openpyxl Documentation

Release 2.2.4

Eric Gazoni

June 30, 2015

Author Eric Gazoni

Source code http://bitbucket.org/openpyxl/openpyxl/src

Issues http://bitbucket.org/openpyxl/openpyxl/issues

Generated June 30, 2015

License MIT/Expat

Version 2.2.4

CHAPTER 1

Introduction

Openpyxl is a Python library for reading and writing Excel 2010 xlsx/xlsm/xltx/xltm files.

It was born from lack of existing library to read/write natively from Python the Office Open XML format.

All kudos to the PHPExcel team as openpyxl was initially based on PHPExcel http://www.phpexcel.net/

1.1 Sample code:

```
from openpyxl import Workbook
wb = Workbook()

# grab the active worksheet
ws = wb.active

# Data can be assigned directly to cells
ws['A1'] = 42

# Rows can also be appended
ws.append([1, 2, 3])

# Python types will automatically be converted
import datetime
ws['A2'] = datetime.datetime.now()

# Save the file
wb.save("sample.xlsx")
```

CHAPTER 2

User List

Official user list can be found on http://groups.google.com/group/openpyxl-users

CHAPTER 3

How to Contribute Code

Any help will be greatly appreciated, just follow those steps:

1. Please start a new fork (https://bitbucket.org/openpyxl/openpyxl/fork) for each independent feature, don't try to fix all problems at the same time, it's easier for those who will review and merge your changes ;-)

2. Hack hack hack

3. Don't forget to add unit tests for your changes ! (YES, even if it's a one-liner, or there is a high probability your work will not be taken into consideration). There are plenty of examples in the /test directory if you lack know-how or inspiration.

4. If you added a whole new feature, or just improved something, you can be proud of it, so add yourself to the AUTHORS file :-)

5. Let people know about the shiny thing you just implemented, update the docs !

6. When it's done, just issue a pull request (click on the large "pull request" button on *your* repository) and wait for your code to be reviewed, and, if you followed all theses steps, merged into the main repository.

For further information see Development Tools

This is an open-source project, maintained by volunteers on their spare time, so while we try to work on this project as often as possible, sometimes life gets in the way. Please be patient.

CHAPTER 4

Other ways to help

There are several ways to contribute, even if you can't code (or can't code well):

- triaging bugs on the bug tracker: closing bugs that have already been closed, are not relevant, cannot be reproduced, ...
- updating documentation in virtually every area: many large features have been added (mainly about charts and images at the moment) but without any documentation, it's pretty hard to do anything with it
- proposing compatibility fixes for different versions of Python: we support 2.6 to 3.4, so if it does not work on your environment, let us know :-)

CHAPTER 5

Installation

The best method to install openpyxl is using a PyPi client such as easy_install (setuptools) or pip. It is advisable to do this in a Python virtualenv without system packages:

```
$ pip install openpyxl
```

or

```
$ easy_install openpyxl
```

Note: To install from sources (there is nothing to build, openpyxl is 100% pure Python), you can download an archive from bitbucket (look in the “tags” tab).

There is support for the popular lxml library which will be used if it is installed.

After extracting the archive, you can do:

```
$ python setup.py install
```

Warning: To be able to include images (jpeg,png,bmp,...) into an openpyxl file, you will also need the ‘PIL’ library that can be installed with:

```
$ pip install pillow
```

or browse https://pypi.python.org/pypi/Pillow/, pick the latest version and head to the bottom of the page for Windows binaries.

CHAPTER 6

Getting the source

Source code is hosted on bitbucket.org. You can get it using a Mercurial client and the following URLs:

- $ hg clone https://bitbucket.org/openpyxl/openpyxl -r 2.2.4

or to get the latest development version:

- $ hg clone https://bitbucket.org/openpyxl/openpyxl

CHAPTER 7

Usage examples

7.1 Tutorial

7.1.1 Manipulating a workbook in memory

Create a workbook

There is no need to create a file on the filesystem to get started with openpyxl. Just import the Worbook class and start using it

```
>>> from openpyxl import Workbook
>>> wb = Workbook()
```

A workbook is always created with at least one worksheet. You can get it by using the `openpyxl.workbook.Workbook.active()` property

```
>>> ws = wb.active
```

Note: This function uses the *_active_sheet_index* property, set to 0 by default. Unless you modify its value, you will always get the first worksheet by using this method.

You can also create new worksheets by using the `openpyxl.workbook.Workbook.create_sheet()` method

```
>>> ws1 = wb.create_sheet() # insert at the end (default)
# or
>>> ws2 = wb.create_sheet(0) # insert at first position
```

Sheets are given a name automatically when they are created. They are numbered in sequence (Sheet, Sheet1, Sheet2, ...). You can change this name at any time with the *title* property:

```
ws.title = "New Title"
```

The background color of the tab holding this title is white by default. You can change this providing an RRGGBB color code to the sheet_properties.tabColor property:

```
ws.sheet_properties.tabColor = "1072BA"
```

Once you gave a worksheet a name, you can get it as a key of the workbook or using the `openpyxl.workbook.Workbook.get_sheet_by_name()` method

```
>>> ws3 = wb["New Title"]
>>> ws4 = wb.get_sheet_by_name("New Title")
>>> ws is ws3 is ws4
True
```

You can review the names of all worksheets of the workbook with the `openpyxl.workbook.Workbook.get_sheet_names()` method

```
>>> print(wb.get_sheet_names())
['Sheet2', 'New Title', 'Sheet1']
```

You can loop through worksheets

```
>>> for sheet in wb:
...     print(sheet.title)
```

Playing with data

Accessing one cell

Now we know how to access a worksheet, we can start modifying cells content.

Cells can be accessed directly as keys of the worksheet

```
>>> c = ws['A4']
```

This will return the cell at A4 or create one if it does not exist yet. Values can be directly assigned

```
>>> ws['A4'] = 4
```

There is also the `openpyxl.worksheet.Worksheet.cell()` method:

```
>>> c = ws.cell('A4')
```

You can also access a cell using row and column notation:

```
>>> d = ws.cell(row = 4, column = 2)
```

Note: When a worksheet is created in memory, it contains no *cells*. They are created when first accessed. This way we don't create objects that would never be accessed, thus reducing the memory footprint.

Warning: Because of this feature, scrolling through cells instead of accessing them directly will create them all in memory, even if you don't assign them a value.
Something like

```
>>> for i in range(1,101):
...         for j in range(1,101):
...             ws.cell(row = i, column = j)
```

will create 100x100 cells in memory, for nothing.
However, there is a way to clean all those unwanted cells, we'll see that later.

Accessing many cells

Ranges of cells can be accessed using slicing

```
>>> cell_range = ws['A1':'C2']
```

You can also use the `openpyxl.worksheet.Worksheet.iter_rows()` method:

```
>>> tuple(ws.iter_rows('A1:C2'))
((<Cell Sheet1.A1>, <Cell Sheet1.B1>, <Cell Sheet1.C1>),
 (<Cell Sheet1.A2>, <Cell Sheet1.B2>, <Cell Sheet1.C2>))

>>> for row in ws.iter_rows('A1:C2'):
...        for cell in row:
...            print cell
<Cell Sheet1.A1>
<Cell Sheet1.B1>
<Cell Sheet1.C1>
<Cell Sheet1.A2>
<Cell Sheet1.B2>
<Cell Sheet1.C2>
```

If you need to iterate through all the rows or columns of a file, you can instead use the `openpyxl.worksheet.Worksheet.rows()` property:

```
>>> ws = wb.active
>>> ws.['C9'] = 'hello world'
>>> ws.rows
((<Cell Sheet.A1>, <Cell Sheet.B1>, <Cell Sheet.C1>),
(<Cell Sheet.A2>, <Cell Sheet.B2>, <Cell Sheet.C2>),
(<Cell Sheet.A3>, <Cell Sheet.B3>, <Cell Sheet.C3>),
(<Cell Sheet.A4>, <Cell Sheet.B4>, <Cell Sheet.C4>),
(<Cell Sheet.A5>, <Cell Sheet.B5>, <Cell Sheet.C5>),
(<Cell Sheet.A6>, <Cell Sheet.B6>, <Cell Sheet.C6>),
(<Cell Sheet.A7>, <Cell Sheet.B7>, <Cell Sheet.C7>),
(<Cell Sheet.A8>, <Cell Sheet.B8>, <Cell Sheet.C8>),
(<Cell Sheet.A9>, <Cell Sheet.B9>, <Cell Sheet.C9>))
```

or the `openpyxl.worksheet.Worksheet.columns()` property:

```
>>> ws.columns
((<Cell Sheet.A1>,
<Cell Sheet.A2>,
<Cell Sheet.A3>,
<Cell Sheet.A4>,
<Cell Sheet.A5>,
<Cell Sheet.A6>,
...
<Cell Sheet.B7>,
<Cell Sheet.B8>,
<Cell Sheet.B9>),
(<Cell Sheet.C1>,
<Cell Sheet.C2>,
<Cell Sheet.C3>,
<Cell Sheet.C4>,
<Cell Sheet.C5>,
<Cell Sheet.C6>,
<Cell Sheet.C7>,
<Cell Sheet.C8>,
<Cell Sheet.C9>))
```

Data storage

Once we have a `openpyxl.cell.Cell`, we can assign it a value:

```
>>> c.value = 'hello, world'
>>> print(c.value)
'hello, world'

>>> d.value = 3.14
>>> print(d.value)
3.14
```

You can also enable type and format inference:

```
>>> wb = Workbook(guess_types=True)
>>> c.value = '12%'
>>> print(c.value)
0.12

>>> import datetime
>>> d.value = datetime.datetime.now()
>>> print d.value
datetime.datetime(2010, 9, 10, 22, 25, 18)

>>> c.value = '31.50'
>>> print(c.value)
31.5
```

7.1.2 Saving to a file

The simplest and safest way to save a workbook is by using the `openpyxl.workbook.Workbook.save()` method of the `openpyxl.workbook.Workbook` object:

```
>>> wb = Workbook()
>>> wb.save('balances.xlsx')
```

Warning: This operation will overwrite existing files without warning.

Note: Extension is not forced to be xlsx or xlsm, although you might have some trouble opening it directly with another application if you don’t use an official extension.

As OOXML files are basically ZIP files, you can also end the filename with .zip and open it with your favourite ZIP archive manager.

You can specify the attribute as_template=True, to save the document as a template

```
>>> wb = load_workbook('document.xlsx')
>>> wb.save('document_template.xltx', as_template=True)
```

or specify the attribute as_template=False (by default), to save the document template (or document) as document.

```
>>> wb = load_workbook('document_template.xltx')
>>> wb.save('document.xlsx', as_template=False)
```

```
>>> wb = load_workbook('document.xlsx')
>>> wb.save('new_document.xlsx', as_template=False)
```

Warning: You should monitor the data attributes and document extensions for saving documents in the document templates and vice versa, otherwise the result table engine can not open the document.

Note: The following will fail:

```
>>> wb = load_workbook('document.xlsx')
>>> # Need to save with the extension *.xlsx
>>> wb.save('new_document.xlsm')
>>> # MS Excel can't open the document
>>>
>>> # or
>>>
>>> # Need specify attribute keep_vba=True
>>> wb = load_workbook('document.xlsm')
>>> wb.save('new_document.xlsm')
>>> # MS Excel can't open the document
>>>
>>> # or
>>>
>>> wb = load_workbook('document.xltm', keep_vba=True)
>>> # If us need template document, then we need specify extension as *.xltm.
>>> # If us need document, then we need specify attribute as_template=False.
>>> wb.save('new_document.xlsm', as_template=True)
>>> # MS Excel can't open the document
```

7.1.3 Loading from a file

The same way as writing, you can import `openpyxl.load_workbook()` to open an existing workbook:

```
>>> from openpyxl import load_workbook
>>> wb2 = load_workbook('test.xlsx')
>>> print wb2.get_sheet_names()
['Sheet2', 'New Title', 'Sheet1']
```

This ends the tutorial for now, you can proceed to the Simple usage section

7.2 Cookbook

7.2.1 Simple usage

Write a workbook

```
>>> from openpyxl import Workbook
>>> from openpyxl.compat import range
>>> from openpyxl.cell import get_column_letter
>>>
>>> wb = Workbook()
>>>
>>> dest_filename = 'empty_book.xlsx'
>>>
>>> ws1 = wb.active
>>> ws1.title = "range names"
```

```
>>>
>>> for row in range(1, 40):
...     ws1.append(range(600))
>>>
>>> ws2 = wb.create_sheet(title="Pi")
>>>
>>> ws2['F5'] = 3.14
>>>
>>> ws3 = wb.create_sheet(title="Data")
>>> for row in range(10, 20):
...     for col in range(27, 54):
...         _ = ws3.cell(column=col, row=row, value="%s" % get_column_letter(col))
>>> print(ws3['AA10'].value)
AA
>>> wb.save(filename = dest_filename)
```

Write a workbook from *.xltx as *.xlsx

```
>>> from openpyxl import load_workbook
>>>
>>>
>>> wb = load_workbook('sample_book.xltx')
>>> ws = wb.active
>>> ws['D2'] = 42
>>>
>>> wb.save('sample_book.xlsx')
>>>
>>> # or you can overwrite the current document template
>>> # wb.save('sample_book.xltx')
```

Write a workbook from *.xltm as *.xlsm

```
>>> from openpyxl import load_workbook
>>>
>>>
>>> wb = load_workbook('sample_book.xltm', keep_vba=True)
>>> ws = wb.active
>>> ws['D2'] = 42
>>>
>>> wb.save('sample_book.xlsm')
>>>
>>> # or you can overwrite the current document template
>>> # wb.save('sample_book.xltm')
```

Read an existing workbook

```
>>> from openpyxl import load_workbook
>>> wb = load_workbook(filename = 'empty_book.xlsx')
>>> sheet_ranges = wb['range names']
>>> print(sheet_ranges['D18'].value)
3
```

Note: There are several flags that can be used in load_workbook.

- *guess_types* will enable or disable (default) type inference when reading cells.
- *data_only* controls whether cells with formulae have either the formula (default) or the value stored the last time Excel read the sheet.
- *keep_vba* controls whether any Visual Basic elements are preserved or not (default). If they are preserved they are still not editable.

Warning: openpyxl does currently not read all possible items in an Excel file so images and charts will be lost from existing files if they are opened and saved with the same name.

Using number formats

```
>>> import datetime
>>> from openpyxl import Workbook
>>> wb = Workbook(guess_types=True)
>>> ws = wb.active
>>> # set date using a Python datetime
>>> ws['A1'] = datetime.datetime(2010, 7, 21)
>>>
>>> ws['A1'].number_format
'yyyy-mm-dd h:mm:ss'
>>>
>>> # set percentage using a string followed by the percent sign
>>> ws['B1'] = '3.14%'
>>>
>>> ws['B1'].value
0.031400000000000004
>>>
>>> ws['B1'].number_format
'0%'
```

Using formulae

```
>>> from openpyxl import Workbook
>>> wb = Workbook()
>>> ws = wb.active
>>> # add a simple formula
>>> ws["A1"] = "=SUM(1, 1)"
>>> wb.save("formula.xlsx")
```

Warning: NB function arguments *must* be separated by commas and not other punctuation such as semi-colons

Merge / Unmerge cells

```
>>> from openpyxl.workbook import Workbook
>>>
>>> wb = Workbook()
>>> ws = wb.active
>>>
>>> ws.merge_cells('A1:B1')
>>> ws.unmerge_cells('A1:B1')
```

```
>>>
>>> # or
>>> ws.merge_cells(start_row=2,start_column=1,end_row=2,end_column=4)
>>> ws.unmerge_cells(start_row=2,start_column=1,end_row=2,end_column=4)
```

Inserting an image

```
>>> from openpyxl import Workbook
>>> from openpyxl.drawing import Image
>>>
>>> wb = Workbook()
>>> ws = wb.active
>>> ws['A1'] = 'You should see three logos below'
>>> ws['A2'] = 'Resize the rows and cells to see anchor differences'
>>>
>>> # create image instances
>>> img = Image('logo.png')
>>> img2 = Image('logo.png')
>>> img3 = Image('logo.png')
>>>
>>> # place image relative to top left corner of spreadsheet
>>> img.drawing.top = 100
>>> img.drawing.left = 150
>>>
>>> # the top left offset needed to put the image
>>> # at a specific cell can be automatically calculated
>>> img2.anchor(ws['D12'])
(('D', 12), ('D', 21))
>>>
>>> # one can also position the image relative to the specified cell
>>> # this can be advantageous if the spreadsheet is later resized
>>> # (this might not work as expected in LibreOffice)
>>> img3.anchor(ws['G20'], anchortype='oneCell')
((6, 19), None)
>>>
>>> # afterwards one can still add additional offsets from the cell
>>> img3.drawing.left = 5
>>> img3.drawing.top = 5
>>>
>>> # add to worksheet
>>> ws.add_image(img)
>>> ws.add_image(img2)
>>> ws.add_image(img3)
>>> wb.save('logo.xlsx')
```

Fold columns (outline)

```
>>> import openpyxl
>>> wb = openpyxl.Workbook(True)
>>> ws = wb.create_sheet()
>>> ws.column_dimensions.group('A','D', hidden=True)
>>> wb.save('group.xlsx')
```

7.3 Charts

7.3.1 Charts

Warning: Openpyxl currently supports chart creation within a worksheet only. Charts in existing workbooks will be lost.

Chart types

The following charts are available:

- Bar Chart
- Line Chart
- Scatter Chart
- Pie Chart

Creating a chart

Charts are composed of at least one series of one or more data points. Series themselves are comprised of references to cell ranges.

```
>>> from openpyxl import Workbook
>>> wb = Workbook()
>>> ws = wb.active
>>> for i in range(10):
...     ws.append([i])
>>>
>>> from openpyxl.charts import BarChart, Reference, Series
>>> values = Reference(ws, (1, 1), (10, 1))
>>> series = Series(values, title="First series of values")
>>> chart = BarChart()
>>> chart.append(series)
>>> ws.add_chart(chart)
>>> wb.save("SampleChart.xlsx")
```

7.4 Comments

7.4.1 Comments

Warning: Openpyxl currently supports the reading and writing of comment text only. Formatting information is lost. Comments are not currently supported if *use_iterators=True* is used.

Adding a comment to a cell

Comments have a text attribute and an author attribute, which must both be set

```
>>> from openpyxl import Workbook
>>> from openpyxl.comments import Comment
>>> wb = Workbook()
>>> ws = wb.active
>>> comment = ws["A1"].comment
>>> comment = Comment('This is the comment text', 'Comment Author')
>>> comment.text
'This is the comment text'
>>> comment.author
'Comment Author'
```

You cannot assign the same Comment object to two different cells. Doing so raises an AttributeError.

```
>>> from openpyxl import Workbook
>>> from openpyxl.comments import Comment
>>> wb=Workbook()
>>> ws=wb.active
>>> comment = Comment("Text", "Author")
>>> ws["A1"].comment = comment
>>> ws["B2"].comment = comment
Traceback (most recent call last):
AttributeError: Comment already assigned to A1 in worksheet Sheet. Cannot
assign a comment to more than one cell
```

Loading and saving comments

Comments present in a workbook when loaded are stored in the comment attribute of their respective cells automatically. Formatting information such as font size, bold and italics are lost, as are the original dimensions and position of the comment's container box.

Comments remaining in a workbook when it is saved are automatically saved to the workbook file.

7.5 Read/write large files

7.5.1 Optimized reader

Sometimes, you will need to open or write extremely large XLSX files, and the common routines in openpyxl won't be able to handle that load. Fortunately, there are two modes that enable you to read and write unlimited amounts of data with (near) constant memory consumption.

Introducing `openpyxl.worksheet.iter_worksheet.IterableWorksheet`:

```
from openpyxl import load_workbook
wb = load_workbook(filename='large_file.xlsx', read_only=True)
ws = wb['big_data'] # ws is now an IterableWorksheet

for row in ws.rows:
    for cell in row:
        print(cell.value)
```

Warning:
- `openpyxl.worksheet.iter_worksheet.IterableWorksheet` are read-only

Cells returned are not regular `openpyxl.cell.cell.Cell` but `openpyxl.cell.read_only.ReadOnlyCell`.

7.5.2 Optimized writer

Here again, the regular *openpyxl.worksheet.worksheet.Worksheet* has been replaced by a faster alternative, the *openpyxl.writer.dump_worksheet.DumpWorksheet*. When you want to dump large amounts of data, you might find optimized writer helpful.

```
>>> from openpyxl import Workbook
>>> wb = Workbook(write_only=True)
>>> ws = wb.create_sheet()
>>>
>>> # now we'll fill it with 100 rows x 200 columns
>>>
>>> for irow in range(100):
...     ws.append(['%d' % i for i in range(200)])
>>> # save the file
>>> wb.save('new_big_file.xlsx')
```

If you want to have cells with styles or comments then use a *openpyxl.writer.dump_worksheet.WriteOnlyCell()*

```
>>> from openpyxl import Workbook
>>> wb = Workbook(optimized_write = True)
>>> ws = wb.create_sheet()
>>> from openpyxl.writer.dump_worksheet import WriteOnlyCell
>>> from openpyxl.comments import Comment
>>> from openpyxl.styles import Style, Font
>>> cell = WriteOnlyCell(ws, value="hello world")
>>> cell.font = Font(name='Courrier', size=36)
>>> cell.comment = Comment(text="A comment", author="Author's Name")
```

This will append one new row with 3 cells, one text cell with custom font and font size, a float and an empty cell that will be discarded anyway.

Warning:

- Those worksheet only have an append() method, it's not possible to access independent cells directly (through cell() or range()). They are write-only.
- It is able to export unlimited amount of data (even more than Excel can handle actually), while keeping memory usage under 10Mb.
- A workbook using the optimized writer can only be saved once. After that, every attempt to save the workbook or append() to an existing worksheet will raise an *openpyxl.utils.exceptions.WorkbookAlreadySaved* exception.

7.6 Working with styles

7.6.1 Working with styles

Introduction

Styles are used to change the look of your data while displayed on screen. They are also used to determine the number format being used for a given cell or range of cells.

Styles can be applied to the following aspects:

- font to set font size, color, underlining, etc.
- fill to set a pattern or color gradient

- border to set borders on a cell
- cell alignment
- protection

The following are the default values

```
>>> from openpyxl.styles import PatternFill, Border, Side, Alignment, Protection, Font
>>> font = Font(name='Calibri',
...                 size=11,
...                 bold=False,
...                 italic=False,
...                 vertAlign=None,
...                 underline='none',
...                 strike=False,
...                 color='FF000000')
>>> fill = PatternFill(fill_type=None,
...                 start_color='FFFFFFFF',
...                 end_color='FF000000')
>>> border = Border(left=Side(border_style=None,
...                           color='FF000000'),
...                 right=Side(border_style=None,
...                            color='FF000000'),
...                 top=Side(border_style=None,
...                          color='FF000000'),
...                 bottom=Side(border_style=None,
...                             color='FF000000'),
...                 diagonal=Side(border_style=None,
...                               color='FF000000'),
...                 diagonal_direction=0,
...                 outline=Side(border_style=None,
...                              color='FF000000'),
...                 vertical=Side(border_style=None,
...                               color='FF000000'),
...                 horizontal=Side(border_style=None,
...                                color='FF000000')
...                )
>>> alignment=Alignment(horizontal='general',
...                     vertical='bottom',
...                     text_rotation=0,
...                     wrap_text=False,
...                     shrink_to_fit=False,
...                     indent=0)
>>> number_format = 'General'
>>> protection = Protection(locked=True,
...                         hidden=False)
>>>
```

Styles are shared between objects and once they have been assigned they cannot be changed. This stops unwanted side-effects such as changing the style for lots of cells when instead of only one.

```
>>> from openpyxl.styles import colors
>>> from openpyxl.styles import Font, Color
>>> from openpyxl.styles import colors
>>> from openpyxl import Workbook
>>> wb = Workbook()
>>> ws = wb.active
>>>
>>> a1 = ws['A1']
>>> d4 = ws['D4']
>>> ft = Font(color=colors.RED)
>>> a1.font = ft
>>> d4.font = ft
>>>
>>> a1.font.italic = True # is not allowed
>>>
>>> # If you want to change the color of a Font, you need to reassign it::
>>>
>>> a1.font = Font(color=colors.RED, italic=True) # the change only affects A1
```

Copying styles

Styles can also be copied

```
>>> from openpyxl.styles import Font
>>>
>>> ft1 = Font(name='Arial', size=14)
>>> ft2 = ft1.copy(name="Tahoma")
>>> ft1.name
'Arial'
>>> ft2.name
'Tahoma'
>>> ft2.size # copied from the
14.0
```

Basic Font Colors

Colors are usually RGB or aRGB hexvalues. The *colors* module contains some constants

```
>>> from openpyxl.styles import Font
>>> from openpyxl.styles.colors import RED
>>> font = Font(color=RED)
>>> font = Font(color="FFBB00")
```

There is also support for legacy indexed colors as well as themes and tints

```
>>> from openpyxl.styles.colors import Color
>>> c = Color(indexed=32)
>>> c = Color(theme=6, tint=0.5)
```

Applying Styles

Styles are applied directly to cells

```
>>> from openpyxl.workbook import Workbook
>>> from openpyxl.styles import Font, Fill
>>> wb = Workbook()
>>> ws = wb.active
>>> c = ws['A1']
>>> c.font = Font(size=12)
```

Styles can also applied to columns and rows but note that this applies only to cells created (in Excel) after the file is closed. If you want to apply styles to entire rows and columns then you must apply the style to each cell yourself. This is a restriction of the file format:

```
>>> col = ws.column_dimensions['A']
>>> col.font = Font(bold=True)
>>> row = ws.row_dimensions[1]
>>> row.font = Font(underline="single")
```

Edit Page Setup

```
>>> from openpyxl.workbook import Workbook
>>>
>>> wb = Workbook()
>>> ws = wb.active
>>>
>>> ws.page_setup.orientation = ws.ORIENTATION_LANDSCAPE
>>> ws.page_setup.paperSize = ws.PAPERSIZE_TABLOID
>>> ws.page_setup.fitToHeight = 0
>>> ws.page_setup.fitToWidth = 1
```

Edit Print Options

```
>>> from openpyxl.workbook import Workbook
>>>
>>> wb = Workbook()
>>> ws = wb.active
>>>
>>> ws.print_options.horizontalCentered = True
>>> ws.print_options.verticalCentered = True
```

Header / Footer

Headers and footers use their own formatting language. This is fully supported when writing them but, due to the complexity and the possibility of nesting, only partially when reading them.

```
>>> from openpyxl.workbook import Workbook
>>>
>>> wb = Workbook()
>>> ws = wb.worksheets[0]
>>>
>>> ws.header_footer.center_header.text = 'My Excel Page'
>>> ws.header_footer.center_header.font_size = 14
>>> ws.header_footer.center_header.font_name = "Tahoma,Bold"
>>> ws.header_footer.center_header.font_color = "CC3366"
```

# Or just >>> ws.header_footer.right_footer.text = 'My Right Footer'

Worksheet Additional Properties

These are advanced properties for particular behaviours, the most used ones are the "fitTopage" page setup property and the tabColor that define the background color of the worksheet tab.

Available properties for worksheet: "codeName", "enableFormatConditionsCalculation", "filterMode", "published", "syncHorizontal", "syncRef", "syncVertical", "transitionEvaluation", "transitionEntry", "tabColor". Available fields for page setup properties: "autoPageBreaks", "fitToPage". Available fields for outline properties: "applyStyles", "summaryBelow", "summaryRight", "showOutlineSymbols".

see http://msdn.microsoft.com/en-us/library/documentformat.openxml.spreadsheet.sheetproperties%28v=office.14%29.aspx_ for details.

..note:: By default, outline properties are intitialized so you can directly modify each of their 4 attributes, while page setup properties don't. If you want modify the latter, you should first initialize a PageSetupPr object with the required parameters. Once done, they can be directly modified by the routine later if needed.

```
>>> from openpyxl.workbook import Workbook
>>> from openpyxl.worksheet.properties import WorksheetProperties, PageSetupProperties
>>>
>>> wb = Workbook()
>>> ws = wb.active
>>>
>>> wsprops = ws.sheet_properties
>>> wsprops.tabColor = "1072BA"
>>> wsprops.filterMode = False
>>> wsprops.PageSetupProperties = PageSetupProperties(fitToPage=True, autoPageBreaks=False)
>>> wsprops.outlinePr.summaryBelow = False
>>> wsprops.outlinePr.applyStyles = True
>>> wsprops.PageSetupProperties.autoPageBreaks = True
```

7.7 Conditional Formatting

7.7.1 Conditional Formatting

There are many types of conditional formatting - below are some examples for setting this within an excel file.

```
>>> from openpyxl import Workbook
>>> from openpyxl.styles import Color, PatternFill, Font, Border
>>> from openpyxl.formatting import ColorScaleRule, CellIsRule, FormulaRule
>>>
>>> wb = Workbook()
>>> ws = wb.active
>>>
>>> # Create fill
>>> redFill = PatternFill(start_color='FFEE1111',
...                end_color='FFEE1111',
...                fill_type='solid')
>>>
>>> # Add a two-color scale
>>> # add2ColorScale(range_string, start_type, start_value, start_color, end_type, end_value, end_co
>>> # Takes colors in excel 'FFRRGGBB' style.
>>> ws.conditional_formatting.add('A1:A10',
...             ColorScaleRule(start_type='min', start_color=Color('FFAA0000'),
...                           end_type='max', end_color=Color('FF00AA00'))
...                           )
>>>
>>> # Add a three-color scale
>>> ws.conditional_formatting.add('B1:B10',
...                ColorScaleRule(start_type='percentile', start_value=10, start_color=Color('FFAA00
...                            mid_type='percentile', mid_value=50, mid_color=Color('FF0000AA'),
...                            end_type='percentile', end_value=90, end_color=Color('FF00AA00'))
...                              )
>>>
>>> # Add a conditional formatting based on a cell comparison
>>> # addCellIs(range_string, operator, formula, stopIfTrue, wb, font, border, fill)
>>> # Format if cell is less than 'formula'
>>> ws.conditional_formatting.add('C2:C10',
...             CellIsRule(operator='lessThan', formula=['C$1'], stopIfTrue=True, fill=redFill))
>>>
>>> # Format if cell is between 'formula'
>>> ws.conditional_formatting.add('D2:D10',
...             CellIsRule(operator='between', formula=['1','5'], stopIfTrue=True, fill=redFill))
>>>
>>> # Format using a formula
>>> ws.conditional_formatting.add('E1:E10',
...             FormulaRule(formula=['ISBLANK(E1)'], stopIfTrue=True, fill=redFill))
>>>
>>> # Aside from the 2-color and 3-color scales, format rules take fonts, borders and fills for styl
>>> myFont = Font()
>>> myBorder = Border()
>>> ws.conditional_formatting.add('E1:E10',
...             FormulaRule(formula=['E1=0'], font=myFont, border=myBorder, fill=redFill))
>>>
>>> # Custom formatting
>>> # There are many types of conditional formatting - it's possible to add additional types directl
>>> ws.conditional_formatting.add('E1:E10',
...             {'type': 'expression', 'dxf': {'fill': redFill},
...              'formula': ['ISBLANK(E1)'], 'stopIfTrue': '1'})
>>>
>>> wb.save("test.xlsx")
```

7.8 Data Validation

7.8.1 Validating cells

You can add data validation to a workbook but currently cannot read existing data validation.

Examples

```
>>> from openpyxl import Workbook
>>> from openpyxl.worksheet.datavalidation import DataValidation, ValidationType
>>>
>>> # Create the workbook and worksheet we'll be working with
>>> wb = Workbook()
>>> ws = wb.active
>>>
>>> # Create a data-validation object with list validation
>>> dv = DataValidation(type="list", formula1='"Dog,Cat,Bat"', allow_blank=True)
>>>
>>> # Optionally set a custom error message
>>> dv.error ='Your entry is not in the list'
>>> dv.errorTitle = 'Invalid Entry'
>>>
>>> # Optionally set a custom prompt message
>>> dv.prompt = 'Please select from the list'
>>> dv.promptTitle = 'List Selection'
>>>
>>> # Add the data-validation object to the worksheet
>>> ws.add_data_validation(dv)
>>> # Create some cells, and add them to the data-validation object
>>> c1 = ws["A1"]
>>> c1.value = "Dog"
>>> dv.add(c1)
>>> c2 = ws["A2"]
>>> c2.value = "An invalid value"
>>> dv.add(c2)
>>>
>>> # Or, apply the validation to a range of cells
>>> dv.ranges.append('B1:B1048576')
>>>
>>> # Write the sheet out.  If you now open the sheet in Excel, you'll find that
>>> # the cells have data-validation applied.
>>> wb.save("test.xlsx")
```

Other validation examples

Any whole number:

```
dv = DataValidation(type="whole")
```

Any whole number above 100:

```
dv = DataValidation(type="whole",
                    operator="greaterThan",
                    formula1=100)
```

Any decimal number:

```
dv = DataValidation(type="decimal")
```

Any decimal number between 0 and 1:

```
dv = DataValidation(type="decimal",
                    operator="between",
                    formula1=0,
                    formula2=1)
```

Any date:

```
dv = DataValidation(type="date")
```

or time:

```
dv = DataValidation(type="time")
```

Any string at most 15 characters:

```
dv = DataValidation(type="textLength",
                    operator="lessThanOrEqual"),
                    formula1=15)
```

Custom rule:

```
dv = DataValidation(type="custom",
                    formula1"=SOMEFORMULA")
```

Note: See http://www.contextures.com/xlDataVal07.html for custom rules

CHAPTER 8

Information for Developers

8.1 Development

With the ongoing development of openpyxl, there is occasional information useful to assist developers.

8.1.1 What is suppoprted

The primary aim of openpyxl is to support reading and writing Microsoft Excel 2010 files. Where possible support for files generated by other libraries or programs is available but this is not guaranteed.

8.1.2 Supporting different Python versions

We have a small library of utility functions to support development for Python 2 and 3. This is openpyxl.compat for Python and openpyxl.xml for XML functions.

8.1.3 Coding style

Use PEP-8 except when implementing attributes for roundtripping but always use Python data conventions (boolean, None, etc.) Note exceptions in docstrings.

8.1.4 Testing

Contributions without tests will **not** be accepted.

We use pytest as the test runner with pytest-cov for coverage information and pytest-flakes for static code analysis.

Coverage

The goal is 100 % coverage for unit tests - data types and utility functions. Coverage information can be obtained using

```
py.test --cov openpyxl
```

Organisation

Tests can be at library - openpyxl/tests or preferably for unit tests at package / module level e.g openpyxl/cell. This makes testing and getting statistics for code under development easier:

```
py.test --cov openpyxl/cell openpyxl/cell
```

Checking XML

Use the `openpyxl.tests.helper.compare_xml` function to compare generated and expected fragments of XML.

Schema validation

When working on code to generate XML it is possible to validate that the generated XML conforms to the published specification. Note, this won't necessarily guarantee that everything is fine but is preferable to reverse engineering!

Microsoft Tools

Along with the SDK, Microsoft also has a "Productivity Tool" for working with Office OpenXML. http://www.microsoft.com/en-us/download/details.aspx?id=30425

It allows you to quickly inspect a whole Excel file. Unfortunately, validation errors contain many false positives.

Please see Testing on Windows for additional information on setting up and testing on Windows.

8.1.5 Contributing

Contributions in the form of pull requests are always welcome. Don't forget to add yourself to the list of authors!

8.1.6 Branch naming convention

We use a "major.minor.patch" numbering system, ie. 1.8.3 Development branches are named after "major.minor" releases. In general, API change will only happen major releases but there will be exceptions. Always communicate API changes to the mailing list before making them. If you are changing an API try and an implement a fallback (with deprecation warning) for the old behaviour.

The "default branch" is used for releases and always has changes from a development branch merged in. It should never be the target for a pull request.

8.1.7 Pull Requests

In general, pull requests should be submitted to the current, unreleased development branch. Eg. if the current release is 1.8.x, pull requests should be made to the 1.9 branch. Exceptions are bug fixes to released versions which should be made to the relevant release branch and merged upstream into development.

Please use tox to test code for different submissions **before** making a pull request. This is especially important for picking up problems across Python versions.

Documentation

Remember to update the documentation when adding or changing features. Check that documentation is syntactically correct

```
tox -e doc
```

8.1.8 Benchmarking

Benchmarking and profiling are ongoing tasks. Contributions to these are very welcome as we know there is a lot to do.

Memory Use

There is a tox profile for long-running memory benchmarks using the *memory_utils* package

```
    tox -e memory
```

Pympler

As openpyxl does not include any internal memory benchmarking tools, the python *pympler* package was used during the testing of styles to profile the memory usage in `openpyxl.reader.excel.read_style_table()`:

```python
# in openpyxl/reader/style.py
from pympler import muppy, summary

def read_style_table(xml_source):
  ...
  if cell_xfs is not None:  # ~ line 47
      initialState = summary.summarize(muppy.get_objects())  # Capture the initial state
      for index, cell_xfs_node in enumerate(cell_xfs_nodes):
         ...
         table[index] = new_style
      finalState = summary.summarize(muppy.get_objects())  # Capture the final state
      diff = summary.get_diff(initialState, finalState)  # Compare
      summary.print_(diff)
```

`pympler.summary.print_()` prints to the console a report of object memory usage, allowing the comparison of different methods and examination of memory usage. A useful future development would be to construct a benchmarking package to measure the performance of different components.

8.2 Testing on Windows

Although openpyxl itself is pure Python and should run on any Python, we do use some libraries that require compiling for tests and documentation. The setup for testing on Windows is somewhat different.

8.2.1 Getting started

Once you have installed the versions of Python (2.6, 2.7, 3.3, 3.4) you should setup a development environment for testing so that you do not adversely affect the system install.

8.2.2 Setting up a development environment

First of all you should checkout a copy of the repository. Atlassian provides a nice GUI client SourceTree that allows you to do this with a single-click from the browser.

By default the repository will be installed under your user folder. eg. c:UsersYOURUSERopenpyxl

Switch to the branch you want to work on by double-clicking it. The default branch should never be used for development work.

Creating a virtual environment

You will need to manually install virtualenv. This is best done by first installing pip. open a command line and download the script "get_pip.py" to your preferred Python folder:

```
bitsadmin /transfer pip http://bootstrap.pypa.io/get-pip.py c:\python27\get-pip.py # change the path
```

Install pip (it needs to be at least pip 6.0):

```
python get_pip.py
```

Now you can install virtualenv:

```
Scripts\pip install virtualenv
Scripts\virtualenv c:\Users\YOURUSER\openpyxl
```

8.2.3 lxml

openpyxl needs *lxml* in order to run the tests. Unfortunately, automatic installation of lxml on Windows is tricky as pip defaults to try and compile it. This can be avoided by using pre-compiled versions of the library.

1. In the command line switch to your repository folder:

```
    cd c:\Users\YOURUSER\openpyxl
```

2. Activate the virtualenv:

```
    Scripts\activate
```

3. Install a development version of openpyxl:

```
    python setup.py develop
```

4. Download all the relevant lxml Windows wheels
5. Move all these files to a folder called "downloads" in your openpyxl checkout
6. Install the project requirements:

```
    pip install --download downloads -r requirements.txt
    pip install --no-index --find-links downloads -r requirements.txt
```

To run tests for the virtualenv:

```
py.test -xrf openpyxl # the flag will stop testing at the first error
```

8.2.4 tox

We use *tox* to run the tests on different Python versions and configurations. Using it is as simple as:

```
set PIP_FIND_LINKS=downloads
tox openpyxl
```

CHAPTER 9

API Documentation

9.1 openpyxl package

9.1.1 Subpackages

openpyxl.cell package

Submodules

openpyxl.cell.cell module

class `openpyxl.cell.cell.`**`Cell`**(*worksheet*, *column*, *row*, *value=None*, *fontId=0*, *fillId=0*, *borderId=0*, *alignmentId=0*, *protectionId=0*, *numFmtId=0*, *pivotButton=None*, *quotePrefix=None*, *xfId=None*)

Bases: *`openpyxl.styles.styleable.StyleableObject`*

Describes cell associated properties.

Properties of interest include style, type, value, and address.

`ERROR_CODES` = **('#NULL!', '#DIV/0!', '#VALUE!', '#REF!', '#NAME?', '#NUM!', '#N/A')**

`TYPE_BOOL` = **'b'**

`TYPE_ERROR` = **'e'**

`TYPE_FORMULA` = **'f'**

`TYPE_FORMULA_CACHE_STRING` = **'str'**

`TYPE_INLINE` = **'inlineStr'**

`TYPE_NULL` = **'n'**

`TYPE_NUMERIC` = **'n'**

`TYPE_STRING` = **'s'**

`VALID_TYPES` = **('s', 'f', 'n', 'b', 'n', 'inlineStr', 'e', 'str')**

`anchor`

returns the expected position of a cell in pixels from the top-left of the sheet. For example, A1 anchor should be (0,0).

Return type tuple(int, int)

base_date

bind_value(**args*, ***kwargs*)

check_error(*value*)
Tries to convert Error" else N/A

check_string(*value*)
Check string coding, length, and line break character

column

comment
Returns the comment associated with this cell

Return type `openpyxl.comments.Comment`

coordinate

data_type

encoding

guess_types

hyperlink
Return the hyperlink target or an empty string

hyperlink_rel_id
Return the id pointed to by the hyperlink, or None

infer_value(**args*, ***kwargs*)

internal_value
Always returns the value for excel.

is_date
Whether the value is formatted as a date

Return type bool

offset(*row=0*, *column=0*)
Returns a cell location relative to this cell.

Parameters

- **row** (*int*) – number of rows to offset
- **column** (*int*) – number of columns to offset

Return type `openpyxl.cell.Cell`

parent

row

set_explicit_value(*value=None*, *data_type='s'*)
Coerce values according to their explicit type

value
Get or set the value held in the cell. ':rtype: depends on the value (string, float, int or ' '`datetime.datetime`)'

xf_index

openpyxl.cell.formula module

class `openpyxl.cell.formula.`**`SharedFormula`**(*range*, *key*, *expression*)

Bases: `object`

`expression`()

Expression

`key`()

Key

`range`()

Range of cells to which the formula applies

openpyxl.cell.interface module

class `openpyxl.cell.interface.`**`AbstractCell`**(*value=None*)

Bases: `abc.ABC`

`base_date`

`comment`

`coordinate`

`encoding`

`guess_types`

`internal_value`

`is_date`

`number_format`

`offset`(*row=0*, *column=0*)

`style`

`value`

openpyxl.cell.read_only module

class `openpyxl.cell.read_only.`**`ReadOnlyCell`**(*sheet*, *row*, *column*, *value*, *data_type='n'*, *style_id=None*)

Bases: `object`

`alignment`

`base_date`

`border`

`column`

`coordinate`

`data_type`

`fill`

`font`

internal_value

is_date

number_format

parent

protection

row

shared_strings

style

style_id

value

Module contents

openpyxl.charts package

Submodules

openpyxl.charts.axis module

class `openpyxl.charts.axis.`**`Axis`**(*auto_axis=False*)

Bases: `object`

ORIENTATION_MIN_MAX = 'minMax'

POSITION_BOTTOM = 'b'

POSITION_LEFT = 'l'

auto = True

cross_between = None

crosses = None

delete_axis = False

label_align = None

label_offset = None

max

min

number_format = 'General'

orientation = 'minMax'

position = None

sourceLinked = True

tick_label_position = None

unit

class `openpyxl.charts.axis.`**`CategoryAxis`**`(`*auto_axis=False*`)`

Bases: *`openpyxl.charts.axis.Axis`*

`auto` = **True**

`cross` = **60873344**

`cross_between` = **'midCat'**

`crosses` = **'autoZero'**

`id` = **60871424**

`label_align` = **'ctr'**

`label_offset` = **100**

`position` = **'b'**

`sourceLinked` = **False**

`tick_label_position` = **'nextTo'**

`type` = **'catAx'**

class `openpyxl.charts.axis.`**`ValueAxis`**`(`*auto_axis=False*`)`

Bases: *`openpyxl.charts.axis.Axis`*

`auto` = **False**

`cross` = **60871424**

`cross_between` = **'between'**

`crosses` = **'autoZero'**

`id` = **60873344**

`major_gridlines` = **None**

`position` = **'l'**

`tick_label_position` = **'nextTo'**

`type` = **'valAx'**

`openpyxl.charts.axis.`**`less_than_one`**`(`*value*`)`

Recalculate the maximum for a series if it is less than one by scaling by powers of 10 until is greater than 1

openpyxl.charts.bar module

class `openpyxl.charts.bar.`**`BarChart`**`(`*auto_axis=False*`)`

Bases: *`openpyxl.charts.graph.GraphChart`*

`GROUPING` = **'clustered'**

`TYPE` = **'barChart'**

openpyxl.charts.chart module

class `openpyxl.charts.chart.`**`Chart`**

Bases: `object`

raw chart class

GROUPING = 'standard'

TYPE = None

add_serie(*obj*)
Add a series or a shape

add_series(*obj*)
Add a series or a shape

add_shape(*obj*)
Add a series or a shape

append(*obj*)
Add a series or a shape

get_y_chars()
estimate nb of chars for y axis

margin_left

margin_top
get margin in percent

mymax(*values*)

mymin(*values*)

openpyxl.charts.error_bar module

class openpyxl.charts.error_bar.**ErrorBar**(*_type*, *values*)
Bases: object

MINUS = 2

PLUS = 1

PLUS_MINUS = 3

values
Return values from underlying reference

openpyxl.charts.graph module

class openpyxl.charts.graph.**GraphChart**(*auto_axis=False*)
Bases: *openpyxl.charts.chart.Chart*

Chart with axes

compute_axes()
Calculate maximum value and units for axes

get_x_units()
calculate one unit for x axis in EMU

get_y_units()
calculate one unit for y axis in EMU

x_axis
alias of CategoryAxis

y_axis
alias of ValueAxis

openpyxl.charts.legend module

class openpyxl.charts.legend.**Legend**
Bases: object

openpyxl.charts.line module

class openpyxl.charts.line.**LineChart**(*auto_axis=False*)
Bases: *openpyxl.charts.graph.GraphChart*

TYPE = 'lineChart'

openpyxl.charts.pie module

class openpyxl.charts.pie.**PieChart**
Bases: *openpyxl.charts.chart.Chart*

TYPE = 'pieChart'

openpyxl.charts.reference module

class openpyxl.charts.reference.**Reference**(*sheet*, *pos1*, *pos2=None*, *data_type=None*, *number_format=None*)
Bases: *openpyxl.descriptors.Strict*

a simple wrapper around a serie of reference data

data_type
'none' will be treated as None

number_format

pos1

pos2

values
read data in sheet - to be used at writing time

openpyxl.charts.scatter module

class openpyxl.charts.scatter.**ScatterChart**(*auto_axis=False*)
Bases: *openpyxl.charts.graph.GraphChart*

TYPE = 'scatterChart'

openpyxl.charts.series module

`openpyxl.charts.series.`**`Serie`**
alias of *`Series`*

class `openpyxl.charts.series.`**`Series`**(*values*, *title=None*, *labels=None*, *color=None*, *xvalues=None*, *legend=None*)
Bases: `object`

a serie of data and possibly associated labels

`MARKER_NONE` **= 'none'**

`color`

`get_min_max`()
Legacy method. Replaced by properties

`labels`
Return values from reference set as label

`legend`

`max`(*attr='values'*)
Return the maximum value for numeric series. NB None has a value of u" which is ignored

`min`(*attr='values'*)
Return the minimum value for numeric series NB None has a value of u" which is ignored

`title`

`values`
Return values from underlying reference

`xvalues`
Return xvalues

openpyxl.charts.writer module

class `openpyxl.charts.writer.`**`BarChartWriter`**(*chart*)
Bases: *`openpyxl.charts.writer.LineChartWriter`*

class `openpyxl.charts.writer.`**`BaseChartWriter`**(*chart*)
Bases: `object`

`series_type` **= '{http://schemas.openxmlformats.org/drawingml/2006/chart}val'**

`write`()
write a chart

`write_rels`(*drawing_id*)

class `openpyxl.charts.writer.`**`ChartWriter`**(*chart*)
Bases: `object`

Preserve interface for chart writer

`write`()

class `openpyxl.charts.writer.`**`LineChartWriter`**(*chart*)
Bases: *`openpyxl.charts.writer.BaseChartWriter`*

class `openpyxl.charts.writer.`**`PieChartWriter`**(*chart*)
Bases: *`openpyxl.charts.writer.BaseChartWriter`*

class `openpyxl.charts.writer.`**`ScatterChartWriter`**(*chart*)
Bases: *`openpyxl.charts.writer.LineChartWriter`*

`series_type` **= '{http://schemas.openxmlformats.org/drawingml/2006/chart}yVal'**

Module contents

openpyxl.comments package

Submodules

openpyxl.comments.comments module

class `openpyxl.comments.comments.`**`Comment`**(*text*, *author*)
Bases: `object`

`author`
The name recorded for the author

Return type string

`parent`

`text`
The text of the commment

Return type string

Module contents

openpyxl.descriptors package

Submodules

openpyxl.descriptors.base module

class `openpyxl.descriptors.base.`**`ASCII`**(*name=None*, ***kw*)
Bases: *`openpyxl.descriptors.base.Typed`*

`expected_type`
alias of `str`

class `openpyxl.descriptors.base.`**`Alias`**(*alias*)
Bases: *`openpyxl.descriptors.base.Descriptor`*

Aliases can be used when either the desired attribute name is not allowed or confusing in Python (eg. "type") or a more descriptve name is desired (eg. "underline" for "u")

class `openpyxl.descriptors.base.`**`Bool`**(*name=None*, ***kw*)
Bases: *`openpyxl.descriptors.base.Convertible`*

`expected_type`
alias of `bool`

class `openpyxl.descriptors.base.`**`Convertible`**(*name=None*, ***kw*)
Bases: *`openpyxl.descriptors.base.Typed`*

Values must be convertible to a particular type

class `openpyxl.descriptors.base.`**`Default`**(*name=None*, ***kw*)
Bases: *`openpyxl.descriptors.base.Typed`*

When called returns an instance of the expected type. Additional default values can be passed in to the descriptor

class `openpyxl.descriptors.base.`**`Descriptor`**(*name=None*, ***kw*)
Bases: `object`

class `openpyxl.descriptors.base.`**`Float`**(*name=None*, ***kw*)
Bases: *`openpyxl.descriptors.base.Convertible`*

`expected_type`
alias of `float`

class `openpyxl.descriptors.base.`**`Integer`**(*name=None*, ***kw*)
Bases: *`openpyxl.descriptors.base.Convertible`*

`expected_type`
alias of `long`

class `openpyxl.descriptors.base.`**`Length`**(*name=None*, ***kw*)
Bases: *`openpyxl.descriptors.base.Descriptor`*

class `openpyxl.descriptors.base.`**`MatchPattern`**(*name=None*, ***kw*)
Bases: *`openpyxl.descriptors.base.Descriptor`*

Values must match a regex pattern

`allow_none` **= False**

class `openpyxl.descriptors.base.`**`Max`**(*name=None*, ***kw*)
Bases: *`openpyxl.descriptors.base.Typed`*

Values must be less than a *max* value

`expected_type`
alias of `float`

class `openpyxl.descriptors.base.`**`Min`**(*name=None*, ***kw*)
Bases: *`openpyxl.descriptors.base.Typed`*

Values must be greater than a *min* value

`expected_type`
alias of `float`

class `openpyxl.descriptors.base.`**`MinMax`**(*name=None*, ***kw*)
Bases: *`openpyxl.descriptors.base.Min`*, *`openpyxl.descriptors.base.Max`*

Values must be greater than *min* value and less than a *max* one

class `openpyxl.descriptors.base.`**`NoneSet`**(*name=None*, ***kw*)
Bases: *`openpyxl.descriptors.base.Set`*

‘none’ will be treated as None

class `openpyxl.descriptors.base.`**`Sequence`**(*name=None*, ***kw*)
Bases: *`openpyxl.descriptors.base.Descriptor`*

A sequence (list or tuple) that may only contain objects of the declared type

expected_type
alias of NoneType

seq_types = (<type 'list'>, <type 'tuple'>)

class openpyxl.descriptors.base.**Set**(*name=None*, ***kw*)
Bases: *openpyxl.descriptors.base.Descriptor*

Value can only be from a set of know values

class openpyxl.descriptors.base.**String**(*name=None*, ***kw*)
Bases: *openpyxl.descriptors.base.Typed*

expected_type
alias of basestring

class openpyxl.descriptors.base.**Tuple**(*name=None*, ***kw*)
Bases: *openpyxl.descriptors.base.Typed*

expected_type
alias of tuple

class openpyxl.descriptors.base.**Typed**(*name=None*, ***kw*)
Bases: *openpyxl.descriptors.base.Descriptor*

Values must of a particular type

allow_none = False

expected_type
alias of NoneType

nested = False

openpyxl.descriptors.excel module

class openpyxl.descriptors.excel.**HexBinary**(*name=None*, ***kw*)
Bases: *openpyxl.descriptors.base.MatchPattern*

pattern = '[0-9a-fA-F]+$'

class openpyxl.descriptors.excel.**Percentage**(*name=None*, ***kw*)
Bases: *openpyxl.descriptors.base.MatchPattern*

pattern = '((100)|([0-9][0-9]?))(\\.[0-9][0-9]?)?%'

class openpyxl.descriptors.excel.**TextPoint**(*name=None*, ***kw*)
Bases: *openpyxl.descriptors.base.MinMax*

Size in hundredths of points. In theory other units of measurement can be used but these are unbounded

expected_type
alias of int

max = 400000

min = -400000

class openpyxl.descriptors.excel.**UniversalMeasure**(*name=None*, ***kw*)
Bases: *openpyxl.descriptors.base.MatchPattern*

pattern = '[0-9]+(\\.[0-9]+)?(mm|cm|in|pt|pc|pi)'

openpyxl.descriptors.serialisable module

class `openpyxl.descriptors.serialisable.`**`Serialisable`**

Bases: `openpyxl.descriptors._Serialisable`

Objects can serialise to XML their attributes and child objects. The following class attributes are created by the metaclass at runtime: __attrs__ = attributes __nested__ = single-valued child treated as an attribute __elements__ = child elements

classmethod **`from_tree`**(*node*)

Create object from XML

`tagname`

`to_tree`(*tagname=None*)

Module contents

class `openpyxl.descriptors.`**`MetaSerialisable`**

Bases: `type`

class `openpyxl.descriptors.`**`MetaStrict`**

Bases: `type`

class `openpyxl.descriptors.`**`Strict`**

Bases: `object`

openpyxl.drawing package

Submodules

openpyxl.drawing.drawing module

class `openpyxl.drawing.drawing.`**`Drawing`**

Bases: `object`

a drawing object - eg container for shapes or charts we assume user specifies dimensions in pixels; units are converted to EMU in the drawing part

`count` **= 0**

`get_emu_dimensions`()

return (x, y, w, h) in EMU

`height`

`set_dimension`(*w=0*, *h=0*)

`width`

class `openpyxl.drawing.drawing.`**`Image`**(*img*, *coordinates=((0, 0), (1, 1))*, *size=(None, None)*, *nochangeaspect=True*, *nochangearrowheads=True*)

Bases: `object`

Raw Image class

`anchor`(*cell*, *anchortype='absolute'*)

anchors the image to the given cell optional parameter anchortype supports 'absolute' or 'oneCell'

class `openpyxl.drawing.drawing.`**`Shadow`**

Bases: `object`

`SHADOW_BOTTOM` **= 'b'**

`SHADOW_BOTTOM_LEFT` **= 'bl'**

`SHADOW_BOTTOM_RIGHT` **= 'br'**

`SHADOW_CENTER` **= 'ctr'**

`SHADOW_LEFT` **= 'l'**

`SHADOW_TOP` **= 't'**

`SHADOW_TOP_LEFT` **= 'tl'**

`SHADOW_TOP_RIGHT` **= 'tr'**

class `openpyxl.drawing.drawing.`**`Shape`**(*chart*, *coordinates=((0, 0), (1, 1))*, *text=None*, *scheme='accent1'*)

Bases: `object`

a drawing inside a chart coordiantes are specified by the user in the axis units

`FONT_HEIGHT` **= 8**

`FONT_WIDTH` **= 7**

`MARGIN_BOTTOM` **= 28**

`MARGIN_LEFT` **= 20**

`RECT` **= 'rect'**

"line" "lineInv" "triangle" "rtTriangle" "diamond" "parallelogram" "trapezoid" "nonIsoscelesTrapezoid" "pentagon" "hexagon" "heptagon" "octagon" "decagon" "dodecagon" "star4" "star5" "star6" "star7" "star8" "star10" "star12" "star16" "star24" "star32" "roundRect" "round1Rect" "round2SameRect" "round2DiagRect" "snipRoundRect" "snip1Rect" "snip2SameRect" "snip2DiagRect" "plaque" "ellipse" "teardrop" "homePlate" "chevron" "pieWedge" "pie" "blockArc" "donut" "noSmoking" "rightArrow" "leftArrow" "upArrow" "downArrow" "stripedRightArrow" "notchedRightArrow" "bentUpArrow" "leftRightArrow" "upDownArrow" "leftUpArrow" "leftRightUpArrow" "quadArrow" "leftArrowCallout" "rightArrowCallout" "upArrowCallout" "downArrowCallout" "leftRightArrowCallout" "upDownArrowCallout" "quadArrowCallout" "bentArrow" "uturnArrow" "circularArrow" "leftCircularArrow" "leftRightCircularArrow" "curvedRightArrow" "curvedLeftArrow" "curvedUpArrow" "curvedDownArrow" "swooshArrow" "cube" "can" "lightningBolt" "heart" "sun" "moon" "smileyFace" "irregularSeal1" "irregularSeal2" "foldedCorner" "bevel" "frame" "halfFrame" "corner" "diagStripe" "chord" "arc" "leftBracket" "rightBracket" "leftBrace" "rightBrace" "bracketPair" "bracePair" "straightConnector1" "bentConnector2" "bentConnector3" "bentConnector4" "bentConnector5" "curvedConnector2" "curvedConnector3" "curvedConnector4" "curvedConnector5" "callout1" "callout2" "callout3" "accentCallout1" "accentCallout2" "accentCallout3" "borderCallout1" "borderCallout2" "borderCallout3" "accentBorderCallout1" "accentBorderCallout2" "accentBorderCallout3" "wedgeRectCallout" "wedgeRoundRectCallout" "wedgeEllipseCallout" "cloudCallout" "cloud" "ribbon" "ribbon2" "ellipseRibbon" "ellipseRibbon2" "leftRightRibbon" "verticalScroll" "horizontalScroll" "wave" "doubleWave" "plus" "flowChartProcess" "flowChartDecision" "flowChartInputOutput" "flowChartPredefinedProcess" "flowChartInternalStorage" "flowChartDocument" "flowChartMultidocument" "flowChartTerminator" "flowChartPreparation" "flowChartManualInput" "flowChartManualOperation" "flowChartConnector" "flowChartPunchedCard" "flowChartPunchedTape" "flowChartSummingJunction" "flowChartOr" "flowChartCollate" "flowChartSort" "flowChartExtract" "flowChartMerge" "flowChartOfflineStorage" "flowChartOnlineStorage" "flowChartMagneticTape" "flowChartMagneticDisk" "flowChartMagneticDrum" "flowChartDisplay" "flowChartDelay" "flowChartAlternateProcess" "flowChartOffpageConnector" "actionButtonBlank" "actionButtonHome" "actionButtonHelp" "actionButtonInformation" "actionButtonForwardNext" "actionButtonBackPrevious" "actionButtonEnd" "actionButtonBeginning" "actionButtonReturn" "actionButtonDocument" "actionButtonSound" "actionButtonMovie" "gear6" "gear9" "funnel" "mathPlus" "mathMinus" "mathMultiply" "mathDivide" "mathEqual" "mathNotEqual" "cornerTabs" "squareTabs" "plaqueTabs" "chartX" "chartStar" "chartPlus"

ROUND_RECT = 'roundRect'

border_color

border_width

color

coordinates
Return coordindates in axis units

text_color

openpyxl.drawing.drawing.**bounding_box**(*bw*, *bh*, *w*, *h*)
Returns a tuple (new_width, new_height) which has the property that it fits within box_width and box_height and has (close to) the same aspect ratio as the original size

Module contents

openpyxl.formatting package

Submodules

openpyxl.formatting.formatting module

class openpyxl.formatting.formatting.**ConditionalFormatting**
Bases: object

add(*range_string*, *rule*)

openpyxl.formatting.rule module

class openpyxl.formatting.rule.**ColorScale**(*cfvo=None*, *color=None*)
Bases: *openpyxl.formatting.rule.RuleType*

color
A sequence (list or tuple) that may only contain objects of the declared type

tagname = 'colorScale'

class openpyxl.formatting.rule.**DataBar**(*minLength=None*, *maxLength=None*, *showValue=None*, *cfvo=None*, *color=None*)
Bases: *openpyxl.formatting.rule.RuleType*

color

maxLength

minLength

showValue

tagname = 'dataBar'

class openpyxl.formatting.rule.**ExtensionList**
Bases: *openpyxl.descriptors.serialisable.Serialisable*

class `openpyxl.formatting.rule.`**`FormatObject`**(*type*, *val=None*, *gte=None*, *extLst=None*)

Bases: `openpyxl.descriptors.serialisable.Serialisable`

`extLst`
Values must of a particular type

`gte`

`tagname` = **'cfvo'**

`type`
Value can only be from a set of know values

`val`

class `openpyxl.formatting.rule.`**`IconSet`**(*iconSet=None*, *showValue=None*, *percent=None*, *reverse=None*, *cfvo=None*)

Bases: `openpyxl.formatting.rule.RuleType`

`iconSet`
'none' will be treated as None

`percent`

`reverse`

`showValue`

`tagname` = **'iconSet'**

class `openpyxl.formatting.rule.`**`Rule`**(*type*, *dxfId=None*, *priority=None*, *stopIfTrue=None*, *aboveAverage=None*, *percent=None*, *bottom=None*, *operator=None*, *text=None*, *timePeriod=None*, *rank=None*, *stdDev=None*, *equalAverage=None*, *formula=[]*, *colorScale=None*, *dataBar=None*, *iconSet=None*, *extLst=None*, *style=None*)

Bases: `openpyxl.descriptors.serialisable.Serialisable`

`aboveAverage`

`bottom`

`colorScale`
Values must of a particular type

`dataBar`
Values must of a particular type

`dxfId`

`equalAverage`

`extLst`
Values must of a particular type

`formula`
A sequence (list or tuple) that may only contain objects of the declared type

`iconSet`
Values must of a particular type

`operator`
'none' will be treated as None

`percent`

priority

rank

stdDev

stopIfTrue

style
Values must of a particular type

tagname = 'cfRule'

text

timePeriod
'none' will be treated as None

type
Value can only be from a set of know values

class `openpyxl.formatting.rule.`**RuleType**
Bases: *openpyxl.descriptors.serialisable.Serialisable*

cfvo
A sequence (list or tuple) that may only contain objects of the declared type

openpyxl.formatting.rules module

class `openpyxl.formatting.rules.`**CellIsRule**(*operator=None*, *formula=None*, *stopIfTrue=None*, *font=None*, *border=None*, *fill=None*)
Bases: `object`

Conditional formatting rule based on cell contents.

expand = {'>=': 'greaterThanOrEqual', '==': 'equal', '!=': 'notEqual', '<=': 'lessThanOrEqual', '=': 'equal', '<': 'lessT

operator

rule

class `openpyxl.formatting.rules.`**ColorScaleRule**(*start_type=None*, *start_value=None*, *start_color=None*, *mid_type=None*, *mid_value=None*, *mid_color=None*, *end_type=None*, *end_value=None*, *end_color=None*)
Bases: `object`

Conditional formatting rule based on a color scale rule.

attrs = ('type', 'colorScale')

cfvo
Return a dictionary representation

colorScale

end_value

mid_value

rule

start_value

type = 'colorScale'

valid_types = ('min', 'max', 'num', 'percent', 'percentile', 'formula')

class openpyxl.formatting.rules.**FormatRule**

Bases: `_abcoll.Mapping`

Utility dictionary for formatting rules with specified keys only

aboveAverage

bottom

colorScale

dxfId

equalAverage

formula

iconSet

items()

iteritems()

iterkeys()

itervalues()

keys()

operator

percent

priority

rank

stdDev

stopIfTrue

text

type

update(*dictionary*)

values()

class openpyxl.formatting.rules.**FormulaRule**(*formula=None*, *stopIfTrue=None*, *font=None*, *border=None*, *fill=None*)

Bases: `object`

Conditional formatting rule based on a formula.

rule

Module contents

class openpyxl.formatting.**ConditionalFormatting**

Bases: `object`

Conditional formatting rules.

add (*range_string*, *cfRule*)
Add a rule. Rule is either: 1. A dictionary containing a key called type, and other keys, as in *ConditionalFormatting.rule_attributes*. 2. A rule object, such as ColorScaleRule, FormulaRule or CellIsRule

The priority will be added automatically.

icon_attributes = ('iconSet', 'showValue', 'reverse')

rule_attributes = ('aboveAverage', 'bottom', 'dxfId', 'equalAverage', 'operator', 'percent', 'priority', 'rank', 'stdDe

setDxfStyles (**args*, ***kwargs*)

update (*cfRules*)
Set the conditional formatting rules from a dictionary. Intended for use when loading a document. cfRules use the structure: {range_string: [rule1, rule2]}, eg: {'A1:A4': [{'type': 'colorScale', 'priority': 13, 'colorScale': {'cfvo': [{'type': 'min'}, {'type': 'max'}], 'color': [Color('FFFF7128'), Color('FFFFEF9C')]}]}

openpyxl.formatting.**unpack_rules** (*cfRules*)

openpyxl.reader package

Submodules

openpyxl.reader.comments module

openpyxl.reader.comments.**get_comments_file** (*worksheet_path*, *archive*, *valid_files*)
Returns the XML filename in the archive which contains the comments for the spreadsheet with codename sheet_codename. Returns None if there is no such file

openpyxl.reader.comments.**read_comments** (*ws*, *xml_source*)
Given a worksheet and the XML of its comments file, assigns comments to cells

openpyxl.reader.excel module

openpyxl.reader.excel.**load_workbook** (*filename*, *read_only=False*, *use_iterators=False*, *keep_vba=False*, *guess_types=False*, *data_only=False*)
Open the given filename and return the workbook

Parameters

- **filename** (string or a file-like object open in binary mode c.f., `zipfile.ZipFile`) – the path to open or a file-like object
- **read_only** (*bool*) – optimised for reading, content cannot be edited
- **use_iterators** (*bool*) – use lazy load for cells
- **keep_vba** (*bool*) – preseve vba content (this does NOT mean you can use it)
- **guess_types** (*bool*) – guess cell content type and do not read it from the file
- **data_only** (*bool*) – controls whether cells with formulae have either the formula (default) or the value stored the last time Excel read the sheet

Return type `openpyxl.workbook.Workbook`

Note: When using lazy load, all worksheets will be *`openpyxl.worksheet.iter_worksheet.IterableWorksheet`* and the returned workbook will be read-only.

`openpyxl.reader.excel.`**`repair_central_directory`**`(`*zipFile*, *is_file_instance*`)`
trims trailing data from the central directory code taken from http://stackoverflow.com/a/7457686/570216, courtesy of Uri Cohen

openpyxl.reader.strings module

`openpyxl.reader.strings.`**`get_string`**`(`*string_index_node*`)`
Read the contents of a specific string index

`openpyxl.reader.strings.`**`get_text`**`(`*rich_node*`)`
Read rich text, discarding formatting if not disallowed

`openpyxl.reader.strings.`**`read_string_table`**`(`*xml_source*`)`
Read in all shared strings in the table

openpyxl.reader.style module

class `openpyxl.reader.style.`**`SharedStylesParser`**`(`*xml_source*`)`
Bases: `object`

`parse``()`

`parse_borders``()`
Read in the boarders

`parse_cell_styles``()`
Extract individual cell styles

`parse_color_index``()`
Read in the list of indexed colors

`parse_custom_num_formats``()`
Read in custom numeric formatting rules from the shared style table

`parse_dxfs``()`
Read in the dxfs effects - used by conditional formatting.

`parse_fills``()`
Read in the list of fills

`parse_fonts``()`
Read in the fonts

`parse_named_styles``()`
Extract named styles

`openpyxl.reader.style.`**`bool_attrib`**`(`*element*, *attr*`)`
Cast an XML attribute that should be a boolean to a Python equivalent None, ‘f’, ‘0’ and ‘false’ all cast to False, everything else to true

`openpyxl.reader.style.`**`read_style_table`**`(`*archive*`)`

openpyxl.reader.workbook module

`openpyxl.reader.workbook.`**`detect_external_links`**`(`*archive*`)`

`openpyxl.reader.workbook.`**`detect_worksheets`**`(`*archive*`)`
Return a list of worksheets

`openpyxl.reader.workbook.`**`read_content_types`**(*archive*)
    Read content types.

`openpyxl.reader.workbook.`**`read_excel_base_date`**(*archive*)

`openpyxl.reader.workbook.`**`read_rels`**(*archive*)
    Read relationships for a workbook

`openpyxl.reader.workbook.`**`read_sheets`**(*archive*)
    Read worksheet titles and ids for a workbook

`openpyxl.reader.workbook.`**`read_workbook_code_name`**(*xml_source*)

openpyxl.reader.worksheet module

class `openpyxl.reader.worksheet.`**`WorkSheetParser`**(*ws*, *xml_source*, *shared_strings*, *style_table*, *color_index=None*)

Bases: `object`

`CELL_TAG` **= '{http://schemas.openxmlformats.org/spreadsheetml/2006/main}c'**

`COL_TAG` **= '{http://schemas.openxmlformats.org/spreadsheetml/2006/main}col'**

`FORMULA_TAG` **= '{http://schemas.openxmlformats.org/spreadsheetml/2006/main}f'**

`INLINE_RICHTEXT` **= '{http://schemas.openxmlformats.org/spreadsheetml/2006/main}is/{http://schemas.openxmlformat**

`INLINE_STRING` **= '{http://schemas.openxmlformats.org/spreadsheetml/2006/main}is/{http://schemas.openxmlformats.o**

`MERGE_TAG` **= '{http://schemas.openxmlformats.org/spreadsheetml/2006/main}mergeCell'**

`ROW_TAG` **= '{http://schemas.openxmlformats.org/spreadsheetml/2006/main}row'**

`VALUE_TAG` **= '{http://schemas.openxmlformats.org/spreadsheetml/2006/main}v'**

`parse`()

`parse_auto_filter`(*element*)

`parse_cell`(*element*)

`parse_column_dimensions`(*col*)

`parse_data_validation`(*element*)

`parse_header_footer`(*element*)

`parse_legacy_drawing`(*element*)

`parse_margins`(*element*)

`parse_merge`(*element*)

`parse_page_setup`(*element*)

`parse_print_options`(*element*)

`parse_properties`(*element*)

`parse_row_dimensions`(*row*)

`parse_sheet_protection`(*element*)

`parse_sheet_views`(*element*)

`parser_conditional_formatting`(*element*) `openpyxl.reader.worksheet.`**`fast_parse`**(*ws*, *xml_source*, *shared_strings*, *style_table*, *color_index=None*, *keep_vba=False*)

`openpyxl.reader.worksheet.`**`read_worksheet`**(*xml_source*, *parent*, *preset_title*, *shared_strings*, *style_table*, *color_index=None*, *worksheet_path=None*)

Read an xml worksheet

Module contents

openpyxl.styles package

Submodules

openpyxl.styles.alignment module

class `openpyxl.styles.alignment.`**`Alignment`**(*horizontal=None*, *vertical=None*, *textRotation=0*, *wrapText=None*, *shrinkToFit=None*, *indent=0*, *relativeIndent=0*, *justifyLastLine=None*, *readingOrder=0*, *text_rotation=None*, *wrap_text=None*, *shrink_to_fit=None*)

Bases: *`openpyxl.styles.hashable.HashableObject`*

Alignment options for use in styles.

`horizontal`
‘none’ will be treated as None

`indent`
Values must be greater than a *min* value

`justifyLastLine`

`readingOrder`
Values must be greater than a *min* value

`relativeIndent`
Values must be greater than a *min* value

`shrinkToFit`

`tagname` **= ‘alignment’**

`textRotation`
‘none’ will be treated as None

`vertical`
‘none’ will be treated as None

`wrapText`

openpyxl.styles.borders module

class `openpyxl.styles.borders.`**`Border`**(*left=*, *right=*, *top=*, *bottom=*, *diagonal=*, *diagonal_direction=None*, *vertical=None*, *horizontal=None*, *diagonalUp=False*, *diagonalDown=False*, *outline=True*, *start=None*, *end=None*)

Bases: *`openpyxl.styles.hashable.HashableObject`*

Border positioning for use in styles.

bottom
Values must of a particular type

diagonal
Values must of a particular type

diagonalDown

diagonalUp

end
Values must of a particular type

horizontal
Values must of a particular type

left
Values must of a particular type

outline

right
Values must of a particular type

start
Values must of a particular type

tagname = 'border'

top
Values must of a particular type

vertical
Values must of a particular type

class openpyxl.styles.borders.**Side**(*style=None*, *color=None*, *border_style=None*)
Bases: *openpyxl.styles.hashable.HashableObject*

Border options for use in styles. Caution: if you do not specify a border_style, other attributes will have no effect !

color

style
'none' will be treated as None

openpyxl.styles.colors module

class openpyxl.styles.colors.**Color**(*rgb='00000000'*, *indexed=None*, *auto=None*, *theme=None*, *tint=0.0*, *index=None*, *type='rgb'*)
Bases: *openpyxl.styles.hashable.HashableObject*

Named colors for use in styles.

auto

index

indexed

rgb
Descriptor for aRGB values If not supplied alpha is 00

tagname = 'color'

theme

tint
Values must be greater than *min* value and less than a *max* one

type

value

class `openpyxl.styles.colors.`**ColorDescriptor**(*name=None*, ***kw*)
Bases: *openpyxl.descriptors.base.Typed*

expected_type
alias of *Color*

class `openpyxl.styles.colors.`**RGB**(*name=None*, ***kw*)
Bases: *openpyxl.descriptors.base.Typed*

Descriptor for aRGB values If not supplied alpha is 00

expected_type
alias of `basestring`

openpyxl.styles.differential module

class `openpyxl.styles.differential.`**DifferentialStyle**(*font=None*, *numFmt=None*, *fill=None*, *alignment=None*, *border=None*, *protection=None*, *extLst=None*)
Bases: *openpyxl.descriptors.serialisable.Serialisable*

alignment
Values must of a particular type

border
Values must of a particular type

fill
Values must of a particular type

font
Values must of a particular type

numFmt
Values must of a particular type

protection
Values must of a particular type

tagname = 'dxf'

class `openpyxl.styles.differential.`**NumFmt**(*numFmtId=None*, *formatCode=None*)
Bases: *openpyxl.descriptors.serialisable.Serialisable*

formatCode

numFmtId

openpyxl.styles.fills module

class `openpyxl.styles.fills.`**`Fill`**

Bases: `openpyxl.styles.hashable.HashableObject`

Base class

classmethod **`from_tree`**(*el*)

`tagname` = **‘fill’**

class `openpyxl.styles.fills.`**`GradientFill`**(*type=’linear’*, *degree=0*, *left=0*, *right=0*, *top=0*, *bottom=0*, *stop=()*, *fill_type=None*)

Bases: `openpyxl.styles.fills.Fill`

`bottom`

`degree`

`left`

`right`

`stop`

A sequence (list or tuple) that may only contain objects of the declared type

`tagname` = **‘gradientFill’**

`to_tree`(*tagname=None*)

`top`

`type`

Value can only be from a set of know values

class `openpyxl.styles.fills.`**`PatternFill`**(*patternType=None*, *fgColor=Color(indexed=Value must be type ‘long’*, *auto=Value must be type ‘bool’*, *theme=Value must be type ‘long’)*, *bgColor=Color(indexed=Value must be type ‘long’*, *auto=Value must be type ‘bool’*, *theme=Value must be type ‘long’)*, *fill_type=None*, *start_color=None*, *end_color=None*)

Bases: `openpyxl.styles.fills.Fill`

Area fill patterns for use in styles. Caution: if you do not specify a fill_type, other attributes will have no effect !

`bgColor`

`fgColor`

`patternType`

‘none’ will be treated as None

`tagname` = **‘patternFill’**

`to_tree`(*tagname=None*)

openpyxl.styles.fonts module

class `openpyxl.styles.fonts.`**`Font`**(*name='Calibri'*, *sz=11*, *b=False*, *i=False*, *charset=None*, *u=None*, *strike=False*, *color='00000000'*, *scheme=None*, *family=2*, *size=None*, *bold=None*, *italic=None*, *strikethrough=None*, *underline=None*, *vertAlign=None*, *outline=False*, *shadow=False*, *condense=False*, *extend=False*)

Bases: *`openpyxl.styles.hashable.HashableObject`*

Font options used in styles.

`UNDERLINE_DOUBLE` = **'double'**

`UNDERLINE_DOUBLE_ACCOUNTING` = **'doubleAccounting'**

`UNDERLINE_SINGLE` = **'single'**

`UNDERLINE_SINGLE_ACCOUNTING` = **'singleAccounting'**

`b`

`charset`

`color`

`condense`

`extend`

`family`
Values must be greater than *min* value and less than a *max* one

`i`

`name`

`outline`

`scheme`
'none' will be treated as None

`shadow`

`spec` = **'18.8.22, p.3930'**

`strike`

`sz`

`tagname` = **'font'**

`to_tree`(*tagname=None*)

`u`
'none' will be treated as None

`vertAlign`
'none' will be treated as None

openpyxl.styles.hashable module

class `openpyxl.styles.hashable.`**`HashableObject`**

Bases: *`openpyxl.descriptors.serialisable.Serialisable`*

Define how to hash property classes.

copy (***kwargs*)

key
Use a tuple of fields as the basis for a key

openpyxl.styles.named_styles module

class openpyxl.styles.named_styles.**NamedStyle** (*name, font=Font(color=Color(indexed=Value must be type 'long', auto=Value must be type 'bool', theme=Value must be type 'long')), fill=, border=, alignment=, number_format=None, protection=*)
Bases: *openpyxl.descriptors.Strict*

alignment
Values must of a particular type

border
Values must of a particular type

fill
Values must of a particular type

font
Values must of a particular type

number_format

protection
Values must of a particular type

tag = '{http://schemas.openxmlformats.org/spreadsheetml/2006/main}cellStyleXfs'
Named and editable styles

openpyxl.styles.numbers module

openpyxl.styles.numbers.**NumberFormat** (**args, **kwargs*)
Numer formatting for use in styles.

class openpyxl.styles.numbers.**NumberFormatDescriptor** (*name=None, **kw*)
Bases: *openpyxl.descriptors.base.String*

openpyxl.styles.numbers.**builtin_format_code** (*index*)
Return one of the standard format codes by index.

openpyxl.styles.numbers.**builtin_format_id** (*fmt*)
Return the id of a standard style.

openpyxl.styles.numbers.**is_builtin** (*fmt*)

openpyxl.styles.numbers.**is_date_format** (*fmt*)

openpyxl.styles.protection module

class openpyxl.styles.protection.**Protection** (*locked=True, hidden=False*)
Bases: *openpyxl.styles.hashable.HashableObject*

Protection options for use in styles.

hidden

locked

tagname = **'protection'**

openpyxl.styles.proxy module

class `openpyxl.styles.proxy.`**`StyleProxy`**(*target*)

Bases: `object`

Proxy formatting objects so that they cannot be altered

copy(***kw*)

Return a copy of the proxied object. Keyword args will be passed through

openpyxl.styles.style module

class `openpyxl.styles.style.`**`StyleId`**(*numFmtId=0*, *fontId=0*, *fillId=0*, *borderId=0*, *alignmentId=0*, *protectionId=0*, *xfId=0*, *quotePrefix=None*, *pivotButton=None*, *applyNumberFormat=None*, *applyFont=None*, *applyFill=None*, *applyBorder=None*, *applyAlignment=None*, *applyProtection=None*, *extLst=None*)

Bases: `openpyxl.descriptors.serialisable.Serialisable`

Format aggregation class

This is a virtual style composed of references to global format objects

alignmentId

applyAlignment

applyProtection

borderId

fillId

fontId

numFmtId

pivotButton

protectionId

quotePrefix

tagname = **'xf'**

to_tree()

Alignment and protection objects are implemented as child elements. This is a completely different API to other format objects. :-/

xfId

openpyxl.styles.styleable module

class `openpyxl.styles.styleable.`**`NumberFormatDescriptor`**

Bases: `object`

collection = '_number_formats'

key = '_number_format_id'

class openpyxl.styles.styleable.**StyleDescriptor**(*collection*, *key*)
Bases: object

class openpyxl.styles.styleable.**StyleableObject**(*sheet*, *fontId=0*, *fillId=0*, *borderId=0*, *alignmentId=0*, *protectionId=0*, *numFmtId=0*, *pivotButton=None*, *quotePrefix=None*)
Bases: object

Base class for styleble objects implementing proxy and lookup functions

has_style

parent

pivotButton

quotePrefix

style

style_id

Module contents

class openpyxl.styles.**Style**(*font=Font(color=Color(indexed=Value must be type 'long', auto=Value must be type 'bool', theme=Value must be type 'long')), fill=, border=, alignment=, number_format=None, protection=*)
Bases: *openpyxl.styles.hashable.HashableObject*

Style object containing all formatting details.

alignment
Values must of a particular type

border
Values must of a particular type

copy(**args*, ***kwargs*)

fill
Values must of a particular type

font
Values must of a particular type

number_format

protection
Values must of a particular type

openpyxl.utils package

Submodules

openpyxl.utils.datetime module

`openpyxl.utils.datetime.`**`W3CDTF_to_datetime`**(*formatted_string*)
Convert from a timestamp string to a datetime object.

`openpyxl.utils.datetime.`**`datetime_to_W3CDTF`**(*dt*)
Convert from a datetime to a timestamp string.

`openpyxl.utils.datetime.`**`days_to_time`**(**args*, ***kwds*)

`openpyxl.utils.datetime.`**`from_excel`**(**args*, ***kwds*)

`openpyxl.utils.datetime.`**`time_to_days`**(**args*, ***kwds*)
Convert a time value to fractions of day

`openpyxl.utils.datetime.`**`timedelta_to_days`**(**args*, ***kwds*)
Convert a timedelta value to fractions of a day

`openpyxl.utils.datetime.`**`to_excel`**(**args*, ***kwds*)

openpyxl.utils.exceptions module

exception `openpyxl.utils.exceptions.`**`CellCoordinatesException`**
Bases: `exceptions.Exception`

Error for converting between numeric and A1-style cell references.

exception `openpyxl.utils.exceptions.`**`IllegalCharacterError`**
Bases: `exceptions.Exception`

The data submitted which cannot be used directly in Excel files. It must be removed or escaped.

exception `openpyxl.utils.exceptions.`**`InsufficientCoordinatesException`**
Bases: `exceptions.Exception`

Error for partially specified cell coordinates.

exception `openpyxl.utils.exceptions.`**`InvalidFileException`**
Bases: `exceptions.Exception`

Error for trying to open a non-ooxml file.

exception `openpyxl.utils.exceptions.`**`NamedRangeException`**
Bases: `exceptions.Exception`

Error for badly formatted named ranges.

exception `openpyxl.utils.exceptions.`**`ReadOnlyWorkbookException`**
Bases: `exceptions.Exception`

Error for trying to modify a read-only workbook

exception `openpyxl.utils.exceptions.`**`SheetTitleException`**
Bases: `exceptions.Exception`

Error for bad sheet names.

exception `openpyxl.utils.exceptions.`**`WorkbookAlreadySaved`**

Bases: `exceptions.Exception`

Error when attempting to perform operations on a dump workbook while it has already been dumped once

openpyxl.utils.indexed_list module

class `openpyxl.utils.indexed_list.`**`IndexedList`**(*iterable=None*)

Bases: `list`

List with optimised access by value Based on Alex Martelli's recipe

http://code.activestate.com/recipes/52303-the-auxiliary-dictionary-idiom-for-sequences-with-/

`add`(*value*)

`append`(*value*)

`index`(*value*)

openpyxl.utils.units module

`openpyxl.utils.units.`**`DEFAULT_HEADER`** **= 0.3**

From the ECMA Spec (4th Edition part 1) Page setup: "Left Page Margin in inches" p. 1647

Docs from http://startbigthinksmall.wordpress.com/2010/01/04/points-inches-and-emus-measuring-units-in-office-open-xml/

See also http://msdn.microsoft.com/en-us/library/dd560821(v=office.12).aspx

dxa: The main unit in OOXML is a twentieth of a point. Also called twips. pt: point. In Excel there are 72 points to an inch hp: half-points are used to specify font sizes. A font-size of 12pt equals 24 half points pct: Half-points are used to specify font sizes. A font-size of 12pt equals 24 half points

EMU: English Metric Unit, EMUs are used for coordinates in vector-based drawings and embedded pictures. One inch equates to 914400 EMUs and a centimeter is 360000. For bitmaps the default resolution is 96 dpi (known as PixelsPerInch in Excel). Spec p. 1122

For radial geometry Excel uses integert units of 1/60000th of a degree.

`openpyxl.utils.units.`**`EMU_to_cm`**(*value*)

`openpyxl.utils.units.`**`EMU_to_inch`**(*value*)

`openpyxl.utils.units.`**`EMU_to_pixels`**(*value*)

`openpyxl.utils.units.`**`angle_to_degrees`**(*value*)

`openpyxl.utils.units.`**`cm_to_EMU`**(*value*)

1 cm = 360000 EMUs

`openpyxl.utils.units.`**`cm_to_dxa`**(*value*)

`openpyxl.utils.units.`**`degrees_to_angle`**(*value*)

1 degree = 60000 angles

`openpyxl.utils.units.`**`dxa_to_cm`**(*value*)

`openpyxl.utils.units.`**`dxa_to_inch`**(*value*)

`openpyxl.utils.units.`**`inch_to_EMU`**(*value*)

1 inch = 914400 EMUs

openpyxl.utils.units.**inch_to_dxa**(*value*)
: 1 inch = 72 * 20 dxa

openpyxl.utils.units.**pixels_to_EMU**(*value*)
: 1 pixel = 9525 EMUs

openpyxl.utils.units.**pixels_to_points**(*value*, *dpi=96*)
: 96 dpi, 72i

openpyxl.utils.units.**points_to_pixels**(*value*, *dpi=96*)

openpyxl.utils.units.**short_color**(*color*)
: format a color to its short size

Module contents

openpyxl.utils.**absolute_coordinate**(*coord_string*)
: Convert a coordinate to an absolute coordinate string (B12 -> $B$12)

openpyxl.utils.**cells_from_range**(*range_string*)
: Get individual addresses for every cell in a range. Yields one row at a time.

openpyxl.utils.**column_index_from_string**(*str_col*)
: Convert a column name into a numerical index (‘A’ -> 1)

openpyxl.utils.**coordinate_from_string**(*coord_string*)
: Convert a coordinate string like ‘B12’ to a tuple (‘B’, 12)

openpyxl.utils.**get_column_interval**(*start*, *end*)

openpyxl.utils.**get_column_letter**(*idx*)
: Convert a column index into a column letter (3 -> ‘C’)

openpyxl.utils.**range_boundaries**(*range_string*)
: Convert a range string into a tuple of boundaries: (min_col, min_row, max_col, max_row) Cell coordinates will be converted into a range with the cell at both end

openpyxl.workbook package

Subpackages

openpyxl.workbook.names package

Submodules

openpyxl.workbook.names.external module

class openpyxl.workbook.names.external.**ExternalBook**(*Id*, *Target*, *TargetMode=None*, *Type=None*)
: Bases: *openpyxl.descriptors.Strict*

  Map the relationship of one workbook to another

  Id

  Target

  TargetMode = ‘External’

Type = 'http://schemas.openxmlformats.org/officeDocument/2006/relationships/externalLinkPath'

class `openpyxl.workbook.names.external.`**`ExternalRange`**(*name*, *refersTo=None*, *sheetId=None*)

Bases: `openpyxl.descriptors.Strict`

Map external named ranges NB. the specification for these is different to named ranges within a workbook See 18.14.5

name

refersTo

sheetId

`openpyxl.workbook.names.external.`**`detect_external_links`**(*rels*, *archive*)

`openpyxl.workbook.names.external.`**`parse_books`**(*xml*)

`openpyxl.workbook.names.external.`**`parse_ranges`**(*xml*)

`openpyxl.workbook.names.external.`**`write_external_book_rel`**(*book*)

Serialise link to external file

`openpyxl.workbook.names.external.`**`write_external_link`**(*links*)

Serialise links to ranges in a single external worbook

openpyxl.workbook.names.named_range module

class `openpyxl.workbook.names.named_range.`**`NamedRange`**(*name*, *destinations*, *scope=None*)

Bases: `openpyxl.workbook.names.named_range.NamedValue`

A named group of cells

Scope is a worksheet object or None for workbook scope names (the default)

destinations

name

repr_format = u'<%s "%s">'

scope

str_format = u'%s!%s'

value

`openpyxl.workbook.names.named_range.`**`NamedRangeContainingValue`**

alias of `NamedValue`

class `openpyxl.workbook.names.named_range.`**`NamedValue`**(*name*, *value*)

Bases: `object`

A named value

localSheetId

name

scope

value

`openpyxl.workbook.names.named_range.`**`external_range`**(*range_string*)

`openpyxl.workbook.names.named_range.`**`read_named_ranges`**(*xml_source*, *workbook*)

Read named ranges, excluding poorly defined ranges.

`openpyxl.workbook.names.named_range.`**`refers_to_range`**(*range_string*) openpyxl.workbook.names.named_range.**split_named_range**(*range_string*)

Separate a named range into its component parts

Module contents

Submodules

openpyxl.workbook.properties module

class openpyxl.workbook.properties.**DocumentProperties**(*category=None, contentStatus=None, keywords=None, lastModifiedBy=None, lastPrinted=None, revision=None, version=None, created=datetime.datetime(2015, 6, 30, 2, 40, 51, 738612), creator='openpyxl', description=None, identifier=None, language=None, modified=datetime.datetime(2015, 6, 30, 2, 40, 51, 738622), subject=None, title=None*)

Bases: *openpyxl.descriptors.Strict*

High-level properties of the document. Defined in ECMA-376 Par2 Annex D

category

contentStatus

created

creator

description

identifier

keywords

language

lastModifiedBy

lastPrinted

modified

revision

subject

title

version

class openpyxl.workbook.properties.**DocumentSecurity**

Bases: object

Security information about the document.

class `openpyxl.workbook.properties.`**`W3CDateTime`**(*name=None*, ***kw*)

Bases: *`openpyxl.descriptors.base.Typed`*

`expected_type`

alias of `datetime`

`openpyxl.workbook.properties.`**`read_properties`**(*xml_source*)

`openpyxl.workbook.properties.`**`write_properties`**(*props*)

Write the core properties to xml.

openpyxl.workbook.workbook module

class `openpyxl.workbook.workbook.`**`Workbook`**(*optimized_write=False*, *encoding='utf-8'*, *worksheet_class=<class 'openpyxl.worksheet.worksheet.Worksheet'>*, *guess_types=False*, *data_only=False*, *read_only=False*, *write_only=False*)

Bases: `object`

Workbook is the container for all other parts of the document.

`active`

Get the currently active sheet

`add_named_range`(*named_range*)

Add an existing named_range to the list of named_ranges.

`add_sheet`(**args*, ***kwargs*)

`create_named_range`(*name*, *worksheet*, *range*, *scope=None*)

Create a new named_range on a worksheet

`create_sheet`(*index=None*, *title=None*)

Create a worksheet (at an optional index).

Parameters **`index`** (*int*) – optional position at which the sheet will be inserted

`get_active_sheet`()

Returns the current active sheet.

`get_index`(*worksheet*)

Return the index of the worksheet.

`get_named_range`(*name*)

Return the range specified by name.

`get_named_ranges`()

Return all named ranges

`get_sheet_by_name`(*name*)

Returns a worksheet by its name.

Parameters **`name`** (*string*) – the name of the worksheet to look for

`get_sheet_names`()

`read_only`

`read_workbook_settings`(**args*, ***kwargs*)

`remove_named_range`(*named_range*)

Remove a named_range from this workbook.

remove_sheet (*worksheet*)
Remove a worksheet from this workbook.

save (*filename*)
Save the current workbook under the given *filename*. Use this function instead of using an *ExcelWriter*.

> **Warning:** When creating your workbook using *write_only* set to True, you will only be able to call this function once. Subsequents attempts to modify or save the file will raise an `openpyxl.shared.exc.WorkbookAlreadySaved` exception.

shared_styles
Legacy On the fly conversion of style references to style objects

sheetnames
Returns the list of the names of worksheets in the workbook.

Names are returned in the worksheets order.

Return type list of strings

write_only

Module contents

openpyxl.worksheet package

Submodules

openpyxl.worksheet.datavalidation module

class `openpyxl.worksheet.datavalidation.`**DataValidation** (*type=None, formula1=None, formula2=None, allow_blank=False, showErrorMessage=True, showInputMessage=True, showDropDown=None, allowBlank=None, sqref=None, promptTitle=None, errorStyle=None, error=None, prompt=None, errorTitle=None, imeMode=None, operator=None, validation_type=None*)

Bases: *openpyxl.descriptors.Strict*

add (*cell*)
Adds a openpyxl.cell to this validator

add_cell (**args, **kwargs*)
Adds a openpyxl.cell to this validator

allowBlank

allow_blank

error

errorStyle
'none' will be treated as None

errorTitle

formula1

formula2

imeMode
'none' will be treated as None

operator
'none' will be treated as None

prompt

promptTitle

set_error_message(**args*, ***kwargs*)
Creates a custom error message, displayed when a user changes a cell to an invalid value

set_prompt_message(**args*, ***kwargs*)
Creates a custom prompt message

showDropDown

showErrorMessage

showInputMessage

sqref

type
'none' will be treated as None

openpyxl.worksheet.datavalidation.**ValidationErrorStyle**(**args*, ***kwargs*)

openpyxl.worksheet.datavalidation.**ValidationOperator**(**args*, ***kwargs*)

openpyxl.worksheet.datavalidation.**ValidationType**(**args*, ***kwargs*)

openpyxl.worksheet.datavalidation.**collapse_cell_addresses**(*cells*, *input_ranges=()*)
Collapse a collection of cell co-ordinates down into an optimal range or collection of ranges.

E.g. Cells A1, A2, A3, B1, B2 and B3 should have the data-validation object applied, attempt to collapse down to a single range, A1:B3.

Currently only collapsing contiguous vertical ranges (i.e. above example results in A1:A3 B1:B3). More work to come.

openpyxl.worksheet.datavalidation.**expand_cell_ranges**(*range_string*)
Expand cell ranges to a sequence of addresses. Reverse of collapse_cell_addresses Eg. converts "A1:A2 B1:B2" to (A1, A2, B1, B2)

openpyxl.worksheet.datavalidation.**parser**(*element*)
Parse dataValidation tag

openpyxl.worksheet.datavalidation.**writer**(*data_validation*)
Serialse a data validation

openpyxl.worksheet.dimensions module

class `openpyxl.worksheet.dimensions.`**`ColumnDimension`**(*worksheet*, *index='A'*, *width=None*, *bestFit=False*, *hidden=False*, *outlineLevel=0*, *outline_level=None*, *collapsed=False*, *style=None*, *min=None*, *max=None*, *customWidth=False*, *visible=None*, *auto_size=None*)

Bases: *`openpyxl.worksheet.dimensions.Dimension`*

Information about the display properties of a column.

`bestFit`

`collapsed`

`customWidth`

Always true if there is a width for the column

`index`

`max`

`min`

`width`

class `openpyxl.worksheet.dimensions.`**`Dimension`**(*index*, *hidden*, *outlineLevel*, *collapsed*, *worksheet*, *visible=True*, *style=None*)

Bases: *`openpyxl.descriptors.Strict`*, *`openpyxl.styles.styleable.StyleableObject`*

Information about the display properties of a row or column.

`collapsed`

`hidden`

`index`

`outlineLevel`

`visible`

class `openpyxl.worksheet.dimensions.`**`DimensionHolder`**(*worksheet*, *direction*, **args*, ***kwargs*)

Bases: `collections.OrderedDict`

hold (row|column)dimensions and allow operations over them

`group`(*start*, *end=None*, *outline_level=1*, *hidden=False*)

allow grouping a range of consecutive columns together

Parameters

- **`start`** – first column to be grouped (mandatory)
- **`end`** – last column to be grouped (optional, default to start)
- **`outline_level`** – outline level
- **`hidden`** – should the group be hidden on workbook open or not

class `openpyxl.worksheet.dimensions.`**`RowDimension`**(*worksheet*, *index=0*, *ht=None*, *customHeight=None*, *s=None*, *customFormat=None*, *hidden=False*, *outlineLevel=0*, *outline_level=None*, *collapsed=False*, *visible=None*, *height=None*, *r=None*, *spans=None*, *thickBot=None*, *thickTop=None*, ***kw*)

Bases: `openpyxl.worksheet.dimensions.Dimension`

Information about the display properties of a row.

`customFormat`
Always true if there is a style for the row

`customHeight`
Always true if there is a height for the row

`ht`

`thickBot`

`thickTop`

openpyxl.worksheet.filters module

class `openpyxl.worksheet.filters.`**`AutoFilter`**
Bases: `object`

Represents a auto filter.

Don't create auto filters by yourself. It is created by `Worksheet`. You can use via `auto_filter` attribute.

`add_filter_column`(*col_id*, *vals*, *blank=False*)
Add row filter for specified column.

Parameters

- **`col_id`** (*int*) – Zero-origin column id. 0 means first column.
- **`vals`** (*str[]*) – Value list to show.
- **`blank`** (*bool*) – Show rows that have blank cell if True (default=‘‘False‘‘)

`add_sort_condition`(*ref*, *descending=False*)
Add sort condition for cpecified range of cells.

Parameters

- **`ref`** (*string*) – range of the cells (e.g. ‘A2:A150’)
- **`descending`** (*bool*) – Descending sort order (default=‘‘False‘‘)

`filter_columns`
Return filters for columns.

`ref`
Return the reference of this auto filter.

`sort_conditions`
Return sort conditions

class `openpyxl.worksheet.filters.`**`FilterColumn`**(*col_id*, *vals*, *blank*)
Bases: `object`

blank

col_id

vals

class openpyxl.worksheet.filters.**SortCondition**(*ref*, *descending*)
Bases: object

descending

ref
Return the ref for this sheet.

openpyxl.worksheet.filters.**normalize_reference**(*cell_range*)

openpyxl.worksheet.header_footer module

class openpyxl.worksheet.header_footer.**HeaderFooter**
Bases: object

Information about the header/footer for this sheet.

center_footer

center_header

getFooter()

getHeader()

hasFooter()

hasHeader()

left_footer

left_header

right_footer

right_header

setFooter(*item*)

setHeader(*item*)

class openpyxl.worksheet.header_footer.**HeaderFooterItem**(*type*)
Bases: object

Individual left/center/right header/footer items

Header & Footer ampersand codes:

- &A Inserts the worksheet name
- &B Toggles bold
- &D or &[Date] Inserts the current date
- &E Toggles double-underline
- &F or &[File] Inserts the workbook name
- &I Toggles italic
- &N or &[Pages] Inserts the total page count

- &S Toggles strikethrough
- &T Inserts the current time
- &[Tab] Inserts the worksheet name
- &U Toggles underline
- &X Toggles superscript
- &Y Toggles subscript
- &P or &[Page] Inserts the current page number
- &P+n Inserts the page number incremented by n
- &P-n Inserts the page number decremented by n
- &[Path] Inserts the workbook path
- && Escapes the ampersand character
- &"fontname" Selects the named font
- &nn Selects the specified 2-digit font point size

`CENTER` = **'C'**

`LEFT` = **'L'**

`REPLACE_LIST` = **(('\n', '_x000D_'), ('&[Page]', '&P'), ('&[Pages]', '&N'), ('&[Date]', '&D'), ('&[Time]', '&T'), ('&[Pa**

`RIGHT` = **'R'**

`font_color`

`font_name`

`font_size`

`get`()

`has`()

`set`(*text*)

Convert a compound string into attributes # incomplete because formatting commands can be nested

`text`

`type`

openpyxl.worksheet.iter_worksheet module

class `openpyxl.worksheet.iter_worksheet.`**`IterableWorksheet`**(*parent_workbook, title, worksheet_path, xml_source, shared_strings, style_table*)

Bases: *`openpyxl.worksheet.worksheet.Worksheet`*

`calculate_dimension`(*force=False*)

`get_highest_column`()

`get_highest_row`()

get_squared_range (*min_col*, *min_row*, *max_col*, *max_row*)
The source worksheet file may have columns or rows missing. Missing cells will be created.

get_style (*coordinate*)

max_col = None

max_row = None

min_col = 'A'

min_row = 1

rows

xml_source
Parse xml source on demand, default to Excel archive

openpyxl.worksheet.iter_worksheet.**read_dimension** (*source*)

openpyxl.worksheet.page module

class openpyxl.worksheet.page.**PageMargins** (*left=0.75*, *right=0.75*, *top=1*, *bottom=1*, *header=0.5*, *footer=0.5*)
Bases: *openpyxl.descriptors.Strict*

Information about page margins for view/print layouts. Standard values (in inches) left, right = 0.75 top, bottom = 1 header, footer = 0.5

bottom

footer

header

left

right

top

class openpyxl.worksheet.page.**PageSetup** (*worksheet=None*, *orientation=None*, *paperSize=None*, *scale=None*, *fitToHeight=None*, *fitToWidth=None*, *firstPageNumber=None*, *useFirstPageNumber=None*, *paperHeight=None*, *paperWidth=None*, *pageOrder=None*, *usePrinterDefaults=None*, *blackAndWhite=None*, *draft=None*, *cellComments=None*, *errors=None*, *horizontalDpi=None*, *verticalDpi=None*, *copies=None*, *id=None*)
Bases: *openpyxl.descriptors.Strict*

Worksheet page setup

autoPageBreaks

blackAndWhite

cellComments
'none' will be treated as None

copies

draft

errors
  'none' will be treated as None

firstPageNumber

fitToHeight

fitToPage

fitToWidth

horizontalCentered(**args*, ***kwargs*)

horizontalDpi

id

options(**args*, ***kwargs*)

orientation
  'none' will be treated as None

pageOrder
  'none' will be treated as None

paperHeight

paperSize

paperWidth

scale

setup(**args*, ***kwargs*)

sheet_properties
  Proxy property

tag = 'pageSetup'

useFirstPageNumber

usePrinterDefaults

verticalCentered(**args*, ***kwargs*)

verticalDpi

write_xml_element()

class openpyxl.worksheet.page.**PrintOptions**(*horizontalCentered=None*, *verticalCentered=None*, *headings=None*, *gridLines=None*, *gridLinesSet=None*)

Bases: *openpyxl.descriptors.Strict*

Worksheet print options

gridLines

gridLinesSet

headings

horizontalCentered

tag = '{http://schemas.openxmlformats.org/spreadsheetml/2006/main}printOptions'

verticalCentered

write_xml_element()

openpyxl.worksheet.properties module

class `openpyxl.worksheet.properties.`**`Outline`**(*applyStyles=None*, *summaryBelow=None*, *summaryRight=None*, *showOutlineSymbols=None*)

Bases: `openpyxl.descriptors.Strict`

`applyStyles`

`showOutlineSymbols`

`summaryBelow`

`summaryRight`

`tag` **= '{http://schemas.openxmlformats.org/spreadsheetml/2006/main}outlinePr'**

class `openpyxl.worksheet.properties.`**`PageSetupProperties`**(*autoPageBreaks=None*, *fitToPage=None*)

Bases: `openpyxl.descriptors.Strict`

`autoPageBreaks`

`fitToPage`

`tag` **= '{http://schemas.openxmlformats.org/spreadsheetml/2006/main}pageSetUpPr'**

class `openpyxl.worksheet.properties.`**`WorksheetProperties`**(*codeName=None*, *enableFormatConditionsCalculation=None*, *filterMode=None*, *published=None*, *syncHorizontal=None*, *syncRef=None*, *syncVertical=None*, *transitionEvaluation=None*, *transitionEntry=None*, *tabColor=None*, *outlinePr=None*, *pageSetUpPr=None*)

Bases: `openpyxl.descriptors.Strict`

`codeName`

`enableFormatConditionsCalculation`

`filterMode`

`outlinePr`
Values must of a particular type

`pageSetUpPr`
Values must of a particular type

`published`

`syncHorizontal`

`syncRef`

`syncVertical`

`tabColor`

`tag` **= '{http://schemas.openxmlformats.org/spreadsheetml/2006/main}sheetPr'**

`transitionEntry`
Elements

transitionEvaluation

openpyxl.worksheet.properties.**parse_sheetPr**(*node*)

openpyxl.worksheet.properties.**write_sheetPr**(*props*)

openpyxl.worksheet.protection module

class openpyxl.worksheet.protection.**SheetProtection**(*sheet=False*, *objects=False*, *scenarios=False*, *formatCells=True*, *formatRows=True*, *formatColumns=True*, *insertColumns=True*, *insertRows=True*, *insertHyperlinks=True*, *deleteColumns=True*, *deleteRows=True*, *selectLockedCells=False*, *selectUnlockedCells=False*, *sort=True*, *autoFilter=True*, *pivotTables=True*, *password=None*, *algorithmName=None*, *saltValue=None*, *spinCount=None*)

Bases: *openpyxl.descriptors.Strict*

Information about protection of various aspects of a sheet. True values mean that protection for the object or action is active This is the **default** when protection is active, ie. users cannot do something

algorithmName

autoFilter

deleteColumns

deleteRows

disable()

enable()

formatCells

formatColumns

formatRows

insertColumns

insertHyperlinks

insertRows

objects

password

Return the password value, regardless of hash.

pivotTables

saltValue

scenarios

selectLockedCells

selectUnlockedCells

set_password(*value=''*, *already_hashed=False*)
Set a password on this sheet.

sheet

sort

spinCount

openpyxl.worksheet.protection.**hash_password**(*plaintext_password=''*)
Create a password hash from a given string for protecting a worksheet only. This will not work for encrypting a workbook.

This method is based on the algorithm provided by Daniel Rentz of OpenOffice and the PEAR package Spreadsheet_Excel_Writer by Xavier Noguer <xnoguer@rezebra.com>. See also http://blogs.msdn.com/b/ericwhite/archive/2008/02/23/the-legacy-hashing-algorithm-in-open-xml.aspx

openpyxl.worksheet.relationship module

class openpyxl.worksheet.relationship.**Relationship**(*rel_type*, *target=None*, *target_mode=None*, *id=None*)
Bases: object

Represents many kinds of relationships.

TYPES = ('hyperlink', 'drawing', 'image')

openpyxl.worksheet.views module

class openpyxl.worksheet.views.**Pane**(*xSplit=None*, *ySplit=None*, *topLeftCell=None*, *activePane='topLeft'*, *state='split'*)
Bases: *openpyxl.descriptors.serialisable.Serialisable*

activePane
Value can only be from a set of know values

state
Value can only be from a set of know values

topLeftCell

xSplit

ySplit

class openpyxl.worksheet.views.**Selection**(*pane=None*, *activeCell='A1'*, *activeCellId=None*, *sqref='A1'*)
Bases: *openpyxl.descriptors.serialisable.Serialisable*

activeCell

activeCellId

pane
'none' will be treated as None

sqref

class `openpyxl.worksheet.views.`**`SheetView`**(*windowProtection=None, showFormulas=None, showGridLines=True, showRowColHeaders=None, showZeros=None, rightToLeft=None, tabSelected=None, showRuler=None, showOutlineSymbols=None, defaultGridColor=None, showWhiteSpace=None, view=None, topLeftCell=None, colorId=None, zoomScale=None, zoomScaleNormal=None, zoomScaleSheetLayoutView=None, zoomScalePageLayoutView=None, workbookViewId=0, selection=None, pane=None*)

Bases: `openpyxl.descriptors.serialisable.Serialisable`

Information about the visible portions of this sheet.

`colorId`

`defaultGridColor`

`pane`

Values must of a particular type

`rightToLeft`

`selection`

A sequence (list or tuple) that may only contain objects of the declared type

`showFormulas`

`showGridLines`

`showOutlineSymbols`

`showRowColHeaders`

`showRuler`

`showWhiteSpace`

`showZeros`

`tabSelected`

`tagname` = **'sheetView'**

`topLeftCell`

`view`

'none' will be treated as None

`windowProtection`

`workbookViewId`

`zoomScale`

`zoomScaleNormal`

`zoomScalePageLayoutView`

`zoomScaleSheetLayoutView`

openpyxl.worksheet.worksheet module

class `openpyxl.worksheet.worksheet.`**`Worksheet`**(*parent_workbook, title='Sheet'*)

Bases: `object`

Represents a worksheet.

Do not create worksheets yourself, use `openpyxl.workbook.Workbook.create_sheet()` instead

`BREAK_COLUMN` = **2**

`BREAK_NONE` = **0**

`BREAK_ROW` = **1**

`ORIENTATION_LANDSCAPE` = **'landscape'**

`ORIENTATION_PORTRAIT` = **'portrait'**

`PAPERSIZE_A3` = **'8'**

`PAPERSIZE_A4` = **'9'**

`PAPERSIZE_A4_SMALL` = **'10'**

`PAPERSIZE_A5` = **'11'**

`PAPERSIZE_EXECUTIVE` = **'7'**

`PAPERSIZE_LEDGER` = **'4'**

`PAPERSIZE_LEGAL` = **'5'**

`PAPERSIZE_LETTER` = **'1'**

`PAPERSIZE_LETTER_SMALL` = **'2'**

`PAPERSIZE_STATEMENT` = **'6'**

`PAPERSIZE_TABLOID` = **'3'**

`SHEETSTATE_HIDDEN` = **'hidden'**

`SHEETSTATE_VERYHIDDEN` = **'veryHidden'**

`SHEETSTATE_VISIBLE` = **'visible'**

`active_cell`

`add_chart`(*chart*)
Add a chart to the sheet

`add_data_validation`(*data_validation*)
Add a data-validation object to the sheet. The data-validation object defines the type of data-validation to be applied and the cell or range of cells it should apply to.

`add_drawing`(*obj*)
Images and charts both create drawings

`add_image`(*img*)
Add an image to the sheet

`add_print_title`(*n*, *rows_or_cols='rows'*)
Print Titles are rows or columns that are repeated on each printed sheet. This adds n rows or columns at the top or left of the sheet

`add_rel`(*obj*)
Drawings and hyperlinks create relationships

`append`(*iterable*)
Appends a group of values at the bottom of the current sheet.

- If it's a list: all values are added in order, starting from the first column

- If it's a dict: values are assigned to the columns indicated by the keys (numbers or letters)

  Parameters **`iterable`** (*list/tuple/range/generator or dict*) – list, range or generator, or dict containing values to append

Usage:

- append(['This is A1', 'This is B1', 'This is C1'])
- **or** append({'A' : 'This is A1', 'C' : 'This is C1'})
- **or** append({1 : 'This is A1', 3 : 'This is C1'})

  Raise TypeError when iterable is neither a list/tuple nor a dict

`auto_filter`
Return `AutoFilter` object.

auto_filter attribute stores/returns string until 1.8. You should change your code like `ws.auto_filter.ref = "A1:A3"`.

Changed in version 1.9.

`bad_title_char_re` **= <_sre.SRE_Pattern object>**

`calculate_dimension`()
Return the minimum bounding range for all cells containing data.

`cell`(*coordinate=None*, *row=None*, *column=None*, *value=None*)
Returns a cell object based on the given coordinates.

Usage: cell(coodinate='A15') **or** cell(row=15, column=1)

If *coordinates* are not given, then row *and* column must be given.

Cells are kept in a dictionary which is empty at the worksheet creation. Calling *cell* creates the cell in memory when they are first accessed, to reduce memory usage.

Parameters

- **`coordinate`** (*string*) – coordinates of the cell (e.g. 'B12')
- **`row`** (*int*) – row index of the cell (e.g. 4)
- **`column`** (*int*) – column index of the cell (e.g. 3)

Raise InsufficientCoordinatesException when coordinate or (row and column) are not given

Return type :class:openpyxl.cell.Cell

`columns`
Iterate over all columns in the worksheet

`create_relationship`(**args*, ***kwargs*)

`dimensions`

`encoding`

`freeze_panes`

`garbage_collect`(**args*, ***kwargs*)

`get_cell_collection`()
Return an unordered list of the cells in this worksheet.

get_highest_column()
Get the largest value for column currently stored.

Return type int

get_highest_row()
Returns the maximum row index containing data

Return type int

get_named_range(*range_string*)
Returns a 2D array of cells, with optional row and column offsets.

Parameters **range_string** (*string*) – *named range* name

Return type tuples of tuples of `openpyxl.cell.Cell`

get_squared_range(*min_col*, *min_row*, *max_col*, *max_row*)
Returns a 2D array of cells

Parameters

- **min_col** (*int*) – smallest column index (1-based index)
- **min_row** (*int*) – smallest row index (1-based index)
- **max_col** (*int*) – largest column index (1-based index)
- **max_row** (*int*) – smallest row index (1-based index)

Return type generator

get_style(**args*, ***kwargs*)
Return a copy of the style object for the specified cell.

iter_rows(*range_string=None*, *row_offset=0*, *column_offset=0*)
Returns a squared range based on the *range_string* parameter, using generators. If no range is passed, will iterate over all cells in the worksheet

Parameters

- **range_string** (*string*) – range of cells (e.g. 'A1:C4')
- **row_offset** – additional rows (e.g. 4)
- **column_offset** – additonal columns (e.g. 3)

Return type generator

max_column

max_row

merge_cells(*range_string=None*, *start_row=None*, *start_column=None*, *end_row=None*, *end_column=None*)
Set merge on a cell range. Range is a cell range (e.g. A1:E1)

merged_cell_ranges
Public attribute for which cells have been merged

merged_cells
Utility for checking whether a cell has been merged or not

min_col

min_row

parent

point_pos (*left=0*, *top=0*)
tells which cell is under the given coordinates (in pixels) counting from the top-left corner of the sheet. Can be used to locate images and charts on the worksheet

range (**args*, ***kwargs*)
Returns a 2D array of cells, with optional row and column offsets.

Parameters

- **range_string** (*string*) – cell range string or *named range* name
- **row** (*int*) – number of rows to offset
- **column** (*int*) – number of columns to offset

Return type tuples of tuples of `openpyxl.cell.Cell`

repr_format = u'<Worksheet "%s">'

rows
Iterate over all rows in the worksheet

selected_cell

set_printer_settings (*paper_size*, *orientation*)
Set printer settings

set_style (**args*, ***kwargs*)

show_gridlines

show_summary_below

show_summary_right

title
Return the title for this sheet.

unique_sheet_name (**args*, ***kwargs*)

unmerge_cells (*range_string=None*, *start_row=None*, *start_column=None*, *end_row=None*, *end_column=None*)
Remove merge on a cell range. Range is a cell range (e.g. A1:E1)

vba_code

`openpyxl.worksheet.worksheet.`**flatten** (*results*)
Return cell values row-by-row

Module contents

openpyxl.writer package

Submodules

openpyxl.writer.comments module

class `openpyxl.writer.comments.`**CommentWriter** (*sheet*)
Bases: `object`

extract_comments ()
extract list of comments and authors

write_comments()

write_comments_vml()

openpyxl.writer.drawings module

class openpyxl.writer.drawings.**DrawingWriter**(*sheet*)

Bases: object

one main drawing file per sheet

write()

write drawings for one sheet in one file

write_rels(*chart_id*, *image_id*)

class openpyxl.writer.drawings.**ShapeWriter**(*shapes*)

Bases: object

one file per shape

write(*shape_id*)

openpyxl.writer.dump_worksheet module

class openpyxl.writer.dump_worksheet.**CommentParentCell**(*cell*)

Bases: object

column

coordinate

row

class openpyxl.writer.dump_worksheet.**DumpCommentWriter**(*sheet*)

Bases: *openpyxl.writer.comments.CommentWriter*

extract_comments()

class openpyxl.writer.dump_worksheet.**DumpWorksheet**(*parent_workbook*, *title*)

Bases: *openpyxl.worksheet.worksheet.Worksheet*

Streaming worksheet using lxml Optimised to reduce memory by writing rows just in time Cells can be styled and have comments Styles for rows and columns must be applied before writing cells

append(*row*)

Parameters **row** (*iterable*) – iterable containing values to append

cell(**args*, ***kw*)

close()

filename

merge_cells(**args*, ***kw*)

range(**args*, ***kw*)

writer = None

class openpyxl.writer.dump_worksheet.**ExcelDumpWriter**(*workbook*)

Bases: *openpyxl.writer.excel.ExcelWriter* `openpyxl.writer.dump_worksheet.`**`WriteOnlyCell`**(*ws=None*, *value=None*)

`openpyxl.writer.dump_worksheet.`**`create_temporary_file`**(*suffix=''*)

`openpyxl.writer.dump_worksheet.`**`removed_method`**(**args*, ***kw*)

`openpyxl.writer.dump_worksheet.`**`save_dump`**(*workbook*, *filename*)

openpyxl.writer.etree_worksheet module

`openpyxl.writer.etree_worksheet.`**`get_rows_to_write`**(*worksheet*)
Return all rows, and any cells that they contain

`openpyxl.writer.etree_worksheet.`**`row_sort`**(*cell*)
Translate column names for sorting.

`openpyxl.writer.etree_worksheet.`**`write_cell`**(*worksheet*, *cell*)

`openpyxl.writer.etree_worksheet.`**`write_rows`**(*xf*, *worksheet*)
Write worksheet data to xml.

openpyxl.writer.excel module

class `openpyxl.writer.excel.`**`ExcelWriter`**(*workbook*)
Bases: `object`

Write a workbook object to an Excel file.

`save`(*filename*, *as_template=False*)
Write data into the archive.

`write_data`(*archive*, *as_template=False*)
Write the various xml files into the zip archive.

`openpyxl.writer.excel.`**`save_virtual_workbook`**(*workbook*, *as_template=False*)
Return an in-memory workbook, suitable for a Django response.

`openpyxl.writer.excel.`**`save_workbook`**(*workbook*, *filename*, *as_template=False*)
Save the given workbook on the filesystem under the name filename.

Parameters

- **`workbook`** (`openpyxl.workbook.Workbook`) – the workbook to save
- **`filename`** (*string*) – the path to which save the workbook

Return type bool

openpyxl.writer.lxml_worksheet module

`openpyxl.writer.lxml_worksheet.`**`write_cell`**(*xf*, *worksheet*, *cell*)

`openpyxl.writer.lxml_worksheet.`**`write_rows`**(*xf*, *worksheet*)
Write worksheet data to xml.

openpyxl.writer.relations module

`openpyxl.writer.relations.`**`write_rels`**`(`*worksheet*, *drawing_id*, *comments_id*, *vba_controls_id*`)`
Write relationships for the worksheet to xml.

openpyxl.writer.strings module

`openpyxl.writer.strings.`**`write_string_table`**`(`*string_table*`)`
Write the string table xml.

openpyxl.writer.styles module

class `openpyxl.writer.styles.`**`StyleWriter`**`(`*workbook*`)`
Bases: `object`

`alignments`

`borders`

`fills`

`fonts`

`number_formats`

`protections`

`styles`

`write_table``()`

openpyxl.writer.theme module

`openpyxl.writer.theme.`**`write_theme`**`()`
Write the theme xml.

openpyxl.writer.workbook module

`openpyxl.writer.workbook.`**`write_content_types`**`(`*workbook*, *as_template=False*`)`
Write the content-types xml.

`openpyxl.writer.workbook.`**`write_properties_app`**`(`*workbook*`)`
Write the properties xml.

`openpyxl.writer.workbook.`**`write_root_rels`**`(`*workbook*`)`
Write the relationships xml.

`openpyxl.writer.workbook.`**`write_workbook`**`(`*workbook*`)`
Write the core workbook xml.

`openpyxl.writer.workbook.`**`write_workbook_rels`**`(`*workbook*`)`
Write the workbook relationships xml.

openpyxl.writer.worksheet module

openpyxl.writer.worksheet.**write_autofilter**(*worksheet*)

openpyxl.writer.worksheet.**write_cols**(*worksheet*)
Write worksheet columns to xml.

<cols> may never be empty - spec says must contain at least one child

openpyxl.writer.worksheet.**write_conditional_formatting**(*worksheet*)
Write conditional formatting to xml.

openpyxl.writer.worksheet.**write_datavalidation**(*worksheet*)
Write data validation(s) to xml.

openpyxl.writer.worksheet.**write_format**(*worksheet*)

openpyxl.writer.worksheet.**write_header_footer**(*worksheet*)

openpyxl.writer.worksheet.**write_hyperlinks**(*worksheet*)
Write worksheet hyperlinks to xml.

openpyxl.writer.worksheet.**write_mergecells**(*worksheet*)
Write merged cells to xml.

openpyxl.writer.worksheet.**write_pagebreaks**(*worksheet*)

openpyxl.writer.worksheet.**write_properties**(*worksheet*)

openpyxl.writer.worksheet.**write_worksheet**(*worksheet*, *shared_strings*)
Write a worksheet to an xml file.

Module contents

openpyxl.xml package

Submodules

openpyxl.xml.constants module

openpyxl.xml.functions module

openpyxl.xml.functions.**ConditionalElement**(*node*, *tag*, *condition*, *attr=None*)
Utility function for adding nodes if certain criteria are fulfilled An optional attribute can be passed in which will always be serialised as '1'

openpyxl.xml.functions.**get_document_content**(*xml_node*)
Print nicely formatted xml to a string.

openpyxl.xml.functions.**iterparse**(*source*, **args*, ***kw*)

openpyxl.xml.functions.**localname**(*node*)

openpyxl.xml.functions.**pretty_indent**(*elem*, *level=0*)
Format xml with nice indents and line breaks.

openpyxl.xml.functions.**safe_iterator**(*node*, *tag=None*)
Return an iterator that is compatible with Python 2.6

openpyxl.xml.functions.**safe_iterparse**(*source*, **args*, ***kw*)

openpyxl.xml.namespace module

openpyxl.xml.xmlfile module

exception `openpyxl.xml.xmlfile.`**`LxmlSyntaxError`**
    Bases: `exceptions.Exception`

class `openpyxl.xml.xmlfile.`**`xmlfile`**(*output_file*, *buffered=False*, *encoding=None*, *close=False*)
    Bases: `object`

    Context manager that can replace lxml.etree.xmlfile.

Module contents

`openpyxl.xml.`**`lxml_available`**()

`openpyxl.xml.`**`lxml_env_set`**()

9.1.2 Module contents

Imports for the openpyxl package.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

CHAPTER 11

Release Notes

11.1 2.2.4 (2015-06-17)

11.1.1 Bug fixes

- #464 Cannot use images when preserving macros
- #465 ws.cell() returns an empty cell on read-only workbooks
- #467 Cannot edit a file with ActiveX components
- #471 Sheet properties elements must be in order
- #475 Do not redefine class __slots__ in subclasses
- #477 Write-only support for SheetProtection
- #478 Write-only support for DataValidation
- Improved regex when checking for datetime formats

11.2 2.2.3 (2015-05-26)

11.2.1 Bug fixes

- #451 fitToPage setting ignored
- #458 Trailing spaces lost when saving files.
- #459 setup.py install fails with Python 3
- #462 Vestigial rId conflicts when adding charts, images or comments
- #455 Enable Zip64 extensions for all versions of Python

11.3 2.2.2 (2015-04-28)

11.3.1 Bug fixes

- #447 Uppercase datetime number formats not recognised.

- #453 Borders broken in shared_styles.

11.4 2.2.1 (2015-03-31)

11.4.1 Minor changes

- PR54 Improved precision on times near midnight.
- PR55 Preserve macro buttons

11.4.2 Bug fixes

- #429 Workbook fails to load because header and footers cannot be parsed.
- #433 File-like object with encoding=None
- #434 SyntaxError when writing page breaks.
- #436 Read-only mode duplicates empty rows.
- #437 Cell.offset raises an exception
- #438 Cells with pivotButton and quotePrefix styles cannot be read
- #440 Error when customised versions of builtin formats
- #442 Exception raised when a fill element contains no children
- #444 Styles cannot be copied

11.5 2.2.0 (2015-03-11)

11.5.1 Bug fixes

- #415 Improved exception when passing in invalid in memory files.

11.6 2.2.0-b1 (2015-02-18)

11.6.1 Major changes

- Cell styles deprecated, use formatting objects (fonts, fills, borders, etc.) directly instead
- Charts will no longer try and calculate axes by default
- Support for template file types - PR21
- Moved ancillary functions and classes into utils package - single place of reference
- PR 34 Fully support page setup
- Removed SAX-based XML Generator. Special thanks to Elias Rabel for implementing xmlfile for xml.etree
- Preserve sheet view definitions in existing files (frozen panes, zoom, etc.)

11.6.2 Bug fixes

- #103 Set the zoom of a sheet
- #199 Hide gridlines
- #215 Preserve sheet view setings
- #262 Set the zoom of a sheet
- #392 Worksheet header not read
- #387 Cannot read files without styles.xml
- #410 Exception when preserving whitespace in strings
- #417 Cannot create print titles
- #420 Rename confusing constants
- #422 Preserve color index in a workbook if it differs from the standard

11.6.3 Minor changes

- Use a 2-way cache for column index lookups
- Clean up tests in cells
- PR 40 Support frozen panes and autofilter in write-only mode
- Use ws.calculate_dimension(force=True) in read-only mode for unsized worksheets

11.7 2.1.5 (2015-02-18)

11.7.1 Bug fixes

- #403 Cannot add comments in write-only mode
- #401 Creating cells in an empty row raises an exception
- #408 from_excel adjustment for Julian dates 1 < x < 60
- #409 refersTo is an optional attribute

11.7.2 Minor changes

- Allow cells to be appended to standard worksheets for code compatibility with write-only mode.

11.8 2.1.4 (2014-12-16)

11.8.1 Bug fixes

- #393 IterableWorksheet skips empty cells in rows
- #394 Date format is applied to all columns (while only first column contains dates)
- #395 temporary files not cleaned properly

- #396 Cannot write "=" in Excel file
- #398 Cannot write empty rows in write-only mode with LXML installed

11.8.2 Minor changes

- Add relation namespace to root element for compatibility with iWork
- Serialize comments relation in LXML-backend

11.9 2.1.3 (2014-12-09)

11.9.1 Minor changes

- PR 31 Correct tutorial
- PR 32 See #380
- PR 37 Bind worksheet to ColumnDimension objects

11.9.2 Bug fixes

- #379 ws.append() doesn't set RowDimension Correctly
- #380 empty cells formatted as datetimes raise exceptions

11.10 2.1.2 (2014-10-23)

11.10.1 Minor changes

- PR 30 Fix regex for positive exponentials
- PR 28 Fix for #328

11.10.2 Bug fixes

- #120, #168 defined names with formulae raise exceptions, #292
- #328 ValueError when reading cells with hyperlinks
- #369 IndexError when reading definedNames
- #372 number_format not consistently applied from styles

11.11 2.1.1 (2014-10-08)

11.11.1 Minor changes

- PR 20 Support different workbook code names
- Allow auto_axis keyword for ScatterCharts

11.11.2 Bug fixes

- #332 Fills lost in ConditionalFormatting
- #360 Support value="none" in attributes
- #363 Support undocumented value for textRotation
- #364 Preserve integers in read-only mode
- #366 Complete read support for DataValidation
- #367 Iterate over unsized worksheets

11.12 2.1.0 (2014-09-21)

11.12.1 Major changes

- "read_only" and "write_only" new flags for workbooks
- Support for reading and writing worksheet protection
- Support for reading hidden rows
- Cells now manage their styles directly
- ColumnDimension and RowDimension object manage their styles directly
- Use xmlfile for writing worksheets if available - around 3 times faster
- Datavalidation now part of the worksheet package

11.12.2 Minor changes

- Number formats are now just strings
- Strings can be used for RGB and aRGB colours for Fonts, Fills and Borders
- Create all style tags in a single pass
- Performance improvement when appending rows
- Cleaner conversion of Python to Excel values
- PR6 reserve formatting for empty rows
- standard worksheets can append from ranges and generators

11.12.3 Bug fixes

- #153 Cannot read visibility of sheets and rows
- #181 No content type for worksheets
- 241 Cannot read sheets with inline strings
- 322 1-indexing for merged cells
- 339 Correctly handle removal of cell protection
- 341 Cells with formulae do not round-trip

- 347 Read DataValidations
- 353 Support Defined Named Ranges to external workbooks

11.13 2.0.5 (2014-08-08)

11.13.1 Bug fixes

- #348 incorrect casting of boolean strings
- #349 roundtripping cells with formulae

11.14 2.0.4 (2014-06-25)

11.14.1 Minor changes

- Add a sample file illustrating colours

11.14.2 Bug fixes

- #331 DARKYELLOW was incorrect
- Correctly handle extend attribute for fonts

11.15 2.0.3 (2014-05-22)

11.15.1 Minor changes

- Updated docs

11.15.2 Bug fixes

- #319 Cannot load Workbooks with vertAlign styling for fonts

11.16 2.0.2 (2014-05-13)

11.17 2.0.1 (2014-05-13) brown bag

11.18 2.0.0 (2014-05-13) brown bag

11.18.1 Major changes

- This is last release that will support Python 3.2
- Cells are referenced with 1-indexing: A1 == cell(row=1, column=1)

- Use jdcal for more efficient and reliable conversion of datetimes
- Significant speed up when reading files
- Merged immutable styles
- Type inference is disabled by default
- RawCell renamed ReadOnlyCell
- ReadOnlyCell.internal_value and ReadOnlyCell.value now behave the same as Cell
- Provide no size information on unsized worksheets
- Lower memory footprint when reading files

11.18.2 Minor changes

- All tests converted to pytest
- Pyflakes used for static code analysis
- Sample code in the documentation is automatically run
- Support GradientFills
- BaseColWidth set

11.18.3 Pull requests

- #70 Add filterColumn, sortCondition support to AutoFilter
- #80 Reorder worksheets parts
- #82 Update API for conditional formatting
- #87 Add support for writing Protection styles, others
- #89 Better handling of content types when preserving macros

11.18.4 Bug fixes

- #46 ColumnDimension style error
- #86 reader.worksheet.fast_parse sets booleans to integers
- #98 Auto sizing column widths does not work
- #137 Workbooks with chartsheets
- #185 Invalid PageMargins
- #230 Using v in cells creates invalid files
- #243 - IndexError when loading workbook
- #263 - Forded conversion of line breaks
- #267 - Raise exceptions when passed invalid types
- #270 - Cannot open files which use non-standard sheet names or reference Ids
- #269 - Handling unsized worksheets in IterableWorksheet

- #270 - Handling Workbooks with non-standard references
- #275 - Handling auto filters where there are only custom filters
- #277 - Harmonise chart and cell coordinates
- #280- Explicit exception raising for invalid characters
- #286 - Optimized writer can not handle a datetime.time value
- #296 - Cell coordinates not consistent with documentation
- #300 - Missing column width causes load_workbook() exception
- #304 - Handling Workbooks with absolute paths for worksheets (from Sharepoint)

11.19 1.8.6 (2014-05-05)

11.19.1 Minor changes

Fixed typo for import Elementtree

11.19.2 Bugfixes

- #279 Incorrect path for comments files on Windows

11.20 1.8.5 (2014-03-25)

11.20.1 Minor changes

- The '=' string is no longer interpreted as a formula
- When a client writes empty xml tags for cells (e.g. <c r='A1'></c>), reader will not crash

11.21 1.8.4 (2014-02-25)

11.21.1 Bugfixes

- #260 better handling of undimensioned worksheets
- #268 non-ascii in formualae
- #282 correct implementation of register_namepsace for Python 2.6

11.22 1.8.3 (2014-02-09)

11.22.1 Major changes

Always parse using cElementTree

11.22.2 Minor changes

Slight improvements in memory use when parsing

- #256 - error when trying to read comments with optimised reader
- #260 - unsized worksheets
- #264 - only numeric cells can be dates

11.23 1.8.2 (2014-01-17)

- #247 - iterable worksheets open too many files
- #252 - improved handling of lxml
- #253 - better handling of unique sheetnames

11.24 1.8.1 (2014-01-14)

- #246

11.25 1.8.0 (2014-01-08)

11.25.1 Compatibility

Support for Python 2.5 dropped.

11.25.2 Major changes

- Support conditional formatting
- Support lxml as backend
- Support reading and writing comments
- pytest as testrunner now required
- Improvements in charts: new types, more reliable

11.25.3 Minor changes

- load_workbook now accepts data_only to allow extracting values only from formulae. Default is false.
- Images can now be anchored to cells
- Docs updated
- Provisional benchmarking
- Added convenience methods for accessing worksheets and cells by key

11.26 1.7.0 (2013-10-31)

11.26.1 Major changes

Drops support for Python < 2.5 and last version to support Python 2.5

11.26.2 Compatibility

Tests run on Python 2.5, 2.6, 2.7, 3.2, 3.3

11.26.3 Merged pull requests

- 27 Include more metadata
- 41 Able to read files with chart sheets
- 45 Configurable Worksheet classes
- 3 Correct serialisation of Decimal
- 36 Preserve VBA macros when reading files
- 44 Handle empty oddheader and oddFooter tags
- 43 Fixed issue that the reader never set the active sheet
- 33 Reader set value and type explicitly and TYPE_ERROR checking
- 22 added page breaks, fixed formula serialization
- 39 Fix Python 2.6 compatibility
- 47 Improvements in styling

11.26.4 Known bugfixes

- #109
- #165
- #179
- #209
- #112
- #166
- #109
- #223
- #124
- #157

11.26.5 Miscellaneous

Performance improvements in optimised writer

Docs updated