Working with Excel files in Python

Chris Withers with help from John Machin

EuroPython 2009, Birmingham

Simplistix

The Tutorial Materials

These can be obtained by CD, USB drive or downloaded from here:

- http://www.simplistix.co.uk/presentations/europython2009excel.zip

The Website

The best place to start when working with Excel files in python is the website:

- http://www.python-excel.org

Introduction

This tutorial covers the following libraries:

xlrd

- http://pypi.python.org/pypi/xlrd
- reading data and formatting from .xls files
- this tutorial covers version 0.7.1
- API documentation can be found at:
  - https://secure.simplistix.co.uk/svn/xlrd/trunk/xlrd/doc/xlrd.html

xlwt

- http://pypi.python.org/pypi/xlwt
- writing data and formatting to .xls files
- this tutorial covers version 0.7.2
- Incomplete API documentation can be found at:
  - https://secure.simplistix.co.uk/svn/xlwt/trunk/xlwt/doc/xlwt.html
- Fairly complete examples can be found at
  - https://secure.simplistix.co.uk/svn/xlwt/trunk/xlwt/examples/

xlutils

- http://pypi.python.org/pypi/xlutils
- a collection of utilities using both xlrd and xlwt:
  - copying data from a source to a target spreadsheet
  - filtering data from a source to a target spreadsheet
- this tutorial covers version 1.3.0 and above.
- Documentation and examples can be found at:
  - https://secure.simplistix.co.uk/svn/xlutils/trunk/xlutils/docs/

There are still reasons why automating an Excel instance via COM is necessary:

- manipulation of graphs
- rich text cells
- reading formulae in cells
- working with macros and names
- the more esoteric things found in .xls files

Installation

There are several methods of installation available. While the following examples are for `xlrd`, the exact same steps can be used for any of the three libraries.

Install from Source

On Linux:

```
$ tar xzf xlrd.tgz
$ cd xlrd-0.7.1
$ python setup.py install
```

NB: Make sure you use the python you intend to use for your project.

On Windows, having used WinZip or similar to unpack xlrd-0.7.0.zip:

```
C:\> cd xlrd-0.7.1
C:\xlrd-0.7.1> \Python26\python setup.py install
```

NB: Make sure you use the python you intend to use for your project.

Install using Windows Installer

On Windows, you can download and run the xlrd-0.7.1.win32.exe installer.

Beware that this will only install to Python installations that are in the windows registry.

Install using EasyInstall

This cross-platform method requires that you already have EasyInstall installed. For more information on this, please see:

- http://peak.telecommunity.com/DevCenter/EasyInstall

```
easy_install xlrd
```

Installation using Buildout

Buildout provides a cross-platform method of meeting the python package dependencies of a project without interfering with the system python.

Having created a directory called `mybuildout`, download the following file into it:

- http://svn.zope.org/*checkout*/zc.buildout/trunk/bootstrap/bootstrap.py

Now, create a file in `mybuildout` called `buildout.cfg` containing the following:

```
[buildout]
parts = py
versions = versions

[versions]
xlrd=0.7.1
xlwt=0.7.2
xlutils=1.3.2

[py]
recipe = zc.recipe.egg
eggs =
  xlrd
  xlwt
  xlutils
interpreter = py
```

buildout.cfg

NB: The versions section is optional

Finally, run the following:

```
$ python bootstrap.py
$ bin/buildout
```

These lines:

- initialise the buildout environment
- run the buildout. This should be done each time dependencies change.

Now you can do the following:

```
$ bin/py your_xlrd_xlwt_xltuils_script.py
```

Buildout lives at http://pypi.python.org/pypi/zc.buildout

Reading Excel Files

All the examples shown below can be found in the `xlrd` directory of the course material.

Opening Workbooks

Workbooks can be loaded either from a file, an `mmap.mmap` object or from a string:

```
from mmap import mmap,ACCESS_READ
from xlrd import open_workbook

print open_workbook('simple.xls')

with open('simple.xls', 'rb') as f:
    print open_workbook(
        file_contents=mmap(f.fileno(),0,access=ACCESS_READ)
        )

aString = open('simple.xls','rb').read()
print open_workbook(file_contents=aString)
```

open.py

Navigating a Workbook

Here is a simple example of workbook navigation:

```
from xlrd import open_workbook

wb = open_workbook('simple.xls')

for s in wb.sheets():
    print 'Sheet:',s.name
    for row in range(s.nrows):
        values = []
        for col in range(s.ncols):
            values.append(s.cell(row,col).value)
        print ','.join(values)
    print
```

simple.py

The next few sections will cover the navigation of workbooks in more detail.

Introspecting a Book

The `xlrd.Book` object returned by `open_workbook` contains all information to do with the workbook and can be used to retrieve individual sheets within the workbook.

The `nsheets` attribute is an integer containing the number of sheets in the workbook. This attribute, in combination with the `sheet_by_index` method is the most common way of retrieving individual sheets.

The `sheet_names` method returns a list of unicodes containing the names of all sheets in the workbook. Individual sheets can be retrieved using these names by way of the `sheet_by_name` function.

The results of the `sheets` method can be iterated over to retrieve each of the sheets in the workbook.

The following example demonstrates these methods and attributes:

```
from xlrd import open_workbook

book = open_workbook('simple.xls')

print book.nsheets

for sheet_index in range(book.nsheets):
    print book.sheet_by_index(sheet_index)

print book.sheet_names()
for sheet_name in book.sheet_names():
    print book.sheet_by_name(sheet_name)

for sheet in book.sheets():
    print sheet
```

introspect_book.py

`xlrd.Book` objects have other attributes relating to the content of the workbook that are only rarely useful:

- `codepage`
- `countries`
- `user_name`

If you think you may need to use these attributes, please see the `xlrd` documentation.

Introspecting a Sheet

The `xlrd.sheet.Sheet` objects returned by any of the methods described above contain all the information to do with a worksheet and its contents.

The `name` attribute is a unicode representing the name of the worksheet.

The `nrows` and `ncols` attributes contain the number of rows and the number of columns, respectively, in the worksheet.

The following example shows how these can be used to iterate over and display the contents of one worksheet:

```
from xlrd import open_workbook,cellname

book = open_workbook('odd.xls')
sheet = book.sheet_by_index(0)

print sheet.name

print sheet.nrows
print sheet.ncols

for row_index in range(sheet.nrows):
    for col_index in range(sheet.ncols):
        print cellname(row_index,col_index),'-',
        print sheet.cell(row_index,col_index).value
```

introspect_sheet.py

`xlrd.sheet.Sheet` objects have other attributes relating to the content of the worksheet that are only rarely useful:

- col_label_ranges
- row_label_ranges
- visibility

If you think you may need to use these attributes, please see the `xlrd` documentation.

Getting a particular Cell

As already seen in previous examples, the `cell` method of a `Sheet` object can be used to return the contents of a particular cell.

The `cell` method returns an `xlrd.sheet.Cell` object. These objects have very few attributes, of which `value` contains the actual value of the cell and `ctype` contains the type of the cell.

In addition, `Sheet` objects have two methods for returning these two types of data. The `cell_value` method returns the value for a particular cell, while the `cell_type` method returns the type of a particular cell. These methods can be quicker to execute than retrieving the `Cell` object.

Cell types are covered in more detail later. The following example shows the methods, attributes and classes in action:

```
from xlrd import open_workbook,XL_CELL_TEXT

book = open_workbook('odd.xls')
sheet = book.sheet_by_index(1)

cell = sheet.cell(0,0)
print cell
print cell.value
print cell.ctype==XL_CELL_TEXT

for i in range(sheet.ncols):
    print sheet.cell_type(1,i),sheet.cell_value(1,i)
```

cell_access.py

Iterating over the contents of a Sheet

We've already seen how to iterate over the contents of a worksheet and retrieve the resulting individual cells. However, there are methods to retrieve groups of cells more easily. There are a symmetrical set of methods that retrieve groups of cell information either by row or by column.

The `row` method and `col` method return all the `Cell` objects for a whole row or column respectively.

The `row_slice` and `col_slice` methods return a list of `Cell` objects in a row or column, respectively, bounded by and start index and an optional end index.

The `row_types` and `col_types` methods return a list of integers representing the cell types in a row or column, respectively, bounded by and start index and an optional end index.

The `row_values` and `col_values` methods return a list of objects representing the cell values in a row or column, respectively, bounded by and start index and an optional end index.

The following examples demonstrates all of the sheet iteration methods:

```
from xlrd import open_workbook

book = open_workbook('odd.xls')
sheet0 = book.sheet_by_index(0)
sheet1 = book.sheet_by_index(1)

print sheet0.row(0)
print sheet0.col(0)
print
print sheet0.row_slice(0,1)
print sheet0.row_slice(0,1,2)
print sheet0.row_values(0,1)
print sheet0.row_values(0,1,2)
print sheet0.row_types(0,1)
print sheet0.row_types(0,1,2)
print
print sheet1.col_slice(0,1)
print sheet0.col_slice(0,1,2)
print sheet1.col_values(0,1)
print sheet0.col_values(0,1,2)
print sheet1.col_types(0,1)
print sheet0.col_types(0,1,2)
```

sheet_iteration.py

Utility Functions

When navigating around a workbook, it's often useful to be able to convert between row and column indexes and the Excel cell references that users may be used to seeing. The following functions are provided to help with this:

The `cellname` function turns a row and column index into a relative Excel cell reference.

The `cellnameabs` function turns a row and column index into an absolute Excel cell reference.

The `colname` function turns a column index into an Excel column name.

These three functions are demonstrated in the following example:

```
from xlrd import cellname, cellnameabs, colname

print cellname(0,0),cellname(10,10),cellname(100,100)
print cellnameabs(3,1),cellnameabs(41,59),cellnameabs(265,358)
print colname(0),colname(10),colname(100)
```

utility.py

Unicode

All text attributes and values produced by `xlrd` will be either unicode objects or, in rare cases, ascii encoded strings.

Each piece of text in an Excel file written by Microsoft Excel is encoded into one of the following:

- Latin1, if it fits
- UTF_16_LE, if it doesn't find into Latin1
- In older files, by an encoding specified by an MS codepage. These are mapped to Python encodings by `xlrd` and still results in unicode objects.

In rare cases, other software has been know to write no codepage or the wrong codepage into Excel files. In this case, the correct encoding may need to be specified to open_workbook:

```
from xlrd import open_workbook
book = open_workbook('dodgy.xls',encoding='cp1252')
```

Types of Cell

We have already seen the cell type expressed as an integer. This integer corresponds to a set of constants in xlrd that identify the type of the cell. The full set of possible cell types is listed in the following sections.

Text

These are represented by the `xlrd.XL_CELL_TEXT` constant.

Cells of this type will have values that are `unicode` objects.

Number

These are represented by the `xlrd.XL_CELL_NUMBER` constant.

Cells of this type will have values that are `float` objects.

Date

These are represented by the `xlrd.XL_CELL_DATE` constant.

NB: Dates don't really exist in Excel files, they are merely Numbers with a particular number formatting.

`xlrd` will return `xlrd.XL_CELL_DATE` as the cell type if the number format string looks like a date.

The `xldate_as_tuple` method is provided for turning the `float` in a Date cell into a tuple suitable for instantiating various date/time objects. This example shows how to use it:

```
from datetime import date,datetime,time
from xlrd import open_workbook,xldate_as_tuple

book = open_workbook('types.xls')
sheet = book.sheet_by_index(0)

date_value =
xldate_as_tuple(sheet.cell(3,2).value,book.datemode)
print datetime(*date_value),date(*date_value[:3])
datetime_value =
xldate_as_tuple(sheet.cell(3,3).value,book.datemode)
print datetime(*datetime_value)
time_value =
xldate_as_tuple(sheet.cell(3,4).value,book.datemode)
print time(*time_value[3:])
print datetime(*time_value)
```

dates.py

Caveats:

- Excel files have two possible date modes, one for files originally created on Windows and one for files originally created on an Apple machine. This is expressed as the `datemode` attribute of `xlrd.Book` objects and **must** be passed to xldate_as_tuple.
- The Excel file format has various problems with dates before 3 Jan 1904 that can cause date ambiguities that can result in `xldate_as_tuple` raising an XLDateError.
- The Excel formula function `DATE()` can return unexpected dates in certain circumstances.

Boolean

These are represented by the `xlrd.XL_CELL_BOOLEAN` constant.

Cells of this type will have values that are `bool` objects.

Error

These are represented by the `xlrd.XL_CELL_ERROR` constant.

Cells of this type will have values that are integers representing specific error codes.

The `error_text_from_code` function can be used to turn error codes into error messages:

```
from xlrd import open_workbook,error_text_from_code

book = open_workbook('types.xls')
sheet = book.sheet_by_index(0)

print error_text_from_code[sheet.cell(5,2).value]
print error_text_from_code[sheet.cell(5,3).value]
```

errors.py

For a simpler way of sensibly displaying all cell types, see `xlutils.display`.

Empty / Blank

Excel files only store cells that either have information in them or have formatting applied to them. However, `xlrd` presents sheets as rectangular grids of cells.

Cells where no information is present in the Excel file are represented by the `xlrd.XL_CELL_EMPTY` constant. In addition, there is only one “empty cell”, whose value is an empty string, used by `xlrd`, so empty cells may be checked using a Python identity check.

Cells where only formatting information is present in the Excel file are represented by the `xlrd.XL_CELL_BLANK` constant and their value will always be an empty string.

```
from xlrd import open_workbook,empty_cell

print empty_cell.value

book = open_workbook('types.xls')
sheet = book.sheet_by_index(0)
empty = sheet.cell(6,2)
blank = sheet.cell(7,2)
print empty is blank, empty is empty_cell, blank is empty_cell

book = open_workbook('types.xls',formatting_info=True)
sheet = book.sheet_by_index(0)
empty = sheet.cell(6,2)
blank = sheet.cell(7,2)
print empty.ctype,repr(empty.value)
print blank.ctype,repr(blank.value)
```

emptyblank.py

The following example brings all of the above cell types together and shows examples of their use:

```
from xlrd import open_workbook

def cell_contents(sheet,row_x):
    result = []
    for col_x in range(2,sheet.ncols):
        cell = sheet.cell(row_x,col_x)
        result.append((cell.ctype,cell,cell.value))
    return result

sheet = open_workbook('types.xls').sheet_by_index(0)

print 'XL_CELL_TEXT',cell_contents(sheet,1)
print 'XL_CELL_NUMBER',cell_contents(sheet,2)
print 'XL_CELL_DATE',cell_contents(sheet,3)
print 'XL_CELL_BOOLEAN',cell_contents(sheet,4)
print 'XL_CELL_ERROR',cell_contents(sheet,5)
print 'XL_CELL_BLANK',cell_contents(sheet,6)
print 'XL_CELL_EMPTY',cell_contents(sheet,7)

print
sheet = open_workbook(
            'types.xls',formatting_info=True
            ).sheet_by_index(0)

print 'XL_CELL_TEXT',cell_contents(sheet,1)
print 'XL_CELL_NUMBER',cell_contents(sheet,2)
print 'XL_CELL_DATE',cell_contents(sheet,3)
print 'XL_CELL_BOOLEAN',cell_contents(sheet,4)
print 'XL_CELL_ERROR',cell_contents(sheet,5)
print 'XL_CELL_BLANK',cell_contents(sheet,6)
print 'XL_CELL_EMPTY',cell_contents(sheet,7)
```

cell_types.py

Names

These are an infrequently used but powerful way of abstracting commonly used information found within Excel files.

They have many uses, and `xlrd` can extract information from many of them. A notable exception are names that refer to sheet and VBA macros, which are extracted but should be ignored.

Names are created in Excel by navigating to `Insert > Name > Define`. If you plan to use `xlrd` to extract information from Names, familiarity with the definition and use of names in your chosen spreadsheet application is a good idea.

Types

A Name can refer to:

- A constant
  - `CurrentInterestRate = 0.015`
  - `NameOfPHB = "Attila T. Hun"`
- An absolute (i.e. not relative) cell reference
  - `CurrentInterestRate = Sheet1!$B$4`
- Absolute reference to a 1D, 2D, or 3D block of cells
  - `MonthlySalesByRegion = Sheet2:Sheet5!$A$2:$M$100`
- A list of absolute references
  - `Print_Titles = [row_header_ref, col_header_ref])`

Constants can be extracted.

The coordinates of an absolute reference can be extracted so that you can then extract the corresponding data from the relevant sheet(s).

A relative reference is useful only if you have external knowledge of what cells can be used as the origin. Many formulas found in Excel files include function calls and multiple references and are not useful, and can be too hard to evaluate.

A full calculation engine is not included in xlrd.

Scope

The scope of a Name can be global, or it may be specific to a particular sheet. A Name's identifier may be re-used in different scopes. When there are multiple Names with the same identifier, the most appropriate one is used based on scope. A good example of this is the built-in name Print_Area; each worksheet may have one of these.

Examples:

`name=rate, scope=Sheet1, formula=0.015`

`name=rate, scope=Sheet2, formula=0.023`

`name=rate, scope=`*`global`*`, formula=0.040`

A cell formula `(1+rate)^20` is equivalent to `1.015^20` if it appears in `Sheet1` but equivalent to `1.023^20` if it appears in `Sheet2`, and `1.040^20` if it appears in any other sheet.

Usage

Common reasons for using names include:

- Assigning textual names to values that may occur in many places within a workbook
  - eg: `RATE = 0.015`
- Assigning textual names to complex formulae that may be easily mis-copied
  - eg: `SALES_RESULTS = $A$10:$M$999`

Here's an example real-world use case: reporting to head office. A company's head office makes up a template workbook. Each department gets a copy to fill in. The various ranges of data to be provided all have defined names. When the files come back, a script is used to validate that the department hasn't trashed the workbook and the names are used to extract the data for further processing. Using names decouples any artistic repositioning of the ranges, by either head office template-designing user or by departmental users who are filling in the template, from the script which only has to know what the names of the ranges are.

In the examples directory of the `xlrd` distribution you will find `namesdemo.xls` which has examples of most of the non-macro varieties of defined names. There is also `xlrdnamesAPIdemo.py` which shows how to use the name lookup dictionaries, and how to extract constants and references and the data that references point to.

Formatting

We've already seen that `open_workbook` has a parameter to load formatting information from Excel files. When this is done, all the formatting information is available, but the details of how it is presented are beyond the scope of this tutorial.

If you wish to copy existing formatted data to a new Excel file, see `xlutils.copy` and `xlutils.filter`.

If you do wish to inspect formatting information, you'll need to consult the following attributes of the following classes:

xlrd.Book

`colour_map`	`palette_record`
`font_list`	`style_name_map`
`format_list`	`xf_list`
`format_map`	

xlrd.sheet.Sheet

`cell_xf_index`	`default_row_height_mismatch`
`rowinfo_map`	`default_row_hidden`
`colinfo_map`	`defcolwidth`
`computed_column_width`	`gcw`
`default_additional_space_above`	`merged_cells`
`default_additional_space_below`	`standard_width`
`default_row_height`	

xlrd.sheet.Cell

`xf_index`

Other Classes

In addition, the following classes are solely used to represent formatting information:

`xlrd.sheet.Rowinfo`	`xlrd.formatting.XFAlignment`
`xlrd.sheet.Colinfo`	`xlrd.formatting.XFBackground`
`xlrd.formatting.Font`	`xlrd.formatting.XFBorder`
`xlrd.formatting.Format`	`xlrd.formatting.XFProtection`
`xlrd.formatting.XF`	

Working with large Excel files

If you're working with particularly large Excel files then there are two features of xlrd that you should be aware of:

- The on_demand parameter can be passed as True to open_workbook resulting in worksheets only being loaded into memory when they are requested.
- xlrd.Book objects have an unload_sheet method that will unload worksheet, specified by either sheet index or sheet name, from memory.

The following example shows how a large workbook could be iterated over when only sheets matching a certain pattern need to be inspected, and where only one of those sheets ends up in memory at any one time:

```
from xlrd import open_workbook

book = open_workbook('simple.xls',on_demand=True)

for name in book.sheet_names():
    if name.endswith('2'):
        sheet = book.sheet_by_name(name)
        print sheet.cell_value(0,0)
        book.unload_sheet(name)
```

large_files.py

Introspecting Excel files with runxlrd.py

The xlrd source distribution includes a `runxlrd.py` script that is extremely useful for introspecting Excel files without writing a single line of Python.

You are encouraged to run a variety of the commands it provides over the Excel files provided in the course materials.

The following gives an overview of what's available from `runxlrd`, and can be obtained using `python runxlrd.py –-help`:

```
runxlrd.py [options] command [input-file-patterns]

Commands:

2rows           Print the contents of first and last row in each sheet
3rows           Print the contents of first, second and last row in each sheet
bench           Same as "show", but doesn't print -- for profiling
biff_count[1]   Print a count of each type of BIFF record in the file
biff_dump[1]    Print a dump (char and hex) of the BIFF records in the file
fonts           hdr + print a dump of all font objects
hdr             Mini-overview of file (no per-sheet information)
hotshot         Do a hotshot profile run e.g. ... -f1 hotshot bench bigfile*.xls
labels          Dump of sheet.col_label_ranges and ...row... for each sheet
name_dump       Dump of each object in book.name_obj_list
names           Print brief information for each NAME record
ov              Overview of file
profile         Like "hotshot", but uses cProfile
show            Print the contents of all rows in each sheet
version[0]      Print versions of xlrd and Python and exit
xfc             Print "XF counts" and cell-type counts -- see code for details

[0] means no file arg
[1] means only one file arg i.e. no glob.glob pattern


Options:
  -h, --help            show this help message and exit
  -l LOGFILENAME, --logfilename=LOGFILENAME
                        contains error messages
  -v VERBOSITY, --verbosity=VERBOSITY
                        level of information and diagnostics provided
  -p PICKLEABLE, --pickleable=PICKLEABLE
                        1: ensure Book object is pickleable (default); 0:
                        don't bother
  -m MMAP, --mmap=MMAP  1: use mmap; 0: don't use mmap; -1: accept heuristic
  -e ENCODING, --encoding=ENCODING
                        encoding override
  -f FORMATTING, --formatting=FORMATTING
                        0 (default): no fmt info 1: fmt info (all cells)
  -g GC, --gc=GC        0: auto gc enabled; 1: auto gc disabled, manual
                        collect after each file; 2: no gc
  -s ONESHEET, --onesheet=ONESHEET
                        restrict output to this sheet (name or index)
  -u, --unnumbered      omit line numbers or offsets in biff_dump
```

Writing Excel Files

All the examples shown below can be found in the `xlwt` directory of the course material.

Creating elements within a Workbook

Workbooks are created with `xlwt` by instantiating an `xlwt.Workbook` object, manipulating it and then calling its `save` method.

The `save` method may be passed either a string containing the path to write to or a file-like object, opened for writing in binary mode, to which the binary Excel file data will be written.

The following objects can be created within a workbook:

Worksheets

Worksheets are created with the `add_sheet` method of the `Workbook` class.

To retrieve an existing sheet from a `Workbook`, use its `get_sheet` method. This method is particularly useful when the `Workbook` has been instantiated by `xlutils.copy`.

Rows

Rows are created using the `row` method of the `Worksheet` class and contain all of the cells for a given row.

The `row` method is also used to retrieve existing rows from a `Worksheet`.

If a large number of rows have been written to a `Worksheet` and memory usage is becoming a problem, the `flush_row_data` method may be called on the `Worksheet`. Once called, any rows flushed cannot be accessed or modified.

It is recommended that `flush_row_data` is called for every 1000 or so rows of a normal size that are written to an `xlwt.Workbook`. If the rows are huge, that number should be reduced.

Columns

Columns are created using the `col` method of the `Worksheet` class and contain display formatting information for a given column.

The `col` method is also used to retrieve existing columns from a `Worksheet`.

Cells

Cells can be written using either the `write` method of either the `Worksheet` or `Row` class.

A more detailed discussion of different ways of writing cells and the different types of cell that may be written is covered later.

A Simple Example

The following example shows how all of the above methods can be used to build and save a simple workbook:

```
from tempfile import TemporaryFile
from xlwt import Workbook

book = Workbook()
sheet1 = book.add_sheet('Sheet 1')
book.add_sheet('Sheet 2')

sheet1.write(0,0,'A1')
sheet1.write(0,1,'B1')
row1 = sheet1.row(1)
row1.write(0,'A2')
row1.write(1,'B2')
sheet1.col(0).width = 10000

sheet2 = book.get_sheet(1)
sheet2.row(0).write(0,'Sheet 2 A1')
sheet2.row(0).write(1,'Sheet 2 B1')
sheet2.flush_row_data()
sheet2.write(1,0,'Sheet 2 A3')
sheet2.col(0).width = 5000
sheet2.col(0).hidden = True

book.save('simple.xls')
book.save(TemporaryFile())
```

simple.py

Unicode

The best policy is to pass unicode objects to all `xlwt`-related method calls.

If you absolutely have to use encoded strings then make sure that the encoding used is consistent across all calls to any `xlwt`-related methods.

If encoded strings are used and the encoding is not `'ascii'`, then any `Workbook` objects must be created with the appropriate encoding specified:

```
from xlwt import Workbook
book = Workbook(encoding='utf-8')
```

Writing to Cells

A number of different ways of writing a cell are provided by xlwt along with different strategies for handling multiple writes to the same cell.

Different ways of writing cells

There are generally three ways to write to a particular cell:

- Worksheet.write(row_index,column_index,value)
    - This is just syntactic sugar for sheet.row(row_index).write(column_index,value)
    - It can be useful when you only want to write one cell to a row
- Row.write(column_index,value)
    - This will write the correct type of cell based on the value passed
    - Because it figures out what type of cell to write, this method may be slower for writing large workbooks
- Specialist write methods on the Row class
    - Each type of cell has a specialist setter method as covered in the "Types of Cell" section below.
    - These require you to pass the correct type of Python object but can be faster.

In general, use Worksheet.write for convenience and the specialist write methods if you require speed for a large volume of data.

Overwriting Cells

The Excel file format does nothing to prevent multiple records for a particular cell occurring but, if this happens, the results will vary depending on what application is used to open the file. Excel will display a `"File error: data may have been lost"` while OpenOffice.org will show the last record for the cell that occurs in the file.

To help prevent this, `xlwt` provides two modes of operation:

- Writing to the same cell more than once will result in an exception
  This is the default mode.
- Writing to the same cell more than once will replace the record for that cell, and only one record will be written when the Workbook is saved.

The following example demonstrates these two options:

```
from xlwt import Workbook

book = Workbook()
sheet1 = book.add_sheet('Sheet 1',cell_overwrite_ok=True)
sheet1.write(0,0,'original')
sheet = book.get_sheet(0)
sheet.write(0,0,'new')

sheet2 = book.add_sheet('Sheet 2')
sheet2.write(0,0,'original')
sheet2.write(0,0,'new')
```

overwriting.py

The most common case for needing to overwrite cells is when an existing Excel file has been loaded into a Workbook instance using `xlutils.copy`.

Types of Cell

All types of cell supported by the Excel file format can be written:

Text

When passed a `unicode` or string, the `write` methods will write a Text cell.

The `set_cell_text` method of the `Row` class can also be used to write Text cells.

When passed a string, these methods will first decode the string using the Workbook's encoding.

Number

When passed a `float`, `int`, `long`, or `decimal.Decimal`, the `write` methods will write a Number cell.

The `set_cell_number` method of the `Row` class can also be used to write Number cells.

Date

When passed a `datetime.datetime`, `datetime.date` or `datetime.time`, the `write` methods will write a Date cell.

The `set_cell_date` method of the `Row` class can also be used to write Date cells.

Note: As mentioned earlier, a date is not really a separate type in Excel; if you don't apply a date format, it will be treated as a number.

Boolean

When passed a `bool`, the `write` methods will write a Boolean cell.

The `set_cell_boolean` method of the `Row` class can also be used to write Text cells.

Error

You shouldn't ever want to write Error cells!

However, if you absolutely must, the `set_cell_error` method of the Row class can be used to do so. For convenience, it can be called with either hexadecimal error codes, expressed as integers, or the error text that Excel would display.

Blank

It is not normally necessary to write blank cells. The one exception to this is if you wish to apply formatting to a cell that contains nothing.

To do this, either call the `write` methods with an empty string or None, or use the `set_cell_blank` method of the `Row` class.

If you need to do this for more than one cell in a row, using the `set_cell_mulblanks` method will result in a smaller Excel file when the `Workbook` is saved.

The following example brings all of the above cell types together and shows examples use both the generic `write` method and the specialist methods:

```
from datetime import date,time,datetime
from decimal import Decimal
from xlwt import Workbook,Style

wb = Workbook()
ws = wb.add_sheet('Type examples')
ws.row(0).write(0,u'\xa3')
ws.row(0).write(1,'Text')
ws.row(1).write(0,3.1415)
ws.row(1).write(1,15)
ws.row(1).write(2,265L)
ws.row(1).write(3,Decimal('3.65'))
ws.row(2).set_cell_number(0,3.1415)
ws.row(2).set_cell_number(1,15)
ws.row(2).set_cell_number(2,265L)
ws.row(2).set_cell_number(3,Decimal('3.65'))
ws.row(3).write(0,date(2009,3,18))
ws.row(3).write(1,datetime(2009,3,18,17,0,1))
ws.row(3).write(2,time(17,1))
ws.row(4).set_cell_date(0,date(2009,3,18))
ws.row(4).set_cell_date(1,datetime(2009,3,18,17,0,1))
ws.row(4).set_cell_date(2,time(17,1))
ws.row(5).write(0,False)
ws.row(5).write(1,True)
ws.row(6).set_cell_boolean(0,False)
ws.row(6).set_cell_boolean(1,True)
ws.row(7).set_cell_error(0,0x17)
ws.row(7).set_cell_error(1,'#NULL!')
ws.row(8).write(
    0,'',Style.easyxf('pattern: pattern solid, fore_colour
green;'))
ws.row(8).write(
    1,None,Style.easyxf('pattern: pattern solid, fore_colour
blue;'))
ws.row(9).set_cell_blank(
    0,Style.easyxf('pattern: pattern solid, fore_colour
yellow;'))
ws.row(10).set_cell_mulblanks(
    5,10,Style.easyxf('pattern: pattern solid, fore_colour
red;')
    )

wb.save('types.xls')
```

cell_types.py

Styles

Most elements of an Excel file can be formatted. For many elements including cells, rows and columns, this is done by assigning a style, known as an XF record, to that element.

This is done by passing an xlwt.XFStyle instance to the optional last argument to the various write methods and specialist set_cell_ methods. xlwt.Row and xlwt.Column instances have set_style methods to which an xlwt.XFStyle instance can be passed.

XFStyle

In xlwt, the XF record is represented by the XFStyle class and its related attribute classes.

The following example shows how to create a red Date cell with Arial text and a black border:

```
from datetime import date
from xlwt import Workbook, XFStyle, Borders, Pattern, Font

fnt = Font()
fnt.name = 'Arial'

borders = Borders()
borders.left = Borders.THICK
borders.right = Borders.THICK
borders.top = Borders.THICK
borders.bottom = Borders.THICK

pattern = Pattern()
pattern.pattern = Pattern.SOLID_PATTERN
pattern.pattern_fore_colour = 0x0A

style = XFStyle()
style.num_format_str='YYYY-MM-DD'
style.font = fnt
style.borders = borders
style.pattern = pattern

book = Workbook()
sheet = book.add_sheet('A Date')
sheet.write(1,1,date(2009,3,18),style)

book.save('date.xls')
```

xfstyle_format.py

This can be quite cumbersome!

easyxf

Thankfully, `xlwt` provides the `easyxf` helper to create `XFStyle` instances from human readable text and an optional string containing a number format.

Here is the above example, this time created with easyxf:

```
from datetime import date
from xlwt import Workbook, easyxf

book = Workbook()
sheet = book.add_sheet('A Date')

sheet.write(1,1,date(2009,3,18),easyxf(
    'font: name Arial;'
    'borders: left thick, right thick, top thick, bottom
thick;'
    'pattern: pattern solid, fore_colour red;',
    num_format_str='YYYY-MM-DD'
    ))

book.save('date.xls')
```

easyxf_format.py

The human readable text breaks roughly as follows, in pseudo-regular expression syntax:

```
(<element>:(<attribute> <value>,)+;)+
```

This means:

- The text contains a semi-colon delimited list of element definitions.
- Each element contains a comma-delimited list of attribute and value pairs.

The following sections describe each of the types of element by providing a table of their attributes and possible values for those attributes. For explanations of how to express boolean values and colours, please see the “Types of attribute” section.

font

bold	A *boolean* value. The default is `False`.
charset	The character set to use for this font, which can be one of the following: `ansi_latin, sys_default, symbol, apple_roman, ansi_jap_shift_jis, ansi_kor_hangul, ansi_kor_johab, ansi_chinese_gbk, ansi_chinese_big5, ansi_greek, ansi_turkish, ansi_vietnamese, ansi_hebrew, ansi_arabic, ansi_baltic, ansi_cyrillic, ansi_thai, ansi_latin_ii, oem_latin_i` The default is `sys_default`.
colour	A *colour* specifying the colour for the text. The default is the `automatic` colour.
escapement	This can be one of `none`, `superscript` or `subscript`. The default is `none`.
family	This should be a string containing the name of the font family to use. You probably want to use `name` instead of this attribute and leave this to its default value. The default is `None`.
height	The height of the font as expressed by multiplying the point size by 20. The default is 200, which equates to 10pt.
italic	A *boolean* value. The default is `False`.
name	This should be a string containing the name of the font family to use. The default is `Arial`.
outline	A *boolean* value. The default is `False`.
shadow	A *boolean* value. The default is `False`.
struck_out	A *boolean* value. The default is `False`.
underline	A *boolean* value or one of `none`, `single`, `single_acc`, `double` or `double_acc`. The default is `none`.
color_index	A synonym for `colour`
colour_index	A synonym for `colour`
color	A synonym for `colour`

alignment

direction	One of `general`, `lr`, or `rl`. The default is `general`.
horizontal	One of the following: `general, left, center|centre, right, filled, justified, center|centre_across_selection, distributed` The default is `general`.
indent	A indentation amount between 0 and 15. The default is 0.
rotation	An integer rotation in degrees between -90 and +90 or one of `stacked` or `none`. The default is `none`.
shrink_to_fit	A *boolean* value. The default is `False`.
vertical	One of the following: `top, center|centre, bottom, justified, distributed` The default is `bottom`.
wrap	A *boolean* value. The default is `False`.
dire	This is a synonym for `direction`.
horiz	This is a synonym for `horizontal`.
horz	This is a synonym for `horizontal`.
inde	This is a synonym for `indent`.
rota	This is a synonym for `rotation`.
shri	This is a synonym for `shrink_to_fit`.
shrink	This is a synonym for `shrink_to_fit`.
vert	This is a synonym for `vertical`.

borders

left	A type of border line*
right	A type of border line*
top	A type of border line*
bottom	A type of border line*
diag	A type of border line*
left_colour	A *colour*. The default is the `automatic` colour.
right_colour	A *colour*. The default is the `automatic` colour.
top_colour	A *colour*. The default is the `automatic` colour.
bottom_colour	A *colour*. The default is the `automatic` colour.
diag_colour	A *colour*. The default is the `automatic` colour.
need_diag_1	A *boolean* value. The default is `False`.
need_diag_2	A *boolean* value. The default is `False`.
left_color	A synonym for `left_colour`
right_color	A synonym for `right_colour`
top_color	A synonym for `top_colour`
bottom_color	A synonym for `bottom_colour`
diag_color	A synonym for `diag_colour`

*This can be either an integer width between 0 and 13 or one of the following:
`no_line, thin, medium, dashed, dotted, thick, double, hair, medium_dashed, thin_dash_dotted, medium_dash_dotted, thin_dash_dot_dotted, medium_dash_dot_dotted, slanted_medium_dash_dotted`

pattern

back_colour	A *colour*. The default is the `automatic` colour.
fore_colour	A *colour*. The default is the `automatic` colour.
pattern	One of the following: `no_fill, none, solid, solid_fill, solid_pattern, fine_dots, alt_bars, sparse_dots, thick_horz_bands, thick_vert_bands, thick_backward_diag, thick_forward_diag, big_spots, bricks, thin_horz_bands, thin_vert_bands, thin_backward_diag, thin_forward_diag, squares, diamonds` The default is `none`.
fore_color	A synonym for `fore_colour`
back_color	A synonym for `back_colour`
pattern_fore_colour	A synonym for `fore_colour`
pattern_fore_color	A synonym for `fore_colour`
pattern_back_colour	A synonym for `back_colour`
pattern_back_color	A synonym for `back_colour`

protection

The protection features of the Excel file format are only partially implemented in `xlwt`. Avoid them unless you plan on finishing their implementation.

cell_locked	A *boolean* value. The default is `True`.
formula_hidden	A *boolean* value. The default is `False`.

align

A synonym for `alignment`

border

A synonym for `borders`

Types of attribute

Boolean values are either True or False, but easyxf allows great flexibility in how you choose to express those two values:

- `True` can be expressed by `1`, `yes`, `true` or `on`
- `False` can be expressed by `0`, `no`, `false`, or `off`

Colours in Excel files are a confusing mess. The safest bet to do is just pick from the following list of colour names that `easyxf` understands.

The names used are those reported by the Excel 2003 GUI when you are inspecting the **default** colour palette.

Warning: There are many differences between this implicit mapping from colour-names to RGB values and the mapping used in standards such as HTML andCSS.

aqua	dark_red_ega	light_blue	plum
black	dark_teal	light_green	purple_ega
blue	dark_yellow	light_orange	red
blue_gray	gold	light_turquoise	rose
bright_green	gray_ega	light_yellow	sea_green
brown	gray25	lime	silver_ega
coral	gray40	magenta_ega	sky_blue
cyan_ega	gray50	ocean_blue	tan
dark_blue	gray80	olive_ega	teal
dark_blue_ega	green	olive_green	teal_ega
dark_green	ice_blue	orange	turquoise
dark_green_ega	indigo	pale_blue	violet
dark_purple	ivory	periwinkle	white
dark_red	lavender	pink	yellow

NB: `grey` can be used instead of `gray` wherever it occurs above.

Formatting Rows and Columns

It is possible to specify default formatting for rows and columns within a worksheet. This is done using the `set_style` method of the `Row` and `Column` instances, respectively.

The precedence of styles is as follows:

- the style applied to a cell
- the style applied to a row
- the style applied to a column

It is also possible to hide whole rows and columns by using the `hidden` attribute of Row and Column instances.

The width of a `Column` can be controlled by setting its `width` attribute to an integer where 1 is 1/256 of the width of the zero character, using the first font that occurs in the Excel file.

Do not be fooled by the `height` attribute of the `Row` class, it does nothing. Specify a style on the row and set its font height attribute instead.

The following example shows these methods and properties in use along with the style precedence:

```
from xlwt import Workbook, easyxf
from xlwt.Utils import rowcol_to_cell

row = easyxf('pattern: pattern solid, fore_colour blue')
col = easyxf('pattern: pattern solid, fore_colour green')
cell = easyxf('pattern: pattern solid, fore_colour red')

book = Workbook()

sheet = book.add_sheet('Precedence')
for i in range(0,10,2):
    sheet.row(i).set_style(row)
for i in range(0,10,2):
    sheet.col(i).set_style(col)
for i in range(10):
    sheet.write(i,i,None,cell)

sheet = book.add_sheet('Hiding')
for rowx in range(10):
    for colx in range(10):
        sheet.write(rowx,colx,rowcol_to_cell(rowx,colx))
for i in range(0,10,2):
    sheet.row(i).hidden = True
    sheet.col(i).hidden = True

sheet = book.add_sheet('Row height and Column width')
for i in range(10):
    sheet.write(0,i,0)
for i in range(10):
    sheet.row(i).set_style(easyxf('font:height '+str(200*i)))
    sheet.col(i).width = 256*i

book.save('format_rowscols.xls')
```

format_rowscols.py

Formatting Sheets and Workbooks

There are many possible settings that can be made on Sheets and Workbooks.

Most of them you will never need or want to touch.

If you think you do, see the “Other Properties” section below.

Style compression

While its fine to create as many XFStyle and their associated Font instances as you like, each one written to Workbook will result in an XF record and a Font record. Excel has fixed limits of around 400 Fonts and 4000 XF records so care needs to be taken when generating large Excel files.

To help with this, `xlwt.Workbook` has an optional `style_compression` parameter with the following meaning:

- 0 – no compression. This is the default.
- 1 – compress Fonts only. Not very useful.
- 2 – compress Fonts and XF records.

The following example demonstrates these three options:

```
from xlwt import Workbook, easyxf

style1 = easyxf('font: name Times New Roman')
style2 = easyxf('font: name Times New Roman')
style3 = easyxf('font: name Times New Roman')

def write_cells(book):
    sheet = book.add_sheet('Content')
    sheet.write(0,0,'A1',style1)
    sheet.write(0,1,'B1',style2)
    sheet.write(0,2,'C1',style3)

book = Workbook()
write_cells(book)
book.save('3xf3fonts.xls')

book = Workbook(style_compression=1)
write_cells(book)
book.save('3xf1font.xls')

book = Workbook(style_compression=2)
write_cells(book)
book.save('1xf1font.xls')
```

stylecompression.py

Be aware that doing this compression involves deeply nested comparison of the XFStyle objects, so may slow down writing of large files where many styles are used.

The recommended best practice is to create all the styles you will need in advance and leave `style_compression` at its default value.

Formulae

Formulae can be written by `xlwt` by passing an `xlwt.Formula` instance to either of the write methods or by using the `set_cell_formula` method of `Row` instances, bugs allowing.

The following are supported:

- all the built-in Excel formula functions
- references to other sheets in the same workbook
- access to all the add-in functions in the Analysis Toolpak (ATP)

- comma or semicolon as the argument separator in function calls
- case-insensitive matching of formula names

The following are not suppoted:

- references to external workbooks
- array aka Ctrl-Shift-Enter aka CSE formulas
- references to defined Names
- using formulas for data validation or conditional formatting
- evaluation of formulae

The following example shows some of these things in action:

```
from xlwt import Workbook, Formula

book = Workbook()

sheet1 = book.add_sheet('Sheet 1')
sheet1.write(0,0,10)
sheet1.write(0,1,20)
sheet1.write(1,0,Formula('A1/B1'))

sheet2 = book.add_sheet('Sheet 2')
row = sheet2.row(0)
row.write(0,Formula('sum(1,2,3)'))
row.write(1,Formula('SuM(1;2;3)'))
row.write(2,Formula("$A$1+$B$1*SUM('ShEEt 1'!$A$1:$b$2)"))

book.save('formula.xls')
```

formulae.py

Names

Names cannot currently be written by `xlwt`.

Utility methods

The Utils module of xlwt contains several useful utility functions:

col_by_name

This will convert a string containing a column identifier into an integer column index.

cell_to_rowcol

This will convert a string containing an excel cell reference into a four-element tuple containing:

`(row,col,row_abs,col_abs)`

`row` – integer row index of the referenced cell

`col` – integer column index of the referenced cell

`row_abs` – boolean indicating whether the row index is absolute (True) or relative (False)

`col_abs` – boolean indicating whether the column index is absolute (True) or relative (False)

cell_to_rowcol2

This will convert a string containing an excel cell reference into a two-element tuple containing:

`(row,col)`

`row` – integer row index of the referenced cell

`col` – integer column index of the referenced cell

rowcol_to_cell

This will covert an integer row and column index into a string excel cell reference, with either index optionally being absolute.

cellrange_to_rowcol_pair

This will convert a string containing an excel range into a four-element tuple containing:

`(row1,col1,row2,col2)`

`row1` – integer row index of the start of the range

`col1` – integer column index of the start of the range

`row2` – integer row index of the end of the range

`col2` – integer column index of the end of the range

rowcol_pair_to_cellrange

This will covert a pair of integer row and column indexes into a string containing an excel cell range. Any of the indexes specified can optionally be made to be absolute.

valid_sheet_name

This function takes a single string argument and returns a boolean value indication whether the sheet name will work without problems (True) or will cause complaints from Excel (False).

The following example shows all of these functions in use:

```
from xlwt import Utils

print 'AA ->',Utils.col_by_name('AA')
print 'A ->',Utils.col_by_name('A')

print 'A1 ->',Utils.cell_to_rowcol('A1')
print '$A$1 ->',Utils.cell_to_rowcol('$A$1')

print 'A1 ->',Utils.cell_to_rowcol2('A1')

print (0,0),'->',Utils.rowcol_to_cell(0,0)
print (0,0,False,True),'->',
print Utils.rowcol_to_cell(0,0,False,True)
print (0,0,True,True),'->',
print Utils.rowcol_to_cell(
          row=0,col=0,row_abs=True,col_abs=True
          )

print '1:3 ->',Utils.cellrange_to_rowcol_pair('1:3')
print 'B:G ->',Utils.cellrange_to_rowcol_pair('B:G')
print 'A2:B7 ->',Utils.cellrange_to_rowcol_pair('A2:B7')
print 'A1 ->',Utils.cellrange_to_rowcol_pair('A1')

print (0,0,100,100),'->',
print Utils.rowcol_pair_to_cellrange(0,0,100,100)
print (0,0,100,100,True,False,False,False),'->',
print Utils.rowcol_pair_to_cellrange(
          row1=0,col1=0,row2=100,col2=100,
          row1_abs=True,col1_abs=False,
          row2_abs=False,col2_abs=True
          )

for name in (
    '','"'quoted'"',"O'hare","X"*32,"[]:\\?/*\x00"
    ):
    print 'Is %r a valid sheet name?' % name,
    if Utils.valid_sheet_name(name):
        print "Yes"
    else:
        print "No"
```

utilities.py

Other properties

There are many other properties that you can set on xlwt-related objects. They are all listed below, for each of the types of object. The names are mostly intuitive but you are warned to experiment thoroughly before attempting to use any of these in an important situation as some properties exist that aren't saved to the resulting Excel files and some others are only partially implemented.

xlwt.Workbook

`owner`	`vpos`	`hscroll_visible`
`country_code`	`width`	`vscroll_visible`
`wnd_protect`	`height`	`tabs_visible`
`obj_protect`	`active_sheet`	`dates_1904`
`protect`	`tab_width`	`use_cell_values`
`backup_on_save`	`wnd_visible`	
`hpos`	`wnd_mini`	

xlwt.Row

`set_style`	`height_mismatch`	`hidden`
`height`	`level`	`space_above`
`has_default_height`	`collapse`	`space_below`

xlwt.Column

`set_style`	`width`	`level`
`width_in_pixels`	`hidden`	`collapse`

xlwt.Worksheet

```
name
visibility
row_default_height_mismatch
row_default_hidden
row_default_space_above
row_default_space_below
show_formulas
show_grid
show_headers
show_zero_values
auto_colour_grid
cols_right_to_left
show_outline
remove_splits
selected
sheet_visible
page_preview
first_visible_row
first_visible_col
grid_colour
dialog_sheet
auto_style_outline
outline_below
outline_right
fit_num_pages
show_row_outline
show_col_outline
alt_expr_eval
alt_formula_entries
row_default_height
col_default_height
calc_mode
calc_count
RC_ref_mode
iterations_on
delta
save_recalc
print_headers
print_grid
header_str
footer_str
print_centered_vert
print_centered_horz
left_margin
right_margin
top_margin
bottom_margin
paper_size_code
print_scaling
start_page_number
fit_width_to_pages
fit_height_to_pages
print_in_rows
portrait
print_colour
print_draft
print_notes
print_notes_at_end
print_omit_errors
print_hres
header_margin
footer_margin
copies_num
wnd_protect
obj_protect
protect
scen_protect
password
```

Some examples of Other Properties

The following sections contain examples of how to use some of the properties listed above.

Hyperlinks

Hyperlinks are a type of formula as shown in the following example:

```
from xlwt import Workbook,easyxf,Formula

style = easyxf('font: underline single')

book = Workbook()
sheet = book.add_sheet('Hyperlinks')

sheet.write(
    0, 0,
    Formula('HYPERLINK("http://www.python.org";"Python")'),
    style)

link = 'HYPERLINK("mailto:python-
excel@googlegroups.com";"help")'
sheet.write(
    1,0,
    Formula(link),
    style)

book.save("hyperlinks.xls")
```

hyperlinks.py

Images

Images can be inserted using the `insert_bitmap` method of the `Sheet` class:

```
from xlwt import Workbook
w = Workbook()
ws = w.add_sheet('Image')
ws.insert_bitmap('python.bmp', 0, 0)
w.save('images.xls')
```

images.py

NB: Images are not displayed by OpenOffice.org

Merged cells

Merged groups of cells can be inserted using the `write_merge` method of the `Sheet` class:

```
from xlwt import Workbook,easyxf
style = easyxf(
    'pattern: pattern solid, fore_colour red;'
    'align: vertical center, horizontal center;'
    )
w = Workbook()
ws = w.add_sheet('Merged')
ws.write_merge(1,5,1,5,'Merged',style)
w.save('merged.xls')
```

merged.py

Borders

Writing a single cell with borders is simple enough, however applying a border to a group of cells is painful as shown in this example:

```
from xlwt import Workbook,easyxf
tl = easyxf('border: left thick, top thick')
t = easyxf('border: top thick')
tr = easyxf('border: right thick, top thick')
r = easyxf('border: right thick')
br = easyxf('border: right thick, bottom thick')
b = easyxf('border: bottom thick')
bl = easyxf('border: left thick, bottom thick')
l = easyxf('border: left thick')

w = Workbook()
ws = w.add_sheet('Border')
ws.write(1,1,style=tl)
ws.write(1,2,style=t)
ws.write(1,3,style=tr)
ws.write(2,3,style=r)
ws.write(3,3,style=br)
ws.write(3,2,style=b)
ws.write(3,1,style=bl)
ws.write(2,1,style=l)

w.save('borders.xls')
```

borders.py

NB: Extra care needs to be taken if you're updating an existing Excel file!

Split and Freeze panes

It is fairly straight forward to create frozen panes using `xlwt`.

The location of the split is specified using the integer `vert_split_pos` and `horz_split_pos` properties of the `Sheet` class.

The first visible cells are specified using the integer `vert_split_first_visible` and `horz_split_first_visible` properties of the Sheet class.

The following example shows them all in action:

```
from xlwt import Workbook
from xlwt.Utils import rowcol_to_cell

w = Workbook()
sheet = w.add_sheet('Freeze')
sheet.panes_frozen = True
sheet.remove_splits = True
sheet.vert_split_pos = 2
sheet.horz_split_pos = 10
sheet.vert_split_first_visible = 5
sheet.horz_split_first_visible = 40

for col in range(20):
    for row in range(80):
        sheet.write(row,col,rowcol_to_cell(row,col))

w.save('panes.xls')
```

panes.py

Split panes are a less frequently used feature and their support is less complete in `xlwt`.

The procedure for creating split panes is exactly the same as for frozen panes except that the `panes_frozen` attribute of the Worksheet should be set to `False` instead of `True`.

However, if you really need split panes, you're advised to see professional help before proceeding!

Outlines

These are a little known and little used feature of the Excel file format that can be very useful when dealing with categorised data.

Their use is best shown by example:

```
from xlwt import Workbook
data = [
    ['','','2008','','2009'],
    ['','','Jan','Feb','Jan','Feb'],
    ['Company X'],
    ['','Division A'],
    ['','',100,200,300,400],
    ['','Division B'],
    ['','',100,99,98,50],
    ['Company Y'],
    ['','Division A'],
    ['','',100,100,100,100],
    ['','Division B'],
    ['','',100,101,102,103],
    ]
w = Workbook()
ws = w.add_sheet('Outlines')
for i,row in enumerate(data):
    for j,cell in enumerate(row):
        ws.write(i,j,cell)
ws.row(2).level = 1
ws.row(3).level = 2
ws.row(4).level = 3
ws.row(5).level = 2
ws.row(6).level = 3
ws.row(7).level = 1
ws.row(8).level = 2
ws.row(9).level = 3
ws.row(10).level = 2
ws.row(11).level = 3
ws.col(2).level = 1
ws.col(3).level = 2
ws.col(4).level = 1
ws.col(5).level = 2
w.save('outlines.xls')
```

outlines.py

Zoom magnification and Page Break Preview

The zoom percentage used when viewing a sheet in normal mode can be controlled by setting the normal_magn attribute of a Sheet instance.

The zoom percentage used when viewing a sheet in page break preview mode can be controlled by setting the preview_magn attribute of a Sheet instance.

A Sheet can also be made to show a page break preview by setting the page_preview attribute of the Sheet instance to True.

Here's an example to show all three in action:

```
from xlwt import Workbook

w = Workbook()

ws = w.add_sheet('Normal')
ws.write(0,0,'Some text')
ws.normal_magn = 75

ws = w.add_sheet('Page Break Preview')
ws.write(0,0,'Some text')
ws.preview_magn = 150
ws.page_preview = True

w.save('zoom.xls')
```

zoom.py

Filtering Excel Files

Any examples shown below can be found in the `xlutils` directory of the course material.

Other utilities in xlutils

The `xlutils` package contains several utilities in addition to those for filtering. The following are often useful:

xlutils.styles

This module contains one class which, when instantiated with an `xlrd.Workbook`, will let you discover the style name and information from a given cell in that workbook as shown in the following example:

```
from xlrd import open_workbook
from xlutils.styles import Styles

book = open_workbook('source.xls',formatting_info=True)
styles = Styles(book)
sheet = book.sheet_by_index(0)

print styles[sheet.cell(1,1)].name
print styles[sheet.cell(1,2)].name

A1_style = styles[sheet.cell(0,0)]
A1_font = book.font_list[A1_style.xf.font_index]
print book.colour_map[A1_font.colour_index]
```

styles.py

NB: For obvious reasons, `open_workbook` must be called with `formatting_info=True` in order to use `xlutils.styles`.

Full documentation and examples can be found in the `styles.txt` file in the docs folder of `xlutils`' source distribution.

xlutils.display

This module contains utility functions for easy and safe display of information returned by xlrd.

`quoted_sheet_name` is called with the `name` attribute of an `xlrd.sheet.Sheet` instance and will return an encoded string containing a quoted version of the sheet's name.

`cell_display` is called with an `xlrd.sheet.Cell` instance and returns an encoded string containing a sensible representation of the cells contents, even for Date and Error cells. If a date cell is to be displayed, `cell_display` **must** be called with the `datemode` attribute of the `xlrd.Book` from which the cell came.

The following examples show both functions in action:

```
from xlrd import open_workbook
from xlutils.display import quoted_sheet_name
from xlutils.display import cell_display

wb = open_workbook('source.xls')

print quoted_sheet_name(wb.sheet_names()[0])
print repr(quoted_sheet_name(u'Price(\xa3)','utf-8'))
print quoted_sheet_name(u'My Sheet')
print quoted_sheet_name(u"John's Sheet")

sheet = wb.sheet_by_index(0)
print cell_display(sheet.cell(1,1))
print cell_display(sheet.cell(1,3),wb.datemode)
```

display.py

Full documentation and examples can be found in the display.txt file in the docs folder of xlutils' source distribution.

xlutils.copy

This module contains one function that will take an `xlrd.Book` and returns an `xlwt.Workbook` populated with the data and formatting found in the `xlrd.Book`.

This is extremely useful for updating an existing spreadsheet as the following example shows:

```
from xlrd import open_workbook
from xlwt import easyxf
from xlutils.copy import copy

rb = open_workbook('source.xls',formatting_info=True)
rs = rb.sheet_by_index(0)
wb = copy(rb)
ws = wb.get_sheet(0)

plain = easyxf('')
for i,cell in enumerate(rs.col(2)):
    if not i:
        continue
    ws.write(i,2,cell.value,plain)

for i,cell in enumerate(rs.col(4)):
    if not i:
        continue
    ws.write(i,4,cell.value-1000)

wb.save('output.xls')
```

copy.py

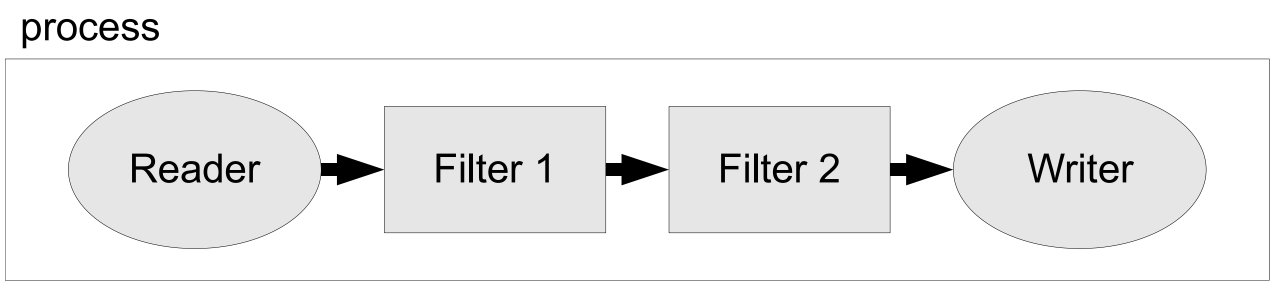
It is important to note that some things won't be copied:

- Formulae
- Names
- anything ignored by xlrd

In addition to the modules described above, there are also `xlutils.margins` and `xlutils.save`, but these are only useful in certain situations. Refer to their documentation in the `xlutils` source distribution.

Structure of xlutils.filter

This framework is designed to filter and split Excel files using a series of modular readers, filters and writers as shown in the diagram below:



The flow of information between the components is by method calls on the next component in the chain. The possible method calls are listed in the table below, where `rdbook` is an `xlrd.Book` instance, `rdsheet` is an `xlrd.sheet.Sheet` instance, `rdrowx`, `rdcolx`, `wtrowx` and `wtcolx` and integer indexes specifying the cell to read from and write to, `wtbook_name` is a string specifying the name of the Excel file to write to and `wtsheet_name` is a `unicode` specifying the name of the sheet to write to:

start()	This method is called before processing of a batch of input. It 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 an rdbook.
workbook(rdbook,wtbook_name)	This method is called every time processing of a new workbook starts
sheet(rdsheet,wtsheet_name)	This method is called every time processing of a new sheet in the current workbook starts
set_rdsheet(rdsheet)	This method is called to indicate a change for the source of cells mid-way through writing a sheet.
row(rdrowx,wtrowx)	The row method is called every time processing of a new row in the current sheet starts.
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 component takes place.
finish	This method is called once processing of all workbooks has been completed.

Readers

A reader's job is to obtain one or more `xlrd.Book` objects and iterate over those objects issuing appropriate calls to the next component in the chain. The order of calling is expected to be as follows:

- `start`
  - `workbook`, once for each `xlrd.Book` object obtained
    - `sheet`, once for each sheet found in the current book
    - `set_rdsheet`, whenever the sheet from which cells to be read needs to be changed. This method may not be called between calls to `row` and `cell`, and between multiple calls to `cell`. It may only be called once all `cell` calls for a row have been made.
      - `row`, once for each row in the current sheet
        - `cell`, once for each cell in the row
- `finish`, once all `xlrd.Book` objects have been processed

Also, for method calls made by a reader, the following should be true:

- `wtbook_name` should be the filename of the file the `xlrd.Book` object originated from.
- `wtsheet_name` should be `rdbook.name`
- `wtrowx` should be equal to `rdrowx`
- `rdcolx` should be equal to `wtcolx`

Because of these restrictions, an `xlutils.filter.BaseReader` class is provided that will normally only need to have one of two methods overridden to get any required functionality:

- `get_filepaths` – if implemented, this must return an iterable sequence of paths to excel files that can be opened with python's builtin file.
- `get_workbooks` – if implemented, this must return an sequence of 2-tuples. Each tuple must contain an `xlrd.Book` object followed by a string containing the filename of the file from which the `xlrd.Book` object was loaded.

Filters

Implementing these components is where the bulk of the work will be done by users of the `xlutils.filter` framework. A Filter's responsibilities are to accept method calls from the preceding component in the chain, do any processing necessary and then emit appropriate method calls to the next component in the chain.

There is very little constraint on what order Filters receive and emit method calls other than that the order of method calls emitted must remain consistent with the structure given above. This enables components to be freely interchanged more easily.

Because Filters may only need to implement few of the full set of method calls, an `xlutils.filter.BaseFilter` is provided that does nothing but pass the method calls on to the next component in the chain. The implementation of this filter is useful to see when embarking on Filter implementation:

```
class BaseFilter:

    def start(self):
        self.next.start()

    def workbook(self,rdbook,wtbook_name):
        self.next.workbook(rdbook,wtbook_name)

    def sheet(self,rdsheet,wtsheet_name):
        self.rdsheet = rdsheet
        self.next.sheet(rdsheet,wtsheet_name)

    def set_rdsheet(self,rdsheet):
        self.rdsheet = rdsheet
        self.next.set_rdsheet(rdsheet)

    def row(self,rdrowx,wtrowx):
        self.next.row(rdrowx,wtrowx)

    def cell(self,rdrowx,rdcolx,wtrowx,wtcolx):
        self.next.cell(rdrowx,rdcolx,wtrowx,wtcolx)

    def finish(self):
        self.next.finish()
```

Writers

These components do the grunt work of actually copying the appropriate information from the `rdbook` and serialising it into an Excel file. This is a complicated process and not for the feint of hard to re-implement.

For this reason, an `xlutils.filter.BaseWriter` component is provided that does all of the hard work and has one method that needs to be implemented. That method is `get_stream` and it is called with the filename of the Excel file to be written. Implementations of this method are expected to return a new file-like object that has a `write` and, by default, a `close` method each time they are called.

Subclasses may also override the boolean `close_after_write` attribute, which is `True` by default, to indicate that the file-like objects returned from `get_stream` should not have their `close` method called once serialisation of the Excel file data is complete.

It is important to note that some things won't be copied from the `rdbook` by `BaseWriter`:

- Formulae
- Names
- anything ignored by xlrd

Process

The process function is responsible for taking a series of components as its arguments. The first of these should be a Reader. The last of these should be a Writer. The rest should be the necessary Filters in the order of processing required.

The process method will wire these components together by way of their `next` attributes and then kick the process off by calling the Reader and passing the first Filter in the chain as its argument.

A worked example

Suppose we want to filter an existing Excel file to omit rows that have an X in the first column.

The following example shows possible components to do this and shows how they would be instantiated and called to achieve this:

```
import os
from xlutils.filter import \
    BaseReader,BaseFilter,BaseWriter,process

class Reader(BaseReader):
    def get_filepaths(self):
        return [os.path.abspath('source.xls')]

class Writer(BaseWriter):
    def get_stream(self,filename):
        return file(filename,'wb')

class Filter(BaseFilter):
    pending_row = None
    wtrowxi = 0
    def workbook(self,rdbook,wtbook_name):
        self.next.workbook(rdbook,'filtered-'+wtbook_name)
    def row(self,rdrowx,wtrowx):
        self.pending_row = (rdrowx,wtrowx)
    def cell(self,rdrowx,rdcolx,wtrowx,wtcolx):
        if rdcolx==0:
            value = self.rdsheet.cell(rdrowx,rdcolx).value
            if value.strip().lower()=='x':
                self.ignore_row = True
                self.wtrowxi -= 1
            else:
                self.ignore_row = False
                rdrowx, wtrowx = self.pending_row
                self.next.row(rdrowx,wtrowx+self.wtrowxi)
        elif not self.ignore_row:
            self.next.cell(
                rdrowx,rdcolx,wtrowx+self.wtrowxi,wtcolx-1
                )

process(Reader(),Filter(),Writer())
```

filter.py

In reality, we would not need to implement the Reader and Writer components, as there are already suitable components included.

Existing components

The xlutils.filter framework comes with a wide range of existing components, each of which is briefly described below. For full descriptions and worked examples of all these components, please see `filter.txt` in the `docs` folder of the `xlutils` source distribution.

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 instantiated with.

XLRDReader

This reader can be used at the start of a chain when you already have an `xlrd.Book` object and you'll looking to process it with `xlutils.filter`.

TestReader

This reader is specifically designed for testing filterimplementations with known sets of cells.

DirectoryWriter

If you want files you're processing to end up on disk, then this is probably the writer for you. It stores files in the directory it is instantiated with.

StreamWriter

If you want to write exactly one workbook to a stream, such as a `tempfile.TemporaryFile` or `sys.stdout`, then this is the writer for you.

XLWTWriter

If you want to change cells after the filtering process is complete then this writer can be used to obtain the `xlwt.Workbook` objects that BaseWriter generates.

ColumnTrimmer

This filter will strip columns containing no useful data from the end of sheets. The definition of “no useful data” can be controlled during instantiation of this filter.

ErrorFilter

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

If Boolean or error Cells are encountered, an error message will be logged to the python logging framework will will also usually mean that no methods will be emitted from this component to the next component in the chain.

Finally, `cell` method calls corresponding to Empty cells in `rdsheet` will not be passed on to the next component in the chain.

Calling this component's `start` method will reset it.

Echo

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

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 more information on heapy, please see http://guppy-pe.sourceforge.net/#Heapy

Possible Tasks for Workshop

The following is a list of tasks that can be attempted by any attendee who hasn't brought their own tasks to attempt.

Installation with IronPython

The libraries have been used successfully with IronPython, but this has not been thoroughly tests or documented.

Installation with Jython

The libraries should all work with Jython, but no one has so far attempted to do so.

Inserting a row into a sheet

Starting with an existing Excel file, attempt to create a new Excel file with a row inserted at a given position.

Splitting a Book into its Sheets

Starting with an existing Excel file, create a directory containing one file for each worksheet in the original file.

Reporting errors in a directory full on Excel files

Scan a directory of Excel files and report the location of any error cells.

A progression of this task is to allow the passing of options to indicate what types of error to report.

Removing Rows containing errors

Starting with an existing Excel file, create a filtering process that generates a new Excel file that excludes any rows containing error cells.

A progression of this task is to generate a new Excel file that contains empty cells where there were errors in the original file.

Filtering Excel files to and from a web server

This task is to create components for xlutils.filter that can read from a website and write back to that website.

The task should result in an HTTPReader and an HTTPWriter.

Producing a report from a database

This task is to take a typical database query and dump it into an Excel file such that the heading row is set up nicely with decent alignment in a frozen pane.

As a precursor to this task, you may need to set up a typical database!