xlutils Documentation

Release 2.0.0

Simplistix Ltd

May 24, 2017

Contents

This package provides a collection of utilities for working with Excel files. Since these utilities may require either or both of the `xlrd` and `xlwt` packages, they are collected together here, separate from either package. The utilities are grouped into several modules within the package, each of them is documented below:

xlutils.copy Tools for copying `xlrd.Book` objects to `xlwt.Workbook` objects.

xlutils.display Utility functions for displaying information about `xlrd`-related objects in a user-friendly and safe fashion.

xlutils.filter A mini framework for splitting and filtering existing Excel files into new Excel files.

xlutils.margins Tools for finding how much of an Excel file contains useful data.

xlutils.save Tools for serializing `xlrd.Book` objects back to Excel files.

xlutils.styles Tools for working with formatting information expressed the styles found in Excel files.

xlutils.view Easy to use views of the data contained in a workbook's sheets.

CHAPTER 1

xlutils.copy

The function in this module copies `xlrd.Book` objects into `xlwt.Workbook` objects so they can be manipulated. You may wish to do this, for example, if you have an existing excel file where you want to change come cells.

You would start by opening the file with `xlrd`:

```
>>> from os.path import join
>>> from xlrd import open_workbook
>>> rb = open_workbook(join(test_files,'testall.xls'), formatting_info=True, on_
↪demand=True)
>>> rb.sheet_by_index(0).cell(0,0).value
u'R0C0'
>>> rb.sheet_by_index(0).cell(0,1).value
u'R0C1'
```

You would then use *`xlutils.copy`* to copy the `xlrd.Book` object into an `xlwt.Workbook` object:

```
>>> from xlutils.copy import copy
>>> wb = copy(rb)
```

Now that you have an `xlwt.Workbook`, you can modify cells and then save the changed workbook back to a file:

```
>>> wb.get_sheet(0).write(0,0,'changed!')
>>> wb.save(join(temp_dir.path,'output.xls'))
>>> temp_dir.listdir()
output.xls
```

This file can now be loaded using `xlrd` to see the changes:

```
>>> rb = open_workbook(join(temp_dir.path,'output.xls'))
>>> rb.sheet_by_index(0).cell(0,0).value
u'changed!'
>>> rb.sheet_by_index(0).cell(0,1).value
u'R0C1'
```

Note: You should always pass *on_demand=True* to `open_workbook()` as this uses much less memory!

CHAPTER 2

xlutils.display

This module contains the `quoted_sheet_name()` and `cell_display()` functions that allow easy and safe display of information returned by `xlrd`.

The *API Reference* contains examples of their use.

CHAPTER 3

xlutils.filter

This framework is designed to filter and split Excel files using a series of modular readers, filters, and writers all tied together by the *process()* function. These components are described below, followed by documentation for the currently available readers, filters and writers. Right at the bottom is an example that puts all the various pieces together.

CHAPTER 4

Readers

Readers are components that get data from a source and parse it into a series of `xlrd.Book` objects before making calls to the appropriate methods on the first filter in the chain.

Because it is usually only the source of the data to be processed that changes, a handy base class is provided for readers.

Here's an example reader that uses this base class to process the file it is initialised with:

```
>>> import os
>>> from xlrd import open_workbook
>>> from xlutils.filter import BaseReader
>>> class MyReader(BaseReader):
...
...     def __init__(self, filename):
...         self.filename = filename
...
...     def get_filepaths(self):
...         return (os.path.join(test_files, self.filename),)
```

If you need to create a more unorthodox reader, such as one that reads its data from a network socket or that needs to pass special parameters to `xlrd.open_workbook()`, then read the implementation of *`BaseReader`*.

CHAPTER 5

Filters

Filters are the important bits of this framework. They are built up in chains to achieve the results required for a particular task. A filter must define certain methods, the full set of these is shown in the example below. The implementation of these methods can do whatever the filter requires, but generally they end up calling the appropriate methods on the next filter.

Here's an example filter that does nothing but print messages when its methods are called and then call the next filter in the chain:

```
>>> from __future__ import print_function
>>> class MyFilter:
...
...     def __init__(self,name):
...         self.name = name
...
...     def start(self):
...         print(self.name, 'start')
...         self.next.start()
...
...     def workbook(self,rdbook,wtbook_name):
...         print(self.name, 'workbook', rdbook, wtbook_name)
...         self.next.workbook(rdbook, wtbook_name)
...
...     def sheet(self,rdsheet,wtsheet_name):
...         print(self.name, 'sheet', rdsheet, wtsheet_name)
...         self.next.sheet(rdsheet, wtsheet_name)
...
...     def set_rdsheet(self,rdsheet):
...         print(self.name, 'set_rdsheet', rdsheet)
...         self.next.sheet(rdsheet, wtsheet_name)
...
...     def row(self,rdrowx,wtrowx):
...         print(self.name, 'row', rdrowx,wtrowx)
...         self.next.row(rdrowx, wtrowx)
...
...     def cell(self,rdrowx,rdcolx,wtrowx,wtcolx):
```

```
...         print(self.name, 'cell', rdrowx, rdcolx, wtrowx, wtcolx)
...         self.next.cell(rdrowx, rdcolx, wtrowx, wtcolx)
...
...     def finish(self):
...         print(self.name, 'finish')
...         self.next.finish()
```

For full details of when each of these methods are called, see the AP documentation for *BaseFilterInterface*.

CHAPTER 6

Writers

Writers are components that get handle calls from the appropriate methods on the last filter in the chain. It is the writer that actually does the work of copying all the information from the appropriate sources and writing them to the output files.

Because there is a lot of work involved in this and it is only usually the method of writing the binary data to its destination that differs, a handy base class is provided for writers.

Here's an example writer that just writes the data to a temporary directory:

```
>>> from xlutils.filter import BaseWriter
>>> our_temp_dir = TempDirectory().path
>>> class MyWriter(BaseWriter):
...
...     def get_stream(self,filename):
...         return open(os.path.join(our_temp_dir, filename), 'wb')
```

CHAPTER 7

Available Readers

Several reader implementations are included that cover many common use cases:

GlobReader

If you're processing files that are on disk, then this is probably the reader for you. It returns all files matching the path specification it's created with. Here's an example:

```
>>> from xlutils.filter import GlobReader
>>> r = GlobReader(os.path.join(test_files,'test*.xls'))
>>> sorted([p[len(test_files)+1:] for p in r.get_filepaths()])
['test.xls', 'testall.xls', 'testnoformatting.xls']
```

All the other functionality is provided by *BaseReader*:

```
>>> isinstance(r,BaseReader)
True
```

XLRDReader

If you want to "save" an `xlrd.Book` object which you've already created in some other way:

```
>>> from xlrd import open_workbook
>>> wb = open_workbook(os.path.join(test_files, 'testall.xls'))
```

Then the *XLRDReader* is likely what you're after:

```
>>> from xlutils.filter import XLRDReader
>>> r = XLRDReader(wb, 'foo.xls')
```

Note: You must supply a filename as shown above as the original filename is not stored in the `xlrd.Book` object

Most functionality is provided by *BaseReader*:

```
>>> isinstance(r, BaseReader)
True
```

However, its *get_workbooks()* method just returns the values it was instantiated with:

```
>>> tuple(r.get_workbooks())
((<xlrd...Book object at ...>, 'foo.xls'),)
>>> tuple(r.get_workbooks())[0][0] is wb
True
```

To show it working, here we send the book straight to a writer:

```
>>> from xlutils.filter import DirectoryWriter, process
>>> os.listdir(temp_dir)
[]
>>> process(r,DirectoryWriter(temp_dir))
>>> os.listdir(temp_dir)
['foo.xls']
```

TestReader

This reader is specifically designed for testing filter implementations with known sets of cells. This example should give a good idea of how to use it:

```
>>> from mock import Mock
>>> from pprint import pprint
>>> from xlutils.tests.test_filter import TestReader
>>> from xlrd import XL_CELL_NUMBER
>>> r = TestReader(
...          ('Sheet1',(('R0C0','R0C1'),
...                     ('R1C0','R1C1'))),
...          ('Sheet2',(((XL_CELL_NUMBER,0.0),),),),)
...      )
>>> c = Mock()
>>> r(c)
>>> pprint(c.method_calls)
[call.start(),
 call.workbook(<xlutils.tests.fixtures.DummyBook...>, 'test.xls'),
 call.sheet(<xlrd.sheet.Sheet...>, 'Sheet1'),
 call.row(0, 0),
 call.cell(0, 0, 0, 0),
 call.cell(0, 1, 0, 1),
 call.row(1, 1),
 call.cell(1, 0, 1, 0),
 call.cell(1, 1, 1, 1),
 call.sheet(<xlrd.sheet.Sheet...>, 'Sheet2'),
 call.row(0, 0),
 call.cell(0, 0, 0, 0),
 call.finish()]
```

CHAPTER 8

Available Writers

Several writer implementations are included that cover many common use cases:

DirectoryWriter

If you're processing files that are on disk, then this is probably the writer for you. It stores files in the directory passed to it during creation. Here's an example:

```
>>> from xlutils.filter import DirectoryWriter
>>> temp_dir = TempDirectory().path
>>> w = DirectoryWriter(temp_dir)
```

Most of the functionality is provided by *BaseWriter*:

```
>>> isinstance(w, BaseWriter)
True
```

The *get_stream()* method makes sure the files end up in the directory specified:

```
>>> os.listdir(temp_dir)
[]
>>> f = w.get_stream('test.xls')
>>> _ = f.write(b'some \r\n data')
>>> f.close()
>>> os.listdir(temp_dir)
['test.xls']
>>> open(os.path.join(temp_dir,'test.xls'),'rb').read()
b'some \r\n data'
```

StreamWriter

If you want to write exactly one workbook to a stream, then this is the writer for you:

```
>>> from tempfile import TemporaryFile
>>> from xlutils.filter import StreamWriter
>>> tf = TemporaryFile()
>>> w = StreamWriter(tf)
```

Most of the functionality is provided by *BaseWriter*:

```
>>> isinstance(w, BaseWriter)
True
```

The *get_stream()* method makes sure the excel data is written to the stream provided:

```
>>> f = w.get_stream('test.xls')
>>> _ = f.write(b'xls data')
>>> _ = tf.seek(0)
>>> tf.read()
b'xls data'
```

Note: Only one file may be written to a *StreamWriter*, further attempts will result in an exception being raised:

```
>>> w.get_stream('test2.xls')
Traceback (most recent call last):
...
Exception: Attempt to write more than one workbook
```

StreamWriter also doesn't close any streams passed to it:

```
>>> tf = TemporaryFile()
>>> process(TestReader(('Sheet1',[['R0C0']])),StreamWriter(tf))
>>> _ = tf.seek(0)
>>> len(tf.read())
5632
```

XLWTWriter

If you want to change cells after the filtering process is complete then an *XLWTWriter* can be used to obtain the `xlwt.Workbook` objects that result:

```
>>> from xlutils.filter import XLWTWriter
>>> w = XLWTWriter()
>>> process(TestReader(('Sheet1',[['R0C0']])),w)
```

The objects can then be manipulated and saved as required:

```
>>> w.output
[('test.xls', <xlwt.Workbook.Workbook object at ...>)]
```

```
>>> book = w.output[0][1]
>>> book.get_sheet(0).write(0,1,'R0C1')
```

```
>>> temp_dir = TempDirectory()
>>> temp_dir.listdir()
No files or directories found.
>>> book.save(os.path.join(temp_dir.path,w.output[0][0]))
>>> temp_dir.listdir()
test.xls
```

As with previous writers, most of the functionality is provided by *BaseWriter*:

```
>>> isinstance(w, BaseWriter)
True
```

CHAPTER 9

Available Filters

A selection of filters are included as described below:

BaseFilter

This is a "do nothing" filter that makes a great base class for your own filters. All the required methods are implemented such that they just call the same method on the next filter in the chain.

ColumnTrimmer

This filter will strip columns containing no useful data from the end of sheets. For example:

```
>>> from xlutils.filter import process, ColumnTrimmer
>>> r = TestReader(
...         ('Sheet1',(('','',''),
...                    ('R1C0','',''),
...                    ('R2C0','R2C1',' \t\r\n'))),
...         ('Sheet2',(('R0C0',),),),
...         ('Sheet3',(('R0C0','',(XL_CELL_NUMBER,0.0)),)),
...     )
>>> c = Mock()
>>> process(r,ColumnTrimmer(),c)
>>> pprint(c.method_calls)
[call.start(),
 call.workbook(<xlutils.tests.fixtures.DummyBook...>, 'test.xls'),
 call.sheet(<xlrd.sheet.Sheet...>, 'Sheet1'),
 call.row(0, 0),
 call.row(1, 1),
 call.row(2, 2),
 call.cell(0, 0, 0, 0),
 call.cell(0, 1, 0, 1),
```

```
 call.cell(1, 0, 1, 0),
 call.cell(1, 1, 1, 1),
 call.cell(2, 0, 2, 0),
 call.cell(2, 1, 2, 1),
 call.sheet(<xlrd.sheet.Sheet...>, 'Sheet2'),
 call.row(0, 0),
 call.cell(0, 0, 0, 0),
 call.sheet(<xlrd.sheet.Sheet...>, 'Sheet3'),
 call.row(0, 0),
 call.cell(0, 0, 0, 0),
 call.finish()]
```

When sheets are trimmed, a message is also logged to aid debugging:

```
>>> from testfixtures import LogCapture
>>> l = LogCapture()
>>> process(r, ColumnTrimmer(), c)
>>> print(l)
xlutils.filter DEBUG
  Number of columns trimmed from 3 to 2 for sheet b'Sheet1'
xlutils.filter DEBUG
  Number of columns trimmed from 3 to 1 for sheet b'Sheet3'
```

The definition of 'no useful data' can also be controlled by passing in a function that returns `True` or `False` for each value:

```
>>> def not_useful(cell):
...     if not cell.value or cell.value=='junk': return True
```

```
>>> r = TestReader(
...         ('Sheet1',(('R1C0','','junk'),)),
...     )
>>> c = Mock()
>>> process(r, ColumnTrimmer(not_useful), c)
>>> pprint(c.method_calls)
[call.start(),
 call.workbook(<xlutils.tests.fixtures.DummyBook...>, 'test.xls'),
 call.sheet(<xlrd.sheet.Sheet...>, 'Sheet1'),
 call.row(0, 0),
 call.cell(0, 0, 0, 0),
 call.finish()]
```

ErrorFilter

This filter caches all method calls in a file on disk and will only pass them on to filters further down in the chain when its *`finish`* method has been called *and* no error messages have been logged to the python logging framework.

Note: To be effective, this filter must be the last in the chain before the writer!

Here's an example of how to set one up. We need to be able to see what messages are logged, so we use a `LogCapture`:

```
>>> h = LogCapture()
```

Now, we install the filter:

```
>>> from xlutils.filter import process,ErrorFilter
>>> f = ErrorFilter()
```

To show the filter in action, we need a little helper:

```
>>> import logging
>>> from xlutils.filter import BaseFilter
>>> class Log(BaseFilter):
...     def __init__(self,level):
...         self.level = level
...     def workbook(self,rdbook,wtbook_name):
...         if wtbook_name=='test.xls':
...             logging.getLogger('theLogger').log(self.level,'a message')
...         self.next.workbook(rdbook,wtbook_name)
```

So, when we have errors logged, no methods other than finish are passed on to the next filter:

```
>>> c = Mock()
>>> process(MyReader('test.xls'), Log(logging.ERROR), f, c)
>>> len(c.method_calls)
0
```

As well as the error message logged, we can also see the *ErrorFilter* logs an error to that that the method calls have not been passed on:

```
>>> print(h)
theLogger ERROR
  a message
xlutils.filter ERROR
  No output as errors have occurred.
>>> h.clear()
```

This error message can be controlled when the *ErrorFilter* is instantiated:

```
>>> f = ErrorFilter(message='wingnuts! errors have occurred!')
>>> process(MyReader('test.xls'), Log(logging.ERROR), f, c)
>>> print(h)
theLogger ERROR
  a message
xlutils.filter ERROR
  wingnuts! errors have occurred!
```

However, when no errors are logged, all method calls are passed:

```
>>> c = Mock()
>>> process(MyReader('test.xls'), Log(logging.WARNING), f, c)
>>> len(c.method_calls)
17
```

In addition to the logging of error messages, error cells will also cause all methods to be filtered:

```
>>> from xlrd import XL_CELL_ERROR
>>> r = TestReader(
...         (u'Price(\xa3)',(((XL_CELL_ERROR,0),),)),
```

```
...          )
>>> c = Mock()
>>> h.clear()
>>> process(r,ErrorFilter(),c)
>>> len(c.method_calls)
0
>>> print(h)
xlutils.filter ERROR
  Cell A1 of sheet b'Price(?)' contains a bad value: error (#NULL!)
xlutils.filter ERROR
  No output as errors have occurred.
```

You can also configure the log level at which messages prevent the *ErrorFilter* from passing its method calls on to the next filter in the chain:

```
>>> f = ErrorFilter(logging.WARNING)
```

Now, warnings will cause methods to not be passed on:

```
>>> c = Mock()
>>> process(MyReader('test.xls'),Log(logging.WARNING),f,c)
>>> len(c.method_calls)
0
```

But if only debug messages are logged, the method calls will still be passed on:

```
>>> c = Mock()
>>> process(MyReader('test.xls'),Log(logging.DEBUG),f,c)
>>> len(c.method_calls)
17
```

An example which may prove useful is how to set up a filter such that if any errors are logged while processing one workbook, that workbook is filtered out but other subsequent workbooks are not filtered out.

This is done by inserting a filter such as the following earlier in the chain:

```
>>> class BatchByWorkbook(BaseFilter):
...     started = False
...     def start(self): pass
...     def workbook(self,rdbook,wtbook_name):
...         if self.started:
...             self.next.finish()
...         self.next.start()
...         self.next.workbook(rdbook,wtbook_name)
...         self.started = True
```

Here it is at work, starting with an empty output directory:

```
>>> temp_dir = TempDirectory().path
>>> os.listdir(temp_dir)
[]
```

Now `test.xls`, `testall.xls` and `testnoformatting.xls` are processed, but errors are only logged while processing `test.xls`:

```
>>> process(
...      GlobReader(os.path.join(test_files,'test*.xls')),
...      BatchByWorkbook(),
```

```
...     Log(logging.ERROR),
...     ErrorFilter(),
...     DirectoryWriter(temp_dir)
... )
```

So, the output directory contains `testall.xls`, but no `test.xls`:

```
>>> sorted(os.listdir(temp_dir))
['testall.xls', 'testnoformatting.xls']
```

Echo

This filter will print calls to the methods configured when the filter is created along with the arguments passed.

```
>>> from xlutils.filter import Echo, process
>>> r = TestReader(
...         ('Sheet1',(('R0C0','R0C1'),
...                    ('R1C0','R1C1'))),
...     )
>>> process(r, Echo(methods=('workbook',)), Mock())
workbook:(<...DummyBook...>, 'test.xls')
```

If `True` is passed instead of a list of method names, then all methods called will be printed:

```
>>> process(r, Echo(methods=True), Mock())
start:()
workbook:(<...DummyBook...>, 'test.xls')
sheet:(<xlrd.sheet.Sheet...>, 'Sheet1')
row:(0, 0)
cell:(0, 0, 0, 0)
cell:(0, 1, 0, 1)
row:(1, 1)
cell:(1, 0, 1, 0)
cell:(1, 1, 1, 1)
finish:()
```

If you need to see what's happening at various points in a chain, you can also give an *Echo* a name:

```
>>> process(r,Echo('first'), Echo('second'), Mock())
'first' start:()
'second' start:()
'first' workbook:(<...DummyBook...>, 'test.xls')
'second' workbook:(<...DummyBook...>, 'test.xls')
'first' sheet:(<xlrd.sheet.Sheet...>, 'Sheet1')
'second' sheet:(<xlrd.sheet.Sheet...>, 'Sheet1')
'first' row:(0, 0)
'second' row:(0, 0)
'first' cell:(0, 0, 0, 0)
'second' cell:(0, 0, 0, 0)
'first' cell:(0, 1, 0, 1)
'second' cell:(0, 1, 0, 1)
'first' row:(1, 1)
'second' row:(1, 1)
'first' cell:(1, 0, 1, 0)
'second' cell:(1, 0, 1, 0)
```

```
'first' cell:(1, 1, 1, 1)
'second' cell:(1, 1, 1, 1)
'first' finish:()
'second' finish:()
```

MemoryLogger

This filter will dump stats to the path it was configured with using the heapy package if it is available. If it is not available, no operations are performed.

For example, with a *MemoryLogger* configured as follows:

```
>>> from xlutils.filter import MemoryLogger
>>> m = MemoryLogger('/some/path', methods=('sheet','cell'))
```

The equivalent of the following call:

```
from guppy import hpy; hpy().heap().stat.dump('/some/path')
```

will be performed whenever the *MemoryLogger*'s `sheet()` and `cell()` methods are called.

A *MemoryLogger* configured as followed will log memory usage whenever any of the *MemoryLogger*'s methods are called:

```
>>> m = MemoryLogger('/some/path',True)
```

For more information on heapy, please see:

http://guppy-pe.sourceforge.net/#Heapy

CHAPTER 10

Example Usage

Here's an example that makes use of all the types of components described above to filter out odd numbered rows from an original workbook's sheets. To do this we need one more filter:

```
>>> from xlutils.filter import BaseFilter
>>> class EvenFilter(BaseFilter):
...
...     def row(self,rdrowx,wtrowx):
...         if not rdrowx%2:
...             self.next.row(rdrowx,wtrowx)
...
...     def cell(self,rdrowx,rdcolx,wtrowx,wtcolx):
...         if not rdrowx%2:
...             self.next.cell(rdrowx,rdcolx,wtrowx,wtcolx)
```

Now we can put it all together with a call to the *process()* function:

```
>>> from xlutils.filter import process
>>> process(
...     MyReader('test.xls'),
...     MyFilter('before'),
...     EvenFilter(),
...     MyFilter('after'),
...     MyWriter()
...     )
before start
after start
before workbook <xlrd...Book object at ...> test.xls
after workbook <xlrd...Book object at ...> test.xls
before sheet <xlrd.sheet.Sheet object at ...> Sheet1
after sheet <xlrd.sheet.Sheet object at ...> Sheet1
before row 0 0
after row 0 0
before cell 0 0 0 0
after cell 0 0 0 0
before cell 0 1 0 1
```

```
after cell 0 1 0 1
before row 1 1
before cell 1 0 1 0
before cell 1 1 1 1
before sheet <xlrd.sheet.Sheet object at ...> Sheet2
after sheet <xlrd.sheet.Sheet object at ...> Sheet2
before row 0 0
after row 0 0
before cell 0 0 0 0
after cell 0 0 0 0
before cell 0 1 0 1
after cell 0 1 0 1
before row 1 1
before cell 1 0 1 0
before cell 1 1 1 1
before finish
after finish
```

As you can see if you've read this far, there's quite a lot of output, but it's certainly informative! However, just to be on the safe side, we can see that the output file was actually written:

```
>>> os.listdir(our_temp_dir)
['test.xls']
```

CHAPTER 11

xlutils.margins

This combined module and script provide information on how much space is taken up in an Excel file by cells containing no meaningful data. If `xlutils` is installed using `easy_install`, `pip` or `zc.buildout`, a console script called `margins` will be created.

The following example shows how it is used as a script:

```
python -m xlutils.margins [options] *.xls
```

To get a list of the available options, do the following:

```
python -m xlutils.margins --help
```

The module also provides the tools that do the work for the above script as the helpful *`ispunc()`*, *`cells_all_junk()`*, *`number_of_good_rows()`* and *`number_of_good_cols()`* functions.

See the *API Reference* for more information.

CHAPTER 12

xlutils.save

This allows serialisation of `xlrd.Book` objects back into binary Excel files.

Here's a simple example:

```
>>> import os
>>> from xlrd import open_workbook
>>> from xlutils.save import save
>>> wb = open_workbook(os.path.join(test_files,'testall.xls'))
>>> os.listdir(temp_dir)
[]
>>> save(wb,os.path.join(temp_dir,'saved.xls'))
>>> os.listdir(temp_dir)
['saved.xls']
```

You can also save the data to a stream that you provide:

```
>>> from xlutils.compat import BytesIO
>>> s = BytesIO()
>>> save(wb,s)
>>> len(s.getvalue())
5632
```

CHAPTER 13

xlutils.styles

This module provides tools for working with formatting information provided by `xlrd` relating and expressed in the Excel file as styles.

To use these tools, you need to open the workbook with `xlrd` and make sure formatting is enabled:

```
>>> import os
>>> from xlrd import open_workbook
>>> book = open_workbook(os.path.join(test_files,'testall.xls'), formatting_info=1)
```

Once you have a `Book` object, you can extract the relevent style information from it as follows:

```
>>> from xlutils.styles import Styles
>>> s = Styles(book)
```

You can now look up style information about any cell:

```
>>> sheet = book.sheet_by_name('Sheet1')
>>> s[sheet.cell(0,0)]
<xlutils.styles.NamedStyle ...>
```

Note: This is *not* a suitable object for copying styles to a new spreadsheet using `xlwt`. If that is your intention, you're recommended to look at *xlutils.save* or *xlutils.filter*.

If you open up `testall.xls` in Microsoft's Excel Viewer or other suitable software, you'll see that the following information is correct for cell A1:

```
>>> A1_style = s[sheet.cell(0,0)]
>>> A1_style.name
u'Style1'
```

While that may be interesting, the actual style information is locked away in an `XF` record. Thankfully, a *`NamedStyle`* provides easy access to this:

```
>>> A1_xf = A1_style.xf
>>> A1_xf
<xlrd.formatting.XF ...>
```

Once we have the XF record, for this particular cell, most of the interesting information is in the font definition for the style:

```
>>> A1_font = book.font_list[A1_xf.font_index]
```

Using the book's colour map, you can get the RGB colour for this style, which is blue in this case:

```
>>> book.colour_map[A1_font.colour_index]
(0, 0, 128)
```

You can also see that this style specifies text should be underlined with a single line:

```
>>> A1_font.underline_type
1
```

Finally, the style specifies that text is not displayed with a "strike through" line:

```
>>> A1_font.struck_out
0
```

For completeness, here's the same information but for cell B1:

```
>>> B1_style = s[sheet.cell(0,1)]
>>> B1_style.name
u'Style2'
>>> B1_font = book.font_list[B1_style.xf.font_index]
```

In this case, though, the style's colour is green:

```
>>> book.colour_map[B1_font.colour_index]
(0, 128, 0)
```

The style specifies that text should not be underlined:

```
>>> B1_font.underline_type
0
```

And finally, it specifies that text should be displayed with a "strike through" line:

```
>>> B1_font.struck_out
1
```

CHAPTER 14

xlutils.view

Iterating over the cells in a `Sheet` can be tricky, especially if you want to exclude headers and the like. This module is designed to make things easier.

For example, to iterate over the cells in the first sheet of a workbook:

```
>>> def print_data(rows):
...     for row in rows:
...         for value in row:
...             print(value, end=' ')
...         print()
```

```
>>> from os.path import join
>>> from xlutils.view import View
>>> view = View(join(test_files,'testall.xls'))
>>> print_data(view[0])
R0C0 R0C1
R1C0 R1C1
A merged cell


More merged cells
```

You can also get a sheet by name:

```
>>> print_data(view['Sheet2'])
R0C0 R0C1
R1C0 R1C1
```

One helpful feature is that dates are converted to `datetime` objects rather than being left as numbers:

```
>>> for row in View(join(test_files,'datetime.xls'))[0]:
...     for value in row:
...         print(repr(value))
datetime.datetime(2012, 4, 13, 0, 0)
```

```
datetime.time(12, 54, 37)
datetime.datetime(2014, 2, 14, 4, 56, 23)
```

Now, things get really interesting when you start slicing the view of a sheet:

```
>>> print_data(view['Sheet1'][:2, :1])
R0C0
R1C0
```

As you can see, these behave exactly as slices into lists would, with the first slice being on rows and the second slice being on columns.

Since looking at a sheet and working with the row and column labels shown is much easier, *Row* and *Col* helpers are provided. When these are used for the `stop` part of a slice, they are inclusive. For example:

```
>>> from xlutils.view import Row, Col
>>> print_data(view['Sheet1'][Row(1):Row(2), Col('A'):Col('B')])
R0C0 R0C1
R1C0 R1C1
```

Finally, to aid with automated tests, there is a *CheckerView* subclass of *View* that provides *CheckSheet* views onto sheets in a workbook. These have a *compare()* method that produces informative `AssertionError` exceptions when the data in the view of the sheet is not as expected:

```
>>> from xlutils.view import CheckerView
>>> sheet_view = CheckerView(join(test_files,'testall.xls'))[0]
>>> sheet_view[:, Col('A'):Col('A')].compare(
...     (u'R0C0', ),
...     (u'R0C1', ),
... )
Traceback (most recent call last):
...
AssertionError: sequence not as expected:

same:
((u'R0C0',),)

expected:
((u'R0C1',),)

actual:
((u'R1C0',), (u'A merged cell',), (u'',), (u'',), (u'More merged cells',))

While comparing [1]: sequence not as expected:

same:
()

expected:
(u'R0C1',)

actual:
(u'R1C0',)

While comparing [1][0]: u'R0C1' (expected) != u'R1C0' (actual)
```

Use of the *compare()* method requires testfixtures to be installed.

Looking at the implementation of *CheckerView* will also show you how you can wire in *SheetView* subclasses to provide any extra methods you may require.

CHAPTER 15

Working with xlutils

The following sections describe how to install the package, contribute to its development and the usual boilerplate:

Installation Instructions

If you want to experiment with xlutils, the easiest way to install it is:

```
pip install xlutils
```

Or, if you're using *zc.buildout*, just specify `xlutils` as a required egg.

If you do not install using easy_install or zc.buildout, you will also need to make sure the following python packages are available on your PYTHONPATH:

- **xlrd**

  You'll need version 0.7.2 or later. Latest versions can be found here:

  http://pypi.python.org/pypi/xlrd

- **xlwt**

  You'll need version 0.7.3 or later. Latest versions can be found here:

  http://pypi.python.org/pypi/xlwt

If you're installing with pip, easy_install or buildout, these dependencies will automatically be installed for you.

Additionally, if you want to use an *`ErrorFilter`*, you should make sure the following package is installed:

- **errorhandler**

  This can be found here:

  http://pypi.python.org/pypi/errorhandler

Since this is a soft dependency, it will not be installed by automatically by pip, easy_install or buildout.

Development

This package is developed using continuous integration which can be found here:

https://travis-ci.org/python-excel/xlutils

If you wish to contribute to this project, then you should fork the repository found here:

https://github.com/python-excel/xlutils

Development of this package also requires local clones of both `xlrd` and `xlwt`. The following example will set up local clones as required, but you should use your own forks so that you may push back to them:

```
git clone git://github.com/python-excel/xlutils.git
git clone git://github.com/python-excel/xlrd.git
git clone git://github.com/python-excel/xlwt.git
cd xlutils
```

Once you have an appropriate set of local repositories, you can follow these instructions to perform various development tasks:

Setting up a virtualenv

The recommended way to set up a development environment is to turn your checkout into a virtualenv and then install the package in editable form as follows:

```
$ virtualenv .
$ bin/pip install -U -e .[test,build]
```

You will now also need to install xlrd and xlwt into the virtualenv:

```
$ source bin/activate
$ cd ../xlrd
$ pip install -e .
$ cd ../xlwt
$ pip install -e .
```

Running the tests

Once you've set up a virtualenv, the tests can be run as follows:

```
$ bin/nosetests
```

Building the documentation

The Sphinx documentation is built by doing the following from the directory containg setup.py:

```
cd docs
make html
```

Making a release

To make a release, just update `xlutils/version.txt`, update the change log, tag it and push to https://github.com/python-excel/xlutils and Travis CI should take care of the rest.

Once the above is done, make sure to go to https://readthedocs.org/projects/xlutils/versions/ and make sure the new release is marked as an Active Version.

API Reference

`xlutils.copy.`**`copy`**(*wb*)

Copy an `xlrd.Book` into an `xlwt.Workbook` preserving as much information from the source object as possible.

See the *xlutils.copy* documentation for an example.

`xlutils.display.`**`quoted_sheet_name`**(*sheet_name*, *encoding='ascii'*)

This returns a string version of the supplied sheet name that is safe to display, including encoding the unicode sheet name into a string:

```
>>> from xlutils.display import quoted_sheet_name
>>> quoted_sheet_name(u'Price(\xa3)','utf-8')
b'Price(\xc2\xa3)'
```

It also quotes the sheet name if it contains spaces:

```
>>> quoted_sheet_name(u'My Sheet')
b"'My Sheet'"
```

Single quotes are replaced with double quotes:

```
>>> quoted_sheet_name(u"John's Sheet")
b"'John''s Sheet'"
```

`xlutils.display.`**`cell_display`**(*cell*, *datemode=0*, *encoding='ascii'*)

This returns a string representation of the supplied cell, no matter what type of cell it is. Here's some example output:

```
>>> import xlrd
>>> from xlrd.sheet import Cell
>>> from xlutils.display import cell_display
>>> from xlutils.compat import PY3
```

```
>>> cell_display(Cell(xlrd.XL_CELL_EMPTY, ''))
'undefined'
```

```
>>> cell_display(Cell(xlrd.XL_CELL_BLANK, ''))
'blank'
```

```
>>> cell_display(Cell(xlrd.XL_CELL_NUMBER, 1.2))
'number (1.2000)'
```

```
>>> cell_display(Cell(xlrd.XL_CELL_BOOLEAN, 0))
'logical (FALSE)'
```

```
>>> cell_display(Cell(xlrd.XL_CELL_DATE, 36892.0))
'date (2001-01-01 00:00:00)'
```

Erroneous date values will be displayed like this:

```
>>> cell_display(Cell(xlrd.XL_CELL_DATE, 1.5))
'date? (1.500000)'
```

Note: To display dates correctly, make sure that *datemode* is passed and is taken from the *datemode* attribute of the `xlrd.Book` from which the cell originated as shown below:

```
>>> wb = open_workbook(join(test_files,'date.xls'))
>>> cell = wb.sheet_by_index(0).cell(0, 0)
>>> cell_display(cell, wb.datemode)
'date (2012-04-13 00:00:00)'
```

If non-unicode characters are to be displayed, they will be masked out:

```
>>> cd = cell_display(Cell(xlrd.XL_CELL_TEXT,u'Price (\xa3)'))
>>> if PY3:
...     str(cd) == "text (b'Price (?)')"
... else:
...     str(cd) == 'text (Price (?))'
True
```

If you want to see these characters, specify an encoding for the output string:

```
>>> cd = cell_display(Cell(xlrd.XL_CELL_TEXT,u'Price (\xa3)'), encoding='utf-8')
>>> if PY3:
...     str(cd) == "text (b'Price (\\xc2\\xa3)')"
... else:
...     str(cd) == 'text (Price (\xc2\xa3))'
True
```

Error cells will have their textual description displayed:

```
>>> cell_display(Cell(xlrd.XL_CELL_ERROR, 0))
'error (#NULL!)'
```

```
>>> cell_display(Cell(xlrd.XL_CELL_ERROR, 2000))
'unknown error code (2000)'
```

If you manage to pass a cell with an unknown cell type, an exception will be raised:

```
>>> cell_display(Cell(69, 0))
Traceback (most recent call last):
...
Exception: Unknown Cell.ctype: 69
```

class `xlutils.filter.`**`BaseFilter`**

A concrete filter that implements pass-through behaviour for *`BaseFilterInterface`*.

This often makes a great base class for your own filters.

class `xlutils.filter.`**`BaseFilterInterface`**

This is the filter interface that shows the correct way to call the next filter in the chain. The *next* attribute is set

up by the *process()* function. It can make a good base class for a new filter, but subclassing *BaseFilter* is often a better idea!

cell(*rdrowx*, *rdcolx*, *wtrowx*, *wtcolx*)
This is called for every cell in the sheet being processed. This is the most common method in which filtering and queuing of onward calls to the next filter takes place.

Parameters

- **rdrowx** – the index of the row to be read from in the current sheet.
- **rdcolx** – the index of the column to be read from in the current sheet.
- **wtrowx** – the index of the row to be written to in the current output sheet.
- **wtcolx** – the index of the column to be written to in the current output sheet.

finish()
This method is called once processing of all workbooks has been completed.

A filter should call this method on the next filter in the chain as an indication that no further calls will be made to any methods or that, if they are, any new calls should be treated as new batch of workbooks with no information retained from the previous batch.

row(*rdrowx*, *wtrowx*)
This is called every time processing of a new row in the current sheet starts. It is primarily for copying row-based formatting from the source row to the target row.

Parameters

- **rdrowx** – the index of the row in the current sheet from which information for the row to be written should be copied.
- **wtrowx** – the index of the row in sheet to be written to which information should be written for the row being read.

set_rdsheet(*rdsheet*)
This is only ever called by a filter that wishes to change the source of cells mid-way through writing a sheet.

Parameters **rdsheet** – the Sheet object from which cells from this point forward should be read from.

sheet(*rdsheet*, *wtsheet_name*)
This method is called every time processing of a new sheet in the current workbook starts.

Parameters

- **rdsheet** – the Sheet object from which the new sheet should be created.
- **wtsheet_name** – the name of the sheet into which content should be written.

start()
This method is called before processing of a batch of input. This allows the filter to initialise any required data structures and dispose of any existing state from previous batches.

It is called once before the processing of any workbooks by the included reader implementations.

This method can be called at any time. One common use is to reset all the filters in a chain in the event of an error during the processing of a *rdbook*.

Implementations of this method should be extremely robust and must ensure that they call the *start()* method of the next filter in the chain regardless of any work they do.

workbook(*rdbook*, *wtbook_name*)
This method is called every time processing of a new workbook starts.

Parameters

- **rdbook** – the `Book` object from which the new workbook should be created.
- **wtbook_name** – the name of the workbook into which content should be written.

class `xlutils.filter.`**BaseReader**
A base reader good for subclassing.

__call__(*filter*)
Once instantiated, a reader will be called and have the first filter in the chain passed to its *__call__()* method. The implementation of this method should call the appropriate methods on the filter based on the cells found in the `Book` objects returned from the *get_workbooks()* method.

get_filepaths()
This is the most common method to implement. It must return an interable sequence of paths to excel files.

get_workbooks()
If the data to be processed is not stored in files or if special parameters need to be passed to `xlrd.open_workbook()` then this method must be overriden. Any implementation must return an iterable sequence of tuples. The first element of which must be an `xlrd.Book` object and the second must be the filename of the file from which the book object came.

class `xlutils.filter.`**BaseWriter**
This is the base writer that copies all data and formatting from the specified sources. It is designed for sequential use so when, for example, writing two workbooks, the calls must be ordered as follows:

- *workbook()* call for first workbook
- *sheet()* call for first sheet
- *row()* call for first row
- *cell()* call for left-most cell of first row
- *cell()* call for second-left-most cell of first row
- ...
- *row()* call for second row
- ...
- *sheet()* call for second sheet
- ...
- *workbook()* call for second workbook
- ...
- *finish()* call

Usually, only the *get_stream()* method needs to be implemented in subclasses.

cell(*rdrowx*, *rdcolx*, *wtrowx*, *wtcolx*)
This should be called for every cell in the sheet being processed.

Parameters

- **rdrowx** – the index of the row to be read from in the current sheet.
- **rdcolx** – the index of the column to be read from in the current sheet.

- **wtrowx** – the index of the row to be written to in the current output sheet.
- **wtcolx** – the index of the column to be written to in the current output sheet.

finish()
This method should be called once processing of all workbooks has been completed.

get_stream(*filename*)
This method is called once for each file written. The filename of the file to be created is passed and something with `write()` and `close()` methods that behave like a `file` object's must be returned.

row(*rdrowx*, *wtrowx*)
This should be called every time processing of a new row in the current sheet starts.

Parameters

- **rdrowx** – the index of the row in the current sheet from which information for the row to be written will be copied.
- **wtrowx** – the index of the row in sheet to be written to which information will be written for the row being read.

set_rdsheet(*rdsheet*)
This should only ever called by a filter that wishes to change the source of cells mid-way through writing a sheet.

Parameters **rdsheet** – the `Sheet` object from which cells from this point forward will be read.

sheet(*rdsheet*, *wtsheet_name*)
This method should be called every time processing of a new sheet in the current workbook starts.

Parameters

- **rdsheet** – the `Sheet` object from which the new sheet will be created.
- **wtsheet_name** – the name of the sheet into which content will be written.

start()
This method should be called before processing of a batch of input. This allows the filter to initialise any required data structures and dispose of any existing state from previous batches.

workbook(*rdbook*, *wtbook_name*)
This method should be called every time processing of a new workbook starts.

Parameters

- **rdbook** – the `Book` object from which the new workbook will be created.
- **wtbook_name** – the name of the workbook into which content will be written.

class xlutils.filter.**ColumnTrimmer**(*is_junk=None*)
This filter will strip columns containing no useful data from the end of sheets.

See the *ColumnTrimmer* documentation for an example.

class xlutils.filter.**DirectoryWriter**(*path*)
A writer that stores files in a filesystem directory

get_stream(*filename*)
Returns a stream for the file in the configured directory with the specified name.

class xlutils.filter.**Echo**(*name=None*, *methods=True*)
This filter will print calls to the methods configured when the filter is created along with the arguments passed.

For more details, see the *documentation*.

class xlutils.filter.**ErrorFilter**(*level=40*, *message='No output as errors have occurred.'*)
A filter that gates downstream writers or filters on whether or not any errors have occurred.

See *ErrorFilter* for details.

finish()
The method that triggers downstream filters and writers if no errors have occurred.

class xlutils.filter.**GlobReader**(*spec*)
A reader that emits events for all files that match the glob in the spec.

class xlutils.filter.**MemoryLogger**(*path*, *methods=True*)
This filter will dump stats to the path it was configured with using the heapy package if it is available.

class xlutils.filter.**MethodFilter**(*methods=True*)
This is a base class that implements functionality for filters that want to do a common task such as logging, printing or memory usage recording on certain calls configured at filter instantiation time.

Echo is an example of this.

method(*name*, **args*)
This is the method that needs to be implemented. It is called with the name of the method that was called on the MethodFilter and the arguments that were passed to that method.

class xlutils.filter.**StreamWriter**(*stream*)
A writer for writing exactly one workbook to the supplied stream

get_stream(*filename*)
Returns the stream passed during instantiation.

class xlutils.filter.**XLRDReader**(*wb*, *filename*)
A reader that uses an in-memory `xlrd.Book` object as its source of events.

get_workbooks()
Yield the workbook passed during instantiation.

class xlutils.filter.**XLWTWriter**
A writer that writes to a sequence of in-memory `xlwt.Workbook` objects.

xlutils.filter.**process**(*reader*, **chain*)
The driver function for the *xlutils.filter* module.

It takes a chain of one *reader*, followed by zero or more *filters* and ending with one *writer*.

All the components are chained together by the *process()* function setting their `next` attributes appropriately. The *reader* is then called with the first *filter* in the chain.

xlutils.margins.**ispunc**(*character*)
This little helper function returns `True` if called with a punctuation character and `False` with any other:

```
>>> from xlutils.margins import ispunc
>>> ispunc('u')
False
>>> ispunc(',')
True
```

It also works fine with unicode characters:

```
>>> ispunc(u',')
True
>>> ispunc(u'w')
False
```

It does not, however, return sensible answers if called with more than one character:

```
>>> ispunc(',,,')
False
```

xlutils.margins.**cells_all_junk**(*cells*, *is_rubbish=None*)

Return True if all cells in the sequence are junk. What qualifies as junk: – empty cell – blank cell – zero-length text – text is all whitespace – number cell and is 0.0 – text cell and is_rubbish(cell.value) returns True.

This function returns `True` if all the cells supplied are junk:

```
>>> from xlutils.margins import cells_all_junk
>>> from xlrd.sheet import Cell,empty_cell
>>> cells_all_junk([empty_cell,empty_cell,empty_cell])
True
```

But it returns `False` as soon as any of the cells supplied are not junk:

```
>>> from xlrd import XL_CELL_NUMBER
>>> cells_all_junk([empty_cell,Cell(XL_CELL_NUMBER,1),empty_cell])
False
```

The definition of 'junk' is as follows:

•Empty cells are junk:

```
>>> from xlrd import XL_CELL_EMPTY
>>> cells_all_junk([Cell(XL_CELL_EMPTY,'')])
True
```

•Blank cells are junk:

```
>>> from xlrd import XL_CELL_BLANK
>>> cells_all_junk([Cell(XL_CELL_BLANK,'')])
True
```

•Number cells containing zero are considered junk:

```
>>> from xlrd import XL_CELL_NUMBER
>>> cells_all_junk([Cell(XL_CELL_NUMBER,0)])
True
```

However, if a number cell contains anything else, it's not junk:

```
>>> cells_all_junk([Cell(XL_CELL_NUMBER,1)])
False
```

•Text cells are junk if they don't contain anything:

```
>>> from xlrd import XL_CELL_TEXT
>>> cells_all_junk([Cell(XL_CELL_TEXT,'')])
True
```

or if they contain only space characters:

```
>>> cells_all_junk([Cell(XL_CELL_TEXT,' \t\n\r')])
True
```

otherwise they aren't considered junk:

```
>>> cells_all_junk([Cell(XL_CELL_TEXT,'not junk')])
False
```

However, you can also pass a checker function such as this one:

```
>>> def isrubbish(cell): return cell.value=='rubbish'
```

Which can then be used to check for junk conditions of your own choice:

```
>>> cells_all_junk([Cell(XL_CELL_TEXT,'rubbish')],isrubbish)
True
>>> cells_all_junk([Cell(XL_CELL_TEXT,'not rubbish')],isrubbish)
False
```

Passing a function like this isn't only limited to text cells:

```
>>> def isnegative(cell): return isinstance(cell.value,float) and cell.value
↪<0 or False
```

```
>>> cells_all_junk([Cell(XL_CELL_NUMBER,-1.0)],isnegative)
True
>>> cells_all_junk([Cell(XL_CELL_NUMBER,1.0)],isnegative)
False
```

•Date, boolean, and error fields are all not considered to be junk:

```
>>> from xlrd import XL_CELL_DATE, XL_CELL_BOOLEAN, XL_CELL_ERROR
>>> cells_all_junk([Cell(XL_CELL_DATE,'')])
False
>>> cells_all_junk([Cell(XL_CELL_BOOLEAN,'')])
False
>>> cells_all_junk([Cell(XL_CELL_ERROR,'')])
False
```

Be careful, though, as if you call *cells_all_junk()* with an empty sequence of cells, you'll get True:

```
>>> cells_all_junk([])
True
```

xlutils.margins.**number_of_good_rows**(*sheet*, *checker=None*, *nrows=None*, *ncols=None*)

Return 1 + the index of the last row with meaningful data in it.

This function returns the number of rows in a sheet that contain anything other than junk, as defined by the *cells_all_junk()* function.

For example:

```
>>> from xlutils.tests.fixtures import make_sheet
>>> sheet = make_sheet((
...           ('X',' ',' ',' ',' '),
...           (' ',' ',' ','X',' '),
...           (' ',' ',' ',' ',' '),
...           ('X',' ',' ',' ',' '),
...           (' ',' ','X',' ',' '),
...           (' ',' ',' ',' ',' '),
...           ))
>>> from xlutils.margins import number_of_good_rows
```

```
>>> number_of_good_rows(sheet)
5
```

You can limit the area searched using the *nrows* and *ncols* parameters:

```
>>> number_of_good_rows(sheet,nrows=3)
2
>>> number_of_good_rows(sheet,ncols=2)
4
>>> number_of_good_rows(sheet,ncols=3,nrows=3)
1
```

You can also pass a checking function through to the *cells_all_junk()* calls:

```
>>> number_of_good_rows(sheet,checker=lambda c:c.value=='X')
0
```

xlutils.margins.**number_of_good_cols**(*sheet*, *checker=None*, *nrows=None*, *ncols=None*)

Return 1 + the index of the last column with meaningful data in it.

This function returns the number of columns in a sheet that contain anything other than junk, as defined by the *cells_all_junk()* function.

For example:

```
>>> sheet = make_sheet((
...             ('X',' ',' ','X',' ',' '),
...             (' ',' ',' ',' ',' ',' '),
...             (' ',' ',' ',' ','X',' '),
...             (' ','X',' ',' ',' ',' '),
...             (' ',' ',' ',' ',' ',' '),
...             (' ',' ',' ',' ',' ',' '),
...             ))
>>> from xlutils.margins import number_of_good_cols
>>> number_of_good_cols(sheet)
5
```

You can limit the area searched using the *nrows* and *ncols* parameters:

```
>>> number_of_good_cols(sheet,nrows=2)
4
>>> number_of_good_cols(sheet,ncols=2)
2
>>> number_of_good_cols(sheet,ncols=3,nrows=3)
1
```

You can also pass a checking function through to the *cells_all_junk()* calls:

```
>>> number_of_good_cols(sheet,checker=lambda c:c.value=='X')
0
```

xlutils.save.**save**(*wb*, *filename_or_stream*)

Save the supplied `xlrd.Book` to the supplied stream or filename.

class xlutils.styles.**NamedStyle**(*name*, *xf*)

An object with `name` and `xf` attributes representing a particular style in a workbook.

class xlutils.styles.**Styles**(*book*)

A mapping-like object that will return a *NamedStyle* instance for the cell passed to the `__getitem__()`

method.

class xlutils.view.**CheckSheet**(*book*, *sheet*, *row_slice=None*, *col_slice=None*)
A special sheet view for use in automated tests.

compare(**expected*)
Call to check whether this view contains the expected data. If it does not, a descriptive AssertionError will be raised. Requires testfixtures.

Parameters **expected** – tuples containing the data that should be present in this view.

class xlutils.view.**CheckerView**(*path*, *class_=None*)
A special subclass of *View* for use in automated tests when you want to check the contents of a generated spreadsheet.

Views of sheets are returned as *CheckSheet* instances which have a handy *compare()* method.

class_
alias of *CheckSheet*

class xlutils.view.**Col**(*name*)
An end-inclusive column label index for use in slices, eg: `[:, Col('A'), Col('B')]`

class xlutils.view.**Row**(*name*)
A one-based, end-inclusive row index for use in slices, eg:: `[Row(1):Row(2), :]`

class xlutils.view.**SheetView**(*book*, *sheet*, *row_slice=None*, *col_slice=None*)
A view on a sheet in a workbook. Should be created by indexing a *View*.

These can be sliced to create smaller views.

Views can be iterated over to return a set of iterables, one for each row in the view. Data is returned as in the cell values with the exception of dates and times which are converted into datetime instances.

__weakref__
list of weak references to the object (if defined)

book = None
The workbook used by this view.

sheet = None
The sheet in the workbook used by this view.

class xlutils.view.**View**(*path*, *class_=None*)
A view wrapper around a Book that allows for easy iteration over the data in a group of cells.

Parameters

- **path** – The path of the .xls from which to create views.
- **class** – An class to use instead of *SheetView* for views of sheets.

__getitem__(*item*)
Returns of a view of a sheet in the workbook this view is created for.

Parameters **item** – either zero-based integer index or a sheet name.

__weakref__
list of weak references to the object (if defined)

class_
This can be replaced in a sub-class to use something other than *SheetView* for the views of sheets returned.

alias of *SheetView*

Changes

2.0.0 (9 June 2016)

- Updated documentation.
- Move to virtualenv/pip based development.
- Move to Read The Docs for documentation.
- Use Travis CI for testing and releases.
- Use features of newer `testfixtures` in *`CheckerView`*.
- Python 3 compatibility.

1.7.1 (25 April 2014)

- Add support for time cells in *`View`*.
- Add support for `.xlsx` files in *`View`* at the expense of formatting information being available.

1.7.0 (29 October 2013)

- Added the *`xlutils.view`* module.

1.6.0 (5 April 2013)

- Moved documentation to be Sphinx based.
- Support for `xlrd` 0.9.1, which no longer has pickleable books.

Note: You may encounter performance problems if you work with large spreadsheets and use `xlutils` 1.6.0 with `xlrd` versions earlier that 0.9.1.

1.5.2 (13 April 2012)

- When using *`xlutils.copy`*, the `datemode` is now copied across from the source solving date problems with certain files.
- The `errorhandler` package is no longer a hard dependency.
- As a minimum, `xlrd` 0.7.2 and `xlwt` 0.7.4 are now required.

1.5.1 (5 March 2012)

- Fix packaging problem caused by the move to git

1.5.0 (5 March 2012)

- Take advantage of "ragged rows" optimisation in xlrd 0.7.3
- Add support for PANE records to *xlutils.copy*, which means that zoom factors are now copied.

1.4.1 (6 September 2009)

- Removal of references in the *finish* methods of several filters, easing memory usage in large filtering runs
- Speed optimisations for *BaseFilter*, bringing those benefits to all subclasses.
- Memory usage reduction when using *MemoryLogger*

1.4.0 (18 August 2009)

- Add sheet density information and onesheet option to *xlutils.margins*.
- Reduced the memory footprint of *ColumnTrimmer* at the expense of speed.
- Fixed incorrect warnings about boolean cells in *ErrorFilter*. `xlwt` has always supported boolean cells.
- *BaseReader* now opens workbooks with `on_demand = True`
- Added support for `xlrd` Books opened with `on_demand` as `True` passed to `open_workbook()`.
- Fixed bug when copying error cells.
- Requires the latest versions of `xlrd` (0.7.1) and `xlwt` (0.7.2).

1.3.2 (18 June 2009)

- Made installation work when *setuptools* isn't present.
- Made *errorhandler* an optional dependency.

1.3.1 (22 May 2009)

- In xlutils.styles, handle there case where two names were mapped to the same xfi, but the first one was empty.

1.3.0 (18 Mar 2009)

- fix bug that cause BaseWriter to raise exceptions when handling source workbooks opened by xlrd 0.7.0 and above where formatting_info had been passed as False
- add xlutils.copy

1.2.1 (19 Dec 2008)

- add extremely limited formatting_info support to DummyBook and TestReader
- move to testfixtures 1.5.3 for tests

1.2.0 (10 Dec 2008)

- add and implement *start* method to components in xlutils.filter.
- fixed bug when using set_rdsheet with ColumnTrimmer.
- improved installation documentation.
- renamed xlutils.styles.CellStyle to more appropriate xlutils.styles.NamedStyle.
- improved documentation for xlutils.styles.
- moved to using TestFixtures and Mock for tests.
- moved to using ErrorHandler rather than duplicating code.

1.1.1 (20 Nov 2008)

- prevented generation of excessively long sheet names that cause Excel to complain.
- added test that will fail if the filesystem used doesn't support filenames with +'s in them.

1.1.0 (14 Nov 2008)

- link to the documentation for xlutils.display
- tighten up version requirements for xlrd and xlwt
- use style compression in xlutils.filter.BaseWriter
- prevent generation of bogus sheet names in xlutils.filter.BaseWriter
- xlutils.filter.BaseFilter now keeps track of rdbook, simplifying the implementation of filters.
- add another example for xlutils.filter
- add xlutils.filter.XLRDReader
- add xlutils.filter.StreamWriter
- add xlutils.styles
- add xlutils.save

1.0.0 (8 Nov 2008)

- initial public release

License

CHAPTER 16

Indices and tables

- genindex
- modindex
- search

Python Module Index

x

Index

V

W

X