current format is already a date
- 0.9.3.1
  - Fixes packaging problem
- 0.9.3
  - [NEW] flatten() method
  - [NEW] sheet and cell protection
- 0.9.2
  - [NEW] grow_DIRECTION_until and shrink_DIRECTION_until methods
  - [NEW] last_row, last_column, first_row, first_column properties
  - [FIX] api documentation error in shift_until()
  - [FIX] __repr__ of empty selector raised error
- 0.9.1
  - Documentation changed to include installation instructions with pip and link to website
  - Code is same as 0.9.0
- 0.9.0
  - first release