



Creating Excel files with Python and XlsxWriter

Release 0.2.4

John McNamara

March 31, 2013

CONTENTS

1	Introduction	3
2	Getting Started with XlsxWriter	5
2.1	Installing XlsxWriter	5
2.2	Running a sample program	6
2.3	Documentation	7
3	Tutorial 1: Create a simple XLSX file	9
4	Tutorial 2: Adding formatting to the XLSX File	13
5	Tutorial 3: Writing different types of data to the XLSX File	17
6	The Workbook Class	23
6.1	Constructor	23
6.2	workbook.add_worksheet()	25
6.3	workbook.add_format()	25
6.4	workbook.close()	26
6.5	workbook.set_properties()	26
6.6	workbook.define_name()	28
6.7	workbook.worksheets()	29
7	The Worksheet Class	31
7.1	worksheet.write()	31
7.2	worksheet.write_string()	34
7.3	worksheet.write_number()	35
7.4	worksheet.write_formula()	36
7.5	worksheet.write_array_formula()	37
7.6	worksheet.write_blank()	38
7.7	worksheet.write_datetime()	39
7.8	worksheet.write_url()	39
7.9	worksheet.write_rich_string()	41
7.10	worksheet.write_row()	43
7.11	worksheet.write_column()	44
7.12	worksheet.set_row()	44

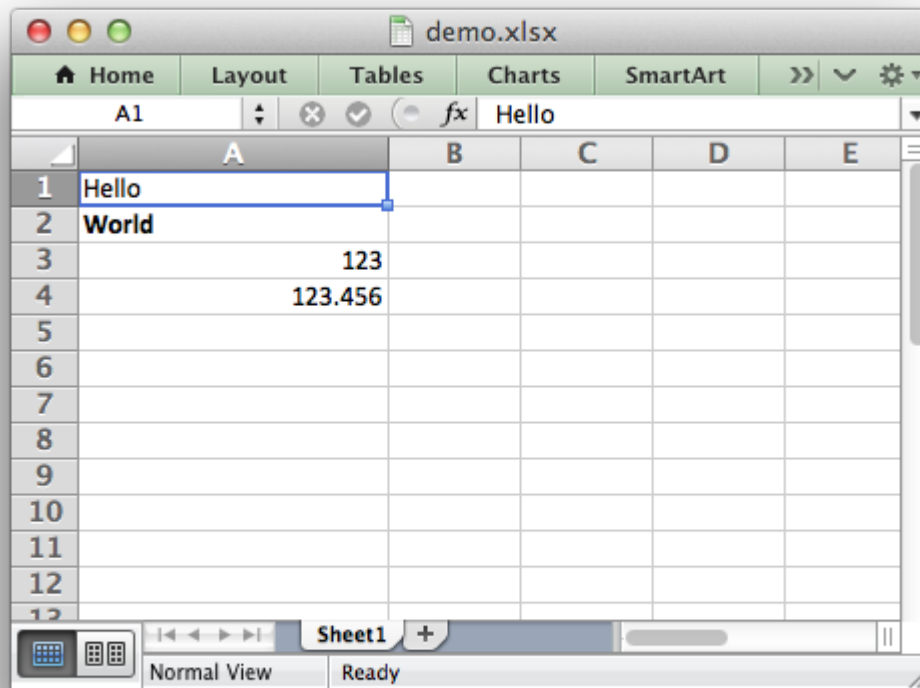
7.13	worksheet.set_column()	46
7.14	worksheet.insert_image()	48
7.15	worksheet.data_validation()	50
7.16	worksheet.conditional_format()	51
7.17	worksheet.write_comment()	53
7.18	worksheet.show_comments()	55
7.19	worksheet.set_comments_author()	55
7.20	worksheet.get_name()	55
7.21	worksheet.activate()	56
7.22	worksheet.select()	56
7.23	worksheet.hide()	57
7.24	worksheet.set_first_sheet()	57
7.25	worksheet.merge_range()	58
7.26	worksheet.autofilter()	59
7.27	worksheet.filter_column()	60
7.28	worksheet.filter_column_list()	61
7.29	worksheet.set_zoom()	61
7.30	worksheet.right_to_left()	62
7.31	worksheet.hide_zero()	62
7.32	worksheet.set_tab_color()	62
7.33	worksheet.protect()	63
8	The Worksheet Class (Page Setup)	65
8.1	worksheet.set_landscape()	65
8.2	worksheet.set_portrait()	65
8.3	worksheet.set_page_view()	65
8.4	worksheet.set_paper()	66
8.5	worksheet.center_horizontally()	67
8.6	worksheet.center_vertically()	67
8.7	worksheet.worksheet.set_margins()	68
8.8	worksheet.set_header()	68
8.9	worksheet.set_footer()	71
8.10	worksheet.repeat_rows()	71
8.11	worksheet.repeat_columns()	72
8.12	worksheet.hide_gridlines()	72
8.13	worksheet.print_row_col_headers()	73
8.14	worksheet.print_area()	73
8.15	worksheet.print_across()	73
8.16	worksheet.fit_to_pages()	74
8.17	worksheet.set_start_page()	75
8.18	worksheet.set_print_scale()	75
8.19	worksheet.set_h_pagebreaks()	75
8.20	worksheet.set_v_pagebreaks()	76
9	The Format Class	77
9.1	format.set_font_name()	78
9.2	format.set_font_size()	78
9.3	format.set_font_color()	78

9.4	format.set_bold()	79
9.5	format.set_italic()	79
9.6	format.set_underline()	79
9.7	format.set_font_strikeout()	79
9.8	format.set_font_script()	80
9.9	format.set_num_format()	80
9.10	format.set_locked()	83
9.11	format.set_hidden()	83
9.12	format.set_align()	84
9.13	format.set_center_across()	84
9.14	format.set_text_wrap()	85
9.15	format.set_rotation()	85
9.16	format.set_indent()	86
9.17	format.set_shrink()	86
9.18	format.set_text_justlast()	86
9.19	format.set_pattern()	86
9.20	format.set_bg_color()	87
9.21	format.set_fg_color()	87
9.22	format.set_border()	88
9.23	format.set_bottom()	89
9.24	format.set_top()	89
9.25	format.set_left()	89
9.26	format.set_right()	89
9.27	format.set_border_color()	89
9.28	format.set_bottom_color()	90
9.29	format.set_top_color()	90
9.30	format.set_left_color()	90
9.31	format.set_right_color()	90
10	Working with Cell Notation	91
11	Working with Formats	93
11.1	Creating and using a Format object	93
11.2	Format methods and Format properties	93
11.3	Format Colors	95
11.4	Format Defaults	95
11.5	Modifying Formats	96
12	Working with Dates and Time	97
13	Working with Autofilters	101
13.1	Applying an autofilter	101
13.2	Filter data in an autofilter	102
13.3	Setting a filter criteria for a column	103
13.4	Setting a column list filter	104
13.5	Example	105
14	Working with Data Validation	107

14.1	<code>data_validation()</code>	109
14.2	Data Validation Examples	114
15	Working with Conditional Formatting	117
15.1	The <code>conditional_format()</code> method	120
15.2	Conditional Format Options	121
15.3	Conditional Formatting Examples	131
16	Working with Cell Comments	133
16.1	Setting Comment Properties	134
17	Working with Memory and Performance	137
17.1	Performance Figures	138
18	Examples	139
18.1	Example: Hello World	139
18.2	Example: Simple Feature Demonstration	140
18.3	Example: Dates and Times in Excel	141
18.4	Example: Adding hyperlinks	143
18.5	Example: Array formulas	145
18.6	Example: Applying Autofilters	147
18.7	Example: Data Validation and Drop Down Lists	152
18.8	Example: Conditional Formatting	157
18.9	Example: Merging Cells	162
18.10	Example: Writing “Rich” strings with multiple formats	163
18.11	Example: Inserting images into a worksheet	165
18.12	Example: Adding Headers and Footers to Worksheets	167
18.13	Example: Adding Cell Comments to Worksheets (Simple)	170
18.14	Example: Adding Cell Comments to Worksheets (Advanced)	172
18.15	Example: Setting Document Properties	177
18.16	Example: Unicode - Polish in UTF-8	179
18.17	Example: Unicode - Shift JIS	181
18.18	Example: Setting Worksheet Tab Colours	183
18.19	Example: Enabling Cell protection in Worksheets	184
19	Comparison with <code>Excel::Writer::XLSX</code>	187
19.1	Compatibility with <code>Excel::Writer::XLSX</code>	188
20	Alternative modules for handling Excel files	193
20.1	XLWT	193
20.2	XLRD	193
20.3	OpenPyXL	193
21	Known Issues and Bugs	195
21.1	‘unknown encoding: utf-8’ Error	195
21.2	Formula results not displaying in Excel	195
21.3	Formula results displaying as zero in non-Excel applications	195
21.4	Strings aren’t displayed in Apple Numbers in ‘constant_memory’ mode	196

22 Reporting Bugs	197
22.1 Upgrade to the latest version of the module	197
22.2 Read the documentation	197
22.3 Look at the example programs	197
22.4 Use the official XlsxWriter Issue tracker on GitHub	197
22.5 Pointers for submitting a bug report	197
23 Frequently Asked Questions	199
23.1 Q. Can XlsxWriter use an existing Excel file as a template?	199
23.2 Q. Why do my formulas show a zero result in some, non-Excel applications?	199
23.3 Q. Can I apply a format to a range of cells in one go?	199
23.4 Q. Is feature X supported or will it be supported?	200
23.5 Q. Is there an “AutoFit” option for columns?	200
23.6 Q. Do people actually ask these questions frequently, or at all?	200
24 Changes in XlsxWriter	201
24.1 Release 0.2.4 - March 31 2013	201
24.2 Release 0.2.3 - March 27 2013	201
24.3 Release 0.2.2 - March 27 2013	201
24.4 Release 0.2.1 - March 25 2013	201
24.5 Release 0.2.0 - March 24 2013	202
24.6 Release 0.1.9 - March 19 2013	202
24.7 Release 0.1.8 - March 18 2013	202
24.8 Release 0.1.7 - March 18 2013	202
24.9 Release 0.1.6 - March 17 2013	202
24.10 Release 0.1.5 - March 10 2013	202
24.11 Release 0.1.4 - March 8 2013	203
24.12 Release 0.1.3 - March 7 2013	203
24.13 Release 0.1.2 - March 6 2013	203
24.14 Release 0.1.1 - March 3 2013	203
24.15 Release 0.1.0 - February 28 2013	203
24.16 Release 0.0.9 - February 27 2013	204
24.17 Release 0.0.8 - February 26 2013	204
24.18 Release 0.0.7 - February 25 2013	204
24.19 Release 0.0.6 - February 22 2013	204
24.20 Release 0.0.5 - February 21 2013	205
24.21 Release 0.0.4 - February 20 2013	205
24.22 Release 0.0.3 - February 19 2013	205
24.23 Release 0.0.2 - February 18 2013	205
24.24 Release 0.0.1 - February 17 2013	206
25 Author	207
26 License	209
Index	211

XlsxWriter is a Python module for creating Excel XLSX files.



XlsxWriter supports the following features in version 0.2.4:

- 100% compatible Excel XLSX files.
- Write text, numbers, formulas, dates to cells.
- Write hyperlinks to cells.
- Full cell formatting.
- Multiple worksheets.
- Page setup methods for printing.
- Merged cells.
- Defined names.
- Autofilters.
- Data validation and drop down lists.
- Conditional formatting.
- Worksheet PNG/JPEG images.
- Rich multi-format strings.

- Cell comments.
- Document properties.
- Worksheet cell protection.
- Memory optimisation mode for writing large files.
- Standard libraries only.
- Python 2.6, 2.7, 3.1, 3.2 and 3.3 support.

INTRODUCTION

XlsxWriter is a Python module for writing files in the Excel 2007+ XLSX file format.

It can be used to write text, numbers, and formulas to multiple worksheets and it supports features such as formatting, images, page setup, autofilters, conditional formatting and many others.

This module cannot be used to modify or write to an existing Excel XLSX file. There are some *[Alternative modules for handling Excel files](#)* Python modules that do that.

XlsxWriter is written by John McNamara who also wrote the perl modules [Excel::Writer::XLSX](#) and [Spreadsheet::WriteExcel](#) and who is the maintainer of [Spreadsheet::ParseExcel](#). The XlsxWriter module is a port of the [Excel::Writer::XLSX](#). See the *[Comparison with Excel::Writer::XLSX](#)* section for a list of currently ported features.

XlsxWriter is intended to have a high degree of compatibility with files produced by Excel. In most cases the files produced are 100% equivalent to files produced by Excel and the [test suite](#) contains a large number of test cases that verify the output of XlsxWriter against actual files created in Excel.

XlsxWriter is licensed under a BSD *[License](#)* and is available as a git repository on [GitHub](#).

GETTING STARTED WITH XLSXWRITER

Here are some easy instructions to get you up and running with the XlsxWriter module.

2.1 Installing XlsxWriter

The first step is to install the XlsxWriter module. There are several ways to do this.

2.1.1 Using PIP

The `pip` installer is the preferred method for installing Python modules from [PyPI](#), the Python Package Index:

```
$ sudo pip install XlsxWriter
```

Note: Windows users can omit `sudo` at the start of the command.

2.1.2 Using Easy_Install

If `pip` doesn't work you can try `easy_install`:

```
$ sudo easy_install install XlsxWriter
```

2.1.3 Installing from a tarball

If you download a tarball of the latest version of XlsxWriter you can install it as follows (change the version number to suit):

```
$ tar -zxvf XlsxWriter-1.2.3.tar.gz
$ cd XlsxWriter-1.2.3
$ sudo python setup.py install
```

A tarball of the latest code can be downloaded from GitHub as follows:

```
$ curl -O -L http://github.com/jmcnamara/XlsxWriter/archive/master.tar.gz
$ tar zxvf master.tar.gz
$ cd XlsxWriter-master/
$ sudo python setup.py install
```

2.1.4 Cloning from GitHub

The XlsxWriter source code and bug tracker is in the [XlsxWriter repository](https://github.com/jmcnamara/XlsxWriter) on GitHub. You can clone the repository and install from it as follows:

```
$ git clone https://github.com/jmcnamara/XlsxWriter.git
$ cd XlsxWriter
$ sudo python setup.py install
```

2.2 Running a sample program

If the installation went correctly you can create a small sample program like the following to verify that the module works correctly:

```
from xlsxwriter.workbook import Workbook

workbook = Workbook('hello.xlsx')
worksheet = workbook.add_worksheet()

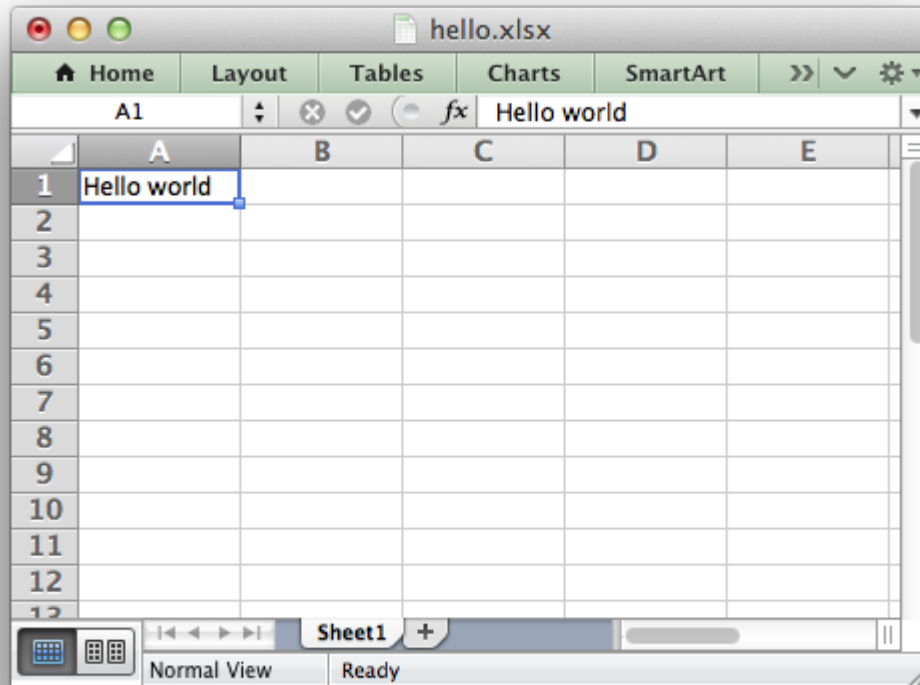
worksheet.write('A1', 'Hello world')

workbook.close()
```

Save this to a file called `hello.py` and run it as follows:

```
$ python hello.py
```

This will output a file called `hello.xlsx` which should look something like the following:



If you downloaded a tarball or cloned the repo, as shown above, you should also have a directory called [examples](#) with some sample applications that demonstrate different features of XlsxWriter.

2.3 Documentation

The latest version of this document is hosted on [Read The Docs](#). It is also available as a [PDF](#).

Once you are happy that the module is installed and operational you can have a look at the rest of the XlsxWriter documentation. [Tutorial 1: Create a simple XLSX file](#) is a good place to start.

TUTORIAL 1: CREATE A SIMPLE XLSX FILE

Let's start by creating a simple spreadsheet using Python and the XlsxWriter module.

Say that we have some data on monthly outgoings that we want to convert into an Excel XLSX file:

```
expenses = (  
    ['Rent', 1000],  
    ['Gas', 100],  
    ['Food', 300],  
    ['Gym', 50],  
)
```

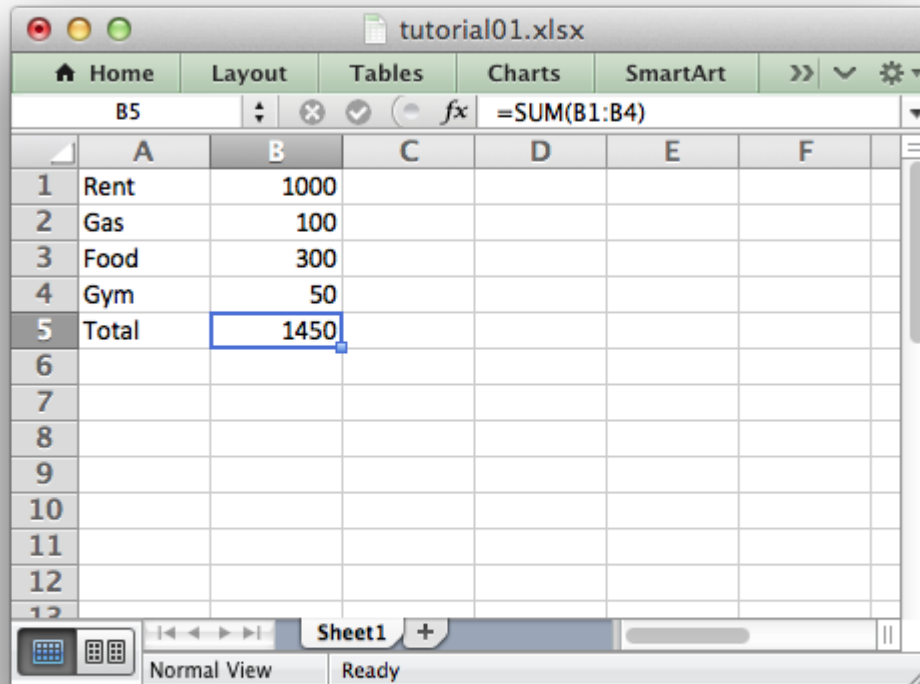
To do that we can start with a small program like the following:

```
from xlsxwriter.workbook import Workbook  
  
# Create a workbook and add a worksheet.  
workbook = Workbook('Expenses01.xlsx')  
worksheet = workbook.add_worksheet()  
  
# Some data we want to write to the worksheet.  
expenses = (  
    ['Rent', 1000],  
    ['Gas', 100],  
    ['Food', 300],  
    ['Gym', 50],  
)  
  
# Start from the first cell. Rows and columns are zero indexed.  
row = 0  
col = 0  
  
# Iterate over the data and write it out row by row.  
for item, cost in expenses:  
    worksheet.write(row, col, item)  
    worksheet.write(row, col + 1, cost)  
    row += 1
```

```
# Write a total using a formula.
worksheet.write(row, 0, 'Total')
worksheet.write(row, 1, '=SUM(B1:B4)')

workbook.close()
```

If we run this program we should get a spreadsheet that looks like this:



This is a simple example but the steps involved are representative of all programs that use XlsxWriter, so let's break it down into separate parts.

The first step is to import the module and the main method that we will call:

```
from xlsxwriter.workbook import Workbook
```

The next step is to create a new workbook object using the `Workbook()` constructor.

`Workbook()` takes one, non-optional, argument which is the filename that we want to create:

```
workbook = Workbook('Expenses01.xlsx')
```

Note: XlsxWriter can only create *new files*. It cannot read or modify existing files.

The workbook object is then used to add a new worksheet via the `add_worksheet()` method:

```
worksheet = workbook.add_worksheet()
```

By default worksheet names in the spreadsheet will be *Sheet1*, *Sheet2* etc., but we can also specify a name:

```
worksheet1 = workbook.add_worksheet()      # Defaults to Sheet1.
worksheet2 = workbook.add_worksheet('Data') # Data.
worksheet3 = workbook.add_worksheet()      # Defaults to Sheet3.
```

We can then use the worksheet object to write data via the `write()` method:

```
worksheet.write(row, col, some_data)
```

Note: Throughout XlsxWriter, *rows* and *columns* are zero indexed. The first cell in a worksheet, A1, is (0, 0).

So in our example we iterate over our data and write it out as follows:

```
# Iterate over the data and write it out row by row.
for item, cost in (expenses):
    worksheet.write(row, col, item)
    worksheet.write(row, col + 1, cost)
    row += 1
```

We then add a formula to calculate the total of the items in the second column:

```
worksheet.write(row, 1, '=SUM(B1:B4)')
```

Finally, we close the Excel file via the `close()` method:

```
workbook.close()
```

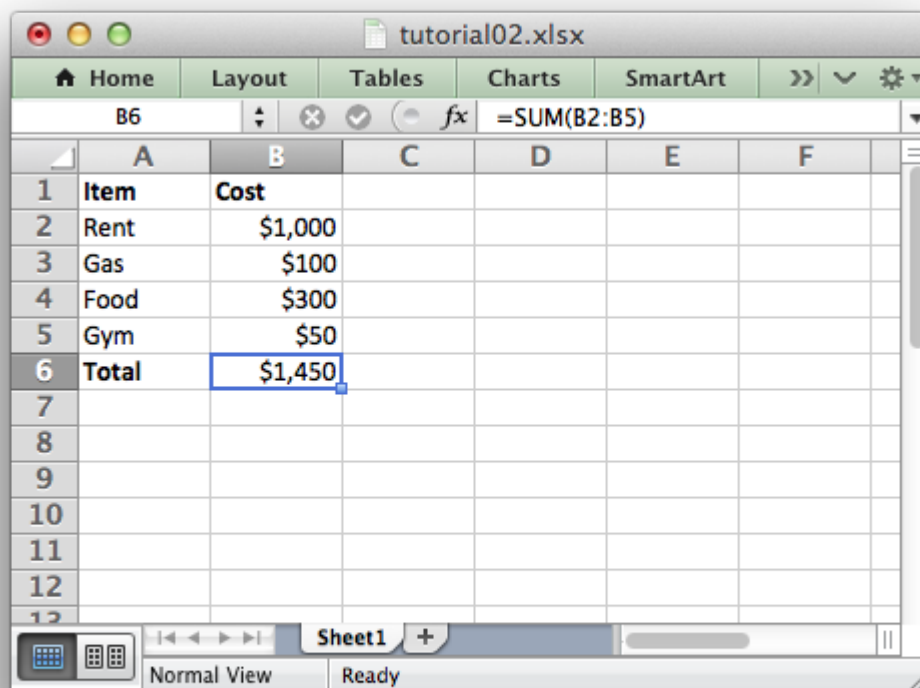
Like most file objects in Python an XlsxWriter file is closed implicitly when it goes out of scope or is no longer referenced in the program. As such this line is generally optional unless you need to close the file explicitly.

And that's it. We now have a file that can be read by Excel and other spreadsheet applications.

In the next sections we will see how we can use the XlsxWriter module to add formatting and other Excel features.

TUTORIAL 2: ADDING FORMATTING TO THE XLSX FILE

In the previous section we created a simple spreadsheet using Python and the XlsxWriter module. This converted the required data into an Excel file but it looked a little bare. In order to make the information clearer we would like to add some simple formatting, like this:



The screenshot shows an Excel spreadsheet with the following data:

	A	B	C	D	E	F
1	Item	Cost				
2	Rent	\$1,000				
3	Gas	\$100				
4	Food	\$300				
5	Gym	\$50				
6	Total	\$1,450				
7						
8						
9						
10						
11						
12						
13						

The spreadsheet interface includes a ribbon with tabs for Home, Layout, Tables, Charts, and SmartArt. The formula bar shows the formula `=SUM(B2:B5)` for cell B6. The status bar at the bottom indicates 'Normal View' and 'Ready'.

The differences here are that we have added **Item** and **Cost** column headers in a bold font, we have formatted the currency in the second column and we have made the **Total** string bold.

To do this we can extend our program as follows:

```
from xlsxwriter.workbook import Workbook

# Create a workbook and add a worksheet.
workbook = Workbook('Expenses02.xlsx')
worksheet = workbook.add_worksheet()

# Add a bold format to use to highlight cells.
bold = workbook.add_format({'bold': True})

# Add a number format for cells with money.
money = workbook.add_format({'num_format': '$#,##0'})

# Write some data header.
worksheet.write('A1', 'Item', bold)
worksheet.write('B1', 'Cost', bold)

# Some data we want to write to the worksheet.
expenses = (
    ['Rent', 1000],
    ['Gas', 100],
    ['Food', 300],
    ['Gym', 50],
)

# Start from the first cell below the headers.
row = 1
col = 0

# Iterate over the data and write it out row by row.
for item, cost in expenses:
    worksheet.write(row, col, item)
    worksheet.write(row, col + 1, cost, money)
    row += 1

# Write a total using a formula.
worksheet.write(row, 0, 'Total', bold)
worksheet.write(row, 1, '=SUM(B2:B5)', money)

workbook.close()
```

The main difference between this and the previous program is that we have added two *Format* objects that we can use to format cells in the spreadsheet.

Format objects represent all of the formatting properties that can be applied to a cell in Excel such as fonts, number formatting, colors and borders. This is explained in more detail in *The Format Class* and *Working with Formats*.

For now we will avoid the getting into the details and just use a limited amount of the format functionality to add some simple formatting:

```
# Add a bold format to use to highlight cells.
bold = workbook.add_format({'bold': True})
```

```
# Add a number format for cells with money.  
money = workbook.add_format({'num_format': '$#,##0'})
```

We can then pass these formats as an optional third parameter to the `worksheet.write()` method to format the data in the cell:

```
write(row, column, token, [format])
```

Like this:

```
worksheet.write(row, 0, 'Total', bold)
```

Which leads us to another new feature in this program. To add the headers in the first row of the worksheet we used `write()` like this:

```
worksheet.write('A1', 'Item', bold)  
worksheet.write('B1', 'Cost', bold)
```

So, instead of `(row, col)` we used the Excel 'A1' style notation. See [Working with Cell Notation](#) for more details but don't be too concerned about it for now. It is just a little syntactic sugar to help with laying out worksheets.

In the next section we will look at handling more data types.

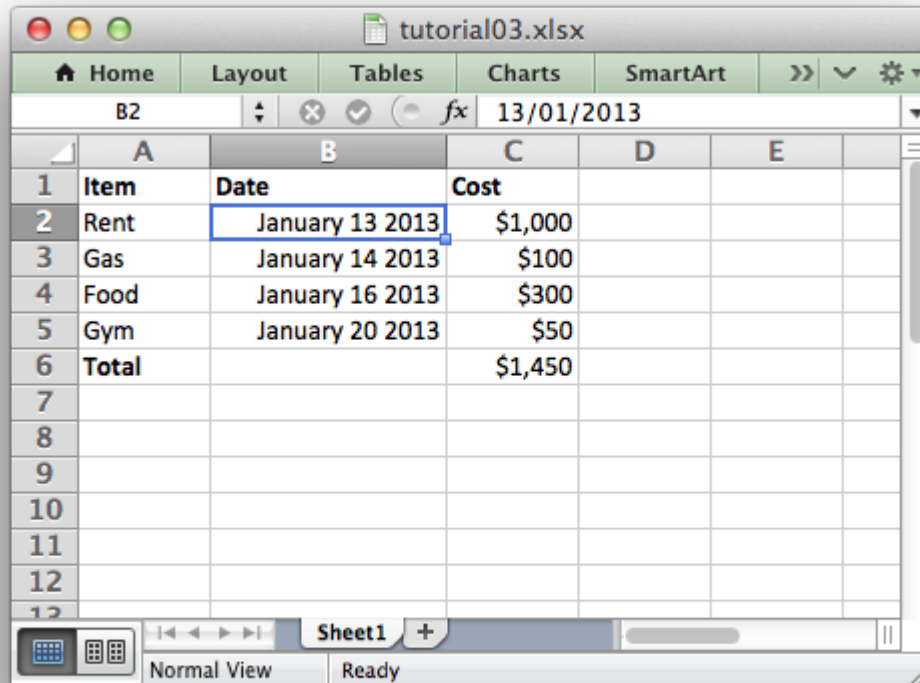
TUTORIAL 3: WRITING DIFFERENT TYPES OF DATA TO THE XLSX FILE

In the previous section we created a simple spreadsheet with formatting using Python and the `XlsxWriter` module.

This time let's extend the data we want to write to include some dates:

```
expenses = (  
    ['Rent', '2013-01-13', 1000],  
    ['Gas', '2013-01-14', 100],  
    ['Food', '2013-01-16', 300],  
    ['Gym', '2013-01-20', 50],  
)
```

The corresponding spreadsheet will look like this:



The screenshot shows an Excel spreadsheet with the following data:

	A	B	C	D	E
1	Item	Date	Cost		
2	Rent	January 13 2013	\$1,000		
3	Gas	January 14 2013	\$100		
4	Food	January 16 2013	\$300		
5	Gym	January 20 2013	\$50		
6	Total		\$1,450		
7					
8					
9					
10					
11					
12					
13					

The differences here are that we have added a **Date** column, formatted the dates and made column 'B' a little wider to accommodate the dates.

To do this we can extend our program as follows:

```
from datetime import datetime
from xlsxwriter.workbook import Workbook

# Create a workbook and add a worksheet.
workbook = Workbook('Expenses03.xlsx')
worksheet = workbook.add_worksheet()

# Add a bold format to use to highlight cells.
bold = workbook.add_format({'bold': 1})

# Add a number format for cells with money.
money_format = workbook.add_format({'num_format': '$#,##0'})

# Add an Excel date format.
date_format = workbook.add_format({'num_format': 'mmm d yyyy'})

# Adjust the column width.
worksheet.set_column(1, 1, 15)

# Write some data headers.
```

```

worksheet.write('A1', 'Item', bold)
worksheet.write('B1', 'Date', bold)
worksheet.write('C1', 'Cost', bold)

# Some data we want to write to the worksheet.
expenses = (
    ['Rent', '2013-01-13', 1000],
    ['Gas', '2013-01-14', 100],
    ['Food', '2013-01-16', 300],
    ['Gym', '2013-01-20', 50],
)

# Start from the first cell below the headers.
row = 1
col = 0

for item, date_str, cost in expenses:
    # Convert the date string into a datetime object.
    date = datetime.strptime(date_str, "%Y-%m-%d")

    worksheet.write_string(row, col, item)
    worksheet.write_datetime(row, col + 1, date, date_format)
    worksheet.write_number(row, col + 2, cost, money_format)
    row += 1

# Write a total using a formula.
worksheet.write(row, 0, 'Total', bold)
worksheet.write(row, 2, '=SUM(C2:C5)', money_format)

workbook.close()

```

The main difference between this and the previous program is that we have added a new *Format* object for dates and we have additional handling for data types.

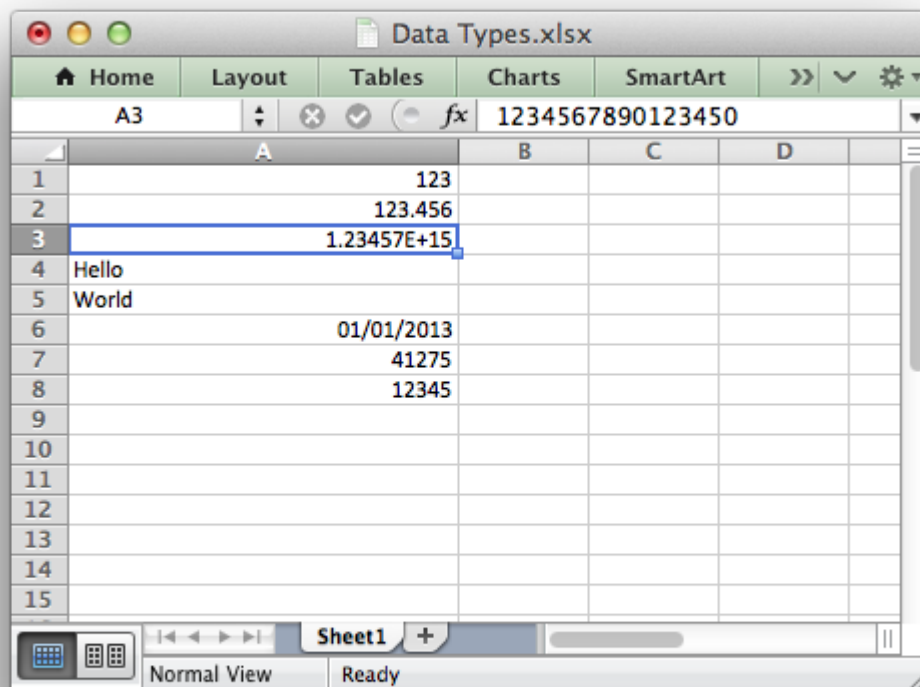
Excel treats different types of input data differently, although it generally does it transparently to the user. To illustrate this, open up a new Excel spreadsheet, make the first column wider and enter the following data:

```

123
123.456
1234567890123456
Hello
World
2013/01/01
2013/01/01          (But change the format from Date to General)
01234

```

You should see something like the following:



There are a few things to notice here. The first is that the numbers in the first three rows are stored as numbers and are aligned to the right of the cell. The second is that the strings in the following rows are stored as strings and are aligned to the left. The third is that the date string format has changed and that it is aligned to the right. The final thing to notice is that Excel has stripped the leading 0 from 012345.

Let's look at each of these in more detail.

Numbers are stored as numbers: In general Excel stores data as either strings or numbers. So it shouldn't be surprising that it stores numbers as numbers. Within a cell a number is right aligned by default. Internally Excel handles numbers as IEEE-754 64-bit double-precision floating point. This means that, in most cases, the maximum number of digits that can be stored in Excel without losing precision is 15. This can be seen in cell 'A3' where the 16 digit number has lost precision in the last digit.

Strings are stored as strings: Again not so surprising. Within a cell a string is left aligned by default. Excel 2007+ stores strings internally as UTF-8.

Dates are stored as numbers: The first clue to this is that the dates are right aligned like numbers. More explicitly, the data in cell 'A7' shows that if you remove the date format the underlying data is a number. When you enter a string that looks like a date Excel converts it to a number and applies the default date format to it so that it is displayed as a date. This is explained in more detail in [Working with Dates and Time](#).

Things that look like numbers are stored as numbers: In cell 'A8' we entered 012345 but Excel converted it to the number 12345. This is something to be aware of if you are writing ID numbers or Zip codes. In order to preserve the leading zero(es) you need to store the data as either a string or a number with a format.

XlsxWriter tries to mimic the way Excel works via the `worksheet.write()` method and separates Python data into types that Excel recognises. The `write()` method acts as a general alias for several more specific methods:

- `write_string()`
- `write_number()`
- `write_datetime()`
- `write_blank()`
- `write_formula()`

So, let's see how all of this affects our program.

The main change in our example program is the addition of date handling. As we saw above Excel stores dates as numbers. XlsxWriter makes the required conversion if the date and time are Python `datetime` objects. To convert the date strings in our example to `datetime.datetime` objects we use the `datetime.strptime` function. We then use the `write_datetime()` function to write it to a file. However, since the date is converted to a number we also need to add a number format to ensure that Excel displays it as a date:

```
from datetime import datetime
...

date_format = workbook.add_format({'num_format': 'mmm d yyyy'})
...

for item, date_str, cost in (expenses):
    # Convert the date string into a datetime object.
    date = datetime.strptime(date_str, "%Y-%m-%d")
    ...
    worksheet.write_datetime(row, col + 1, date, date_format )
    ...
```

The other thing to notice in our program is that we have used explicit write methods for different types of data:

```
worksheet.write_string (row, col, item )
worksheet.write_datetime(row, col + 1, date, date_format )
worksheet.write_number (row, col + 2, cost, money_format)
```

This is mainly to show that if you need more control over the type of data you write to a worksheet you can use the appropriate method. In this simplified example the `write()` method would have worked just as well but it is important to note that in cases where `write()` doesn't do the right thing, such as the number with leading zeroes discussed above, you will need to be explicit.

Finally, the last addition to our program is the `set_column()` method to adjust the width of column 'B' so that the dates are more clearly visible:

```
# Adjust the column width.  
worksheet.set_column('B:B', 15)
```

The `set_column()` and corresponding `set_row()` methods are explained in more detail in *The Worksheet Class*.

Next, let's look at *The Workbook Class* in more detail.

THE WORKBOOK CLASS

The Workbook class is the main class exposed by the XlsxWriter module and it is the only class that you will need to instantiate directly.

The Workbook class represents the entire spreadsheet as you see it in Excel and internally it represents the Excel file as it is written on disk.

6.1 Constructor

Workbook(*filename*[, *options*])

Create a new XlsxWriter Workbook object.

Parameters

- **filename** (*string*) – The name of the new Excel file to create.
- **options** (*dict*) – Optional workbook parameters.

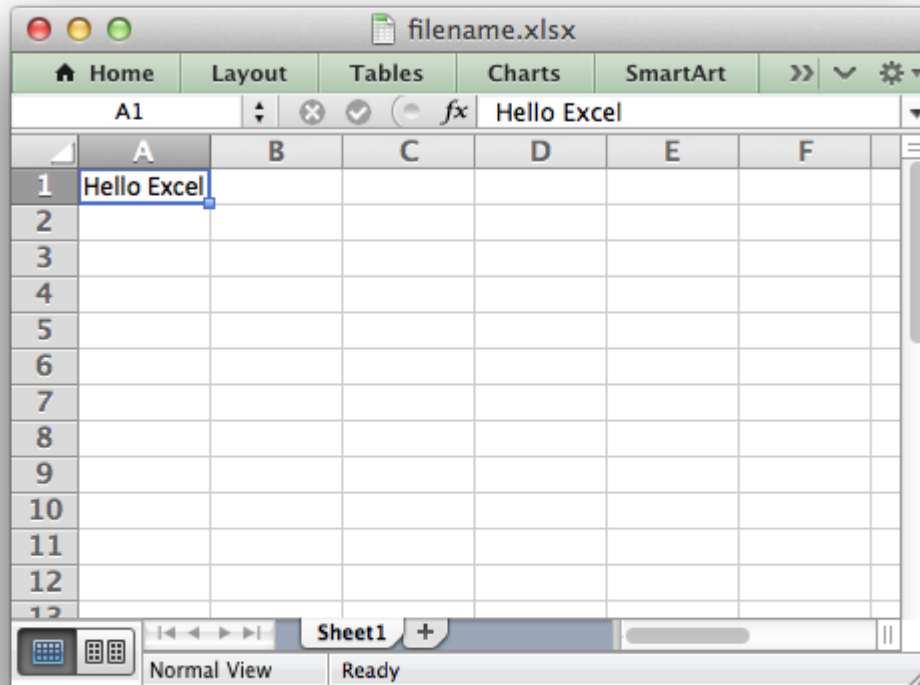
Return type A Workbook object.

The Workbook () constructor is used to create a new Excel workbook with a given filename:

```
from xlsxwriter import Workbook

workbook = Workbook('filename.xlsx')
worksheet = workbook.add_worksheet()

worksheet.write(0, 0, 'Hello Excel')
```



There is currently only one constructor option which is:

- **'constant_memory'**: Reduces the amount of data stored in memory so that large files can be written efficiently:

```
workbook = Workbook(filename, {'constant_memory': True})
```

Note, in this mode a row of data is written and then discarded when a cell in a new row is added via one of the worksheet `write_()` methods. As such data should be written in sequential row order once this mode is on.

See [Working with Memory and Performance](#) for more details.

It is recommended that you always use an `.xlsx` extension in the filename or Excel will generate a warning when the file is opened.

Note: A later version of the module will support writing to filehandles like `Excel::Writer::XLSX`.

6.2 workbook.add_worksheet()

add_worksheet ([*sheetname*])

Add a new worksheet to a workbook.

Parameters *sheetname* (*string*) – Optional worksheet name, defaults to Sheet1, etc.

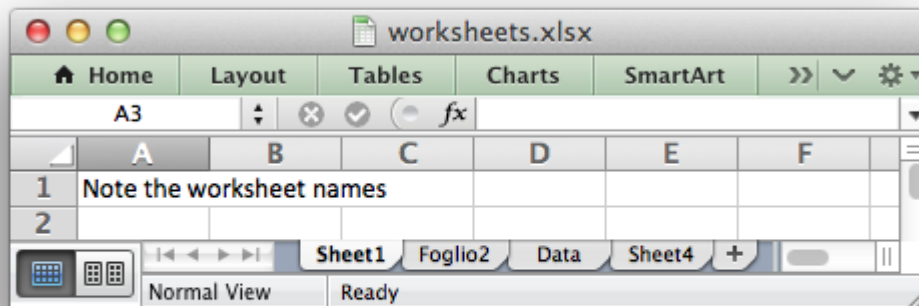
Return type A *Worksheet* object.

The `add_worksheet()` method adds a new worksheet to a workbook.

At least one worksheet should be added to a new workbook. The *Worksheet* object is used to write data and configure a worksheet in the workbook.

The *sheetname* parameter is optional. If it is not specified the default Excel convention will be followed, i.e. Sheet1, Sheet2, etc.:

```
worksheet1 = workbook.add_worksheet()           # Sheet1
worksheet2 = workbook.add_worksheet('Foglio2')  # Foglio2
worksheet3 = workbook.add_worksheet('Data')     # Data
worksheet4 = workbook.add_worksheet()           # Sheet4
```



The worksheet name must be a valid Excel worksheet name, i.e. it cannot contain any of the characters ' [] : * ? / \ ' and it must be less than 32 characters. In addition, you cannot use the same, case insensitive, *sheetname* for more than one worksheet.

6.3 workbook.add_format()

add_format ([*properties*])

Create a new Format object to format cells in worksheets.

Parameters *properties* (*dictionary*) – An optional dictionary of format properties.

Return type A *format* object.

The `add_format()` method can be used to create new *Format* objects which are used to apply formatting to a cell. You can either define the properties at creation time via a dictionary of property values or later via method calls:

```
format1 = workbook.add_format(props); # Set properties at creation.
format2 = workbook.add_format();     # Set properties later.
```

See the *The Format Class* and *Working with Formats* sections for more details about Format properties and how to set them.

6.4 workbook.close()

`close()`

Close the Workbook object and write the XLSX file.

In general your Excel file will be closed automatically when your program ends or when the Workbook object goes out of scope, however the `close()` method can be used to explicitly close an Excel file:

```
workbook.close()
```

An explicit `close()` is required if the file must be closed prior to performing some external action on it such as copying it, reading its size or attaching it to an email.

In addition, `close()` may be occasionally required to prevent Python's garbage collector from disposing of the Workbook, Worksheet and Format objects in the wrong order.

In general, if an XlsxWriter file is created with a size of 0 bytes or fails to be created for some unknown silent reason you should add `close()` to your program.

6.5 workbook.set_properties()

`set_properties()`

Set the document properties such as Title, Author etc.

Parameters `properties` (*dict*) – Dictionary of document properties.

The `set_properties` method can be used to set the document properties of the Excel file created by XlsxWriter. These properties are visible when you use the Office Button -> Prepare -> Properties option in Excel and are also available to external applications that read or index windows files.

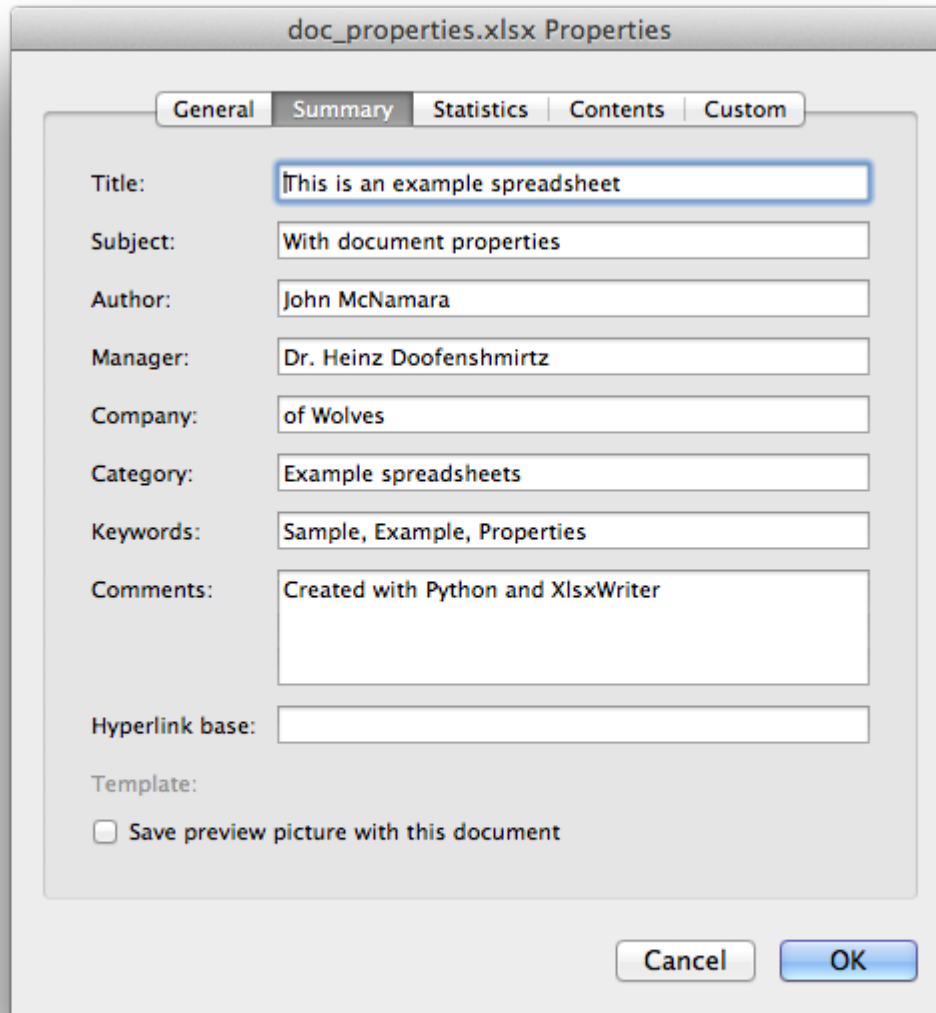
The properties that can be set are:

- title
- subject
- author
- manager

- company
- category
- keywords
- comments
- status

The properties should be passed in dictionary format as follows:

```
workbook.set_properties({
    'title':    'This is an example spreadsheet',
    'subject':  'With document properties',
    'author':   'John McNamara',
    'manager':  'Dr. Heinz Doofenshmirtz',
    'company':  'of Wolves',
    'category': 'Example spreadsheets',
    'keywords': 'Sample, Example, Properties',
    'comments': 'Created with Python and XlsxWriter'})
```



See also *Example: Setting Document Properties*.

6.6 `workbook.define_name()`

`define_name()`

Create a defined name in the workbook to use as a variable.

Parameters

- **name** (*string*) – The defined name.
- **formula** (*string*) – The cell or range that the defined name refers to.

This method is used to defined a name that can be used to represent a value, a single cell or a range of cells in a workbook.

For example to set a global/workbook name:

```
# Global/workbook names.
workbook.define_name('Exchange_rate', '=0.96')
workbook.define_name('Sales', '=Sheet1!$G$1:$H$10')
```

It is also possible to define a local/worksheet name by prefixing it with the sheet name using the syntax 'sheetname!definedname':

```
# Local/worksheet name.
workbook.define_name('Sheet2!Sales', '=Sheet2!$G$1:$G$10')
```

If the sheet name contains spaces or special characters you must enclose it in single quotes like in Excel:

```
workbook.define_name("'New Data'!Sales", '=Sheet2!$G$1:$G$10')
```

See also the `defined_name.py` program in the examples directory.

6.7 workbook.worksheets()

worksheets()

Return a list of the worksheet objects in the workbook.

Return type A list of *worksheet* objects.

The `worksheets()` method returns a list of the worksheets in a workbook. This is useful if you want to repeat an operation on each worksheet in a workbook:

```
for worksheet in workbook.worksheets():
    worksheet.write('A1', 'Hello')
```


THE WORKSHEET CLASS

The worksheet class represents an Excel worksheet. It handles operations such as writing data to cells or formatting worksheet layout.

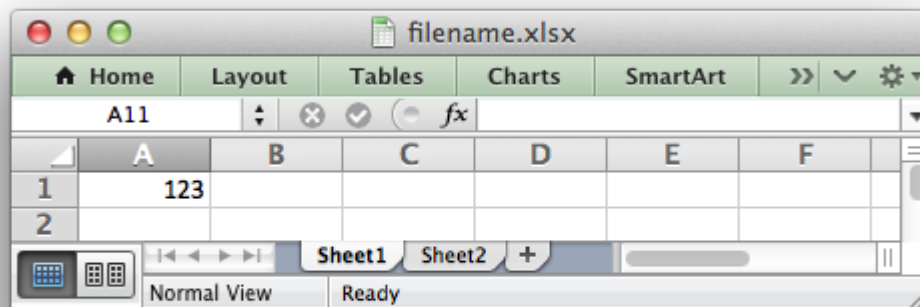
A worksheet object isn't instantiated directly. Instead a new worksheet is created by calling the `add_worksheet()` method from a `Workbook()` object:

```
workbook = Workbook('filename.xlsx')
```

```
worksheet1 = workbook.add_worksheet()
```

```
worksheet2 = workbook.add_worksheet()
```

```
worksheet1.write('A1', 123)
```



7.1 worksheet.write()

write(row, col, data[, cell_format])

Write generic data to a worksheet cell.

Parameters

- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **data** – Cell data to write. Variable types.
- **cell_format** (*Format*) – Optional Format object.

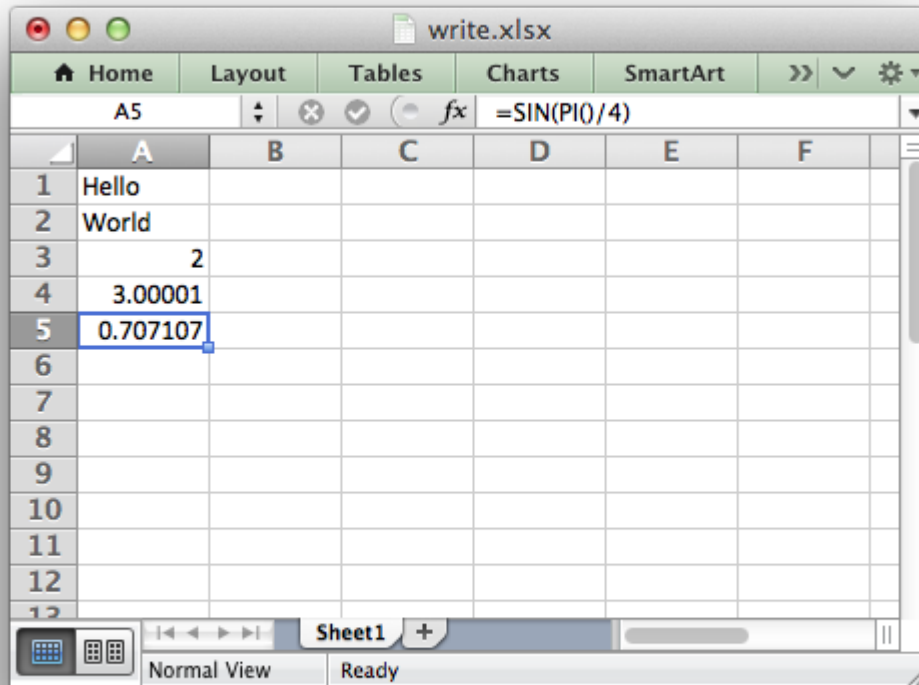
Excel makes a distinction between data types such as strings, numbers, blanks, formulas and hyperlinks. To simplify the process of writing data to an XlsxWriter file the `write()` method acts as a general alias for several more specific methods:

- `write_string()`
- `write_number()`
- `write_blank()`
- `write_formula()`
- `write_url()`

The general rule is that if the data looks like a *something* then a *something* is written. Here are some examples:

```
worksheet.write(0, 0, 'Hello')           # write_string()
worksheet.write(1, 0, 'World')           # write_string()
worksheet.write(2, 0, 2)                  # write_number()
worksheet.write(3, 0, 3.000001)           # write_number()
worksheet.write(4, 0, '=SIN(PI()/4)')     # write_formula()
worksheet.write(5, 0, '')                 # write_blank()
worksheet.write(6, 0, None)               # write_blank()
```

This creates a worksheet like the following:



The `write()` method supports two forms of notation to designate the position of cells: **Row-column** notation and **A1** notation:

```
# These are equivalent.
worksheet.write(0, 0, 'Hello')
worksheet.write('A1', 'Hello')
```

See [Working with Cell Notation](#) for more details.

The `cell_format` parameter is used to apply formatting to the cell. This parameter is optional but when present it should be a valid [Format](#) object:

```
cell_format = workbook.add_format({'bold': True, 'italic': True})

worksheet.write(0, 0, 'Hello', cell_format) # Cell is bold and italic.
```

The `write()` method will ignore empty strings or `None` unless a format is also supplied. As such you needn't worry about special handling for empty or `None` values in your data. See also the [write_blank\(\)](#) method.

One problem with the `write()` method is that occasionally data looks like a number but you don't want it treated as a number. For example, Zip codes or ID numbers or often start with a leading zero. If you write this data as a number then the leading zero(s) will be stripped. In this case you shouldn't use the `write()` method and should use `write_string()` instead.

7.2 worksheet.write_string()

write_string(*row*, *col*, *string*[, *cell_format*])

Write a string to a worksheet cell.

Parameters

- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **string** (*string*) – String to write to cell.
- **cell_format** (*Format*) – Optional Format object.

The `write_string()` method writes a string to the cell specified by row and column:

```
worksheet.write_string(0, 0, 'Your text here')
worksheet.write_string('A2', 'or here')
```

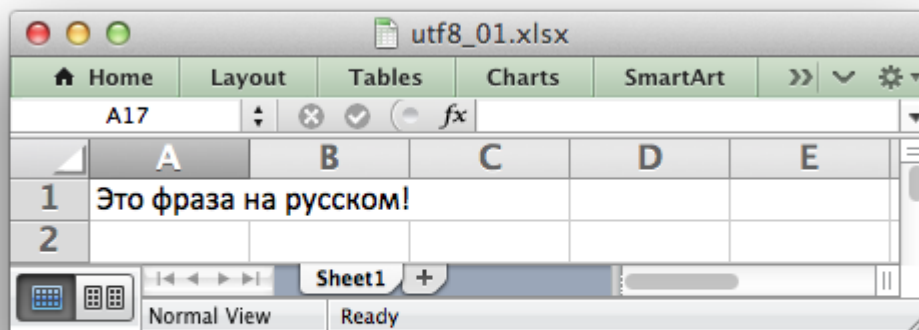
Both row-column and A1 style notation are support. See [Working with Cell Notation](#) for more details.

The `cell_format` parameter is used to apply formatting to the cell. This parameter is optional but when present is should be a valid [Format](#) object.

Unicode strings are supported in UTF-8 encoding. This generally requires that your source file in also UTF-8 encoded:

```
# -*- coding: utf-8

worksheet.write('A1', u'Some UTF-8 text')
```



Alternatively, you can read data from an encoded file, convert it to UTF-8 during reading and then write the data to an Excel file. There are several sample `unicode_*.py` programs like this in the `examples` directory of the XlsxWriter source tree.

The maximum string size supported by Excel is 32,767 characters. Strings longer than this will be truncated by `write_string()`.

Note: Even though Excel allows strings of 32,767 characters in a cell, Excel can only **display** 1000. All 32,767 characters are displayed in the formula bar.

In general it is sufficient to use the `write()` method when dealing with string data. However, you may sometimes need to use `write_string()` to write data that looks like a number but that you don't want treated as a number. For example, Zip codes or phone numbers:

```
# Write ID number as a plain string.
worksheet.write_string('A1', '01209')
```

However, if the user edits this string Excel may convert it back to a number. To get around this you can use the Excel text format '@':

```
# Format as a string. Doesn't change to a number when edited
str_format = workbook.add_format({'num_format', '@'})
worksheet.write_string('A1', '01209', str_format)
```

This behaviour, while slightly tedious, is unfortunately consistent with the way Excel handles string data that looks like numbers. See [Tutorial 3: Writing different types of data to the XLSX File](#).

7.3 worksheet.write_number()

write_number(*row*, *col*, *number*[, *cell_format*])

Write a number to a worksheet cell.

Parameters

- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **number** (*int or float*) – Number to write to cell.
- **cell_format** (*Format*) – Optional Format object.

The `write_number()` method writes an integer or a float to the cell specified by `row` and `col` - `umn`:

```
worksheet.write_number(0, 0, 123456)
worksheet.write_number('A2', 2.3451)
```

Both row-column and A1 style notation are support. See [Working with Cell Notation](#) for more details.

The `cell_format` parameter is used to apply formatting to the cell. This parameter is optional but when present is should be a valid *Format* object.

Excel handles numbers as IEEE-754 64-bit double-precision floating point. This means that, in most cases, the maximum number of digits that can be stored in Excel without losing precision is 15.

7.4 worksheet.write_formula()

write_formula(row, col, formula[, cell_format[, value]])

Write a formula to a worksheet cell.

Parameters

- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **formula** (*string*) – Formula to write to cell.
- **cell_format** (*Format*) – Optional Format object.

The `write_formula()` method writes a formula or function to the cell specified by row and column:

```
worksheet.write_formula(0, 0, '=B3 + B4')
worksheet.write_formula(1, 0, '=SIN(PI()/4)')
worksheet.write_formula(2, 0, '=SUM(B1:B5)')
worksheet.write_formula('A4', '=IF(A3>1,"Yes", "No")')
worksheet.write_formula('A5', '=AVERAGE(1, 2, 3, 4)')
worksheet.write_formula('A6', '=DATEVALUE("1-Jan-2013")')
```

Array formulas are also supported:

```
worksheet.write_formula('A7', '{=SUM(A1:B1*A2:B2)}')
```

See also the `write_array_formula()` method below.

Both row-column and A1 style notation are support. See [Working with Cell Notation](#) for more details.

The `cell_format` parameter is used to apply formatting to the cell. This parameter is optional but when present is should be a valid [Format](#) object.

XlsxWriter doesn't calculate the value of a formula and instead stores the value 0 as the formula result. It then sets a global flag in the XLSX file to say that all formulas and functions should be recalculated when the file is opened. This is the method recommended in the Excel documentation and in general it works fine with spreadsheet applications. However, applications that don't have a facility to calculate formulas, such as Excel Viewer, or some mobile applications will only display the 0 results.

If required, it is also possible to specify the calculated result of the formula using the `value` parameter. This is occasionally necessary when working with non-Excel applications that don't calculate the value of the formula. The calculated value is added at the end of the argument list:

```
worksheet.write('A1', '=2+2', num_format, 4)
```

Note: Some early versions of Excel 2007 do not display the calculated values of formulas written by XlsxWriter. Applying all available Office Service Packs should fix this.

7.5 worksheet.write_array_formula()

write_array_formula(*first_row*, *first_col*, *last_row*, *last_col*, *formula*[, *cell_format*[, *value*]])

Write an array formula to a worksheet cell.

Parameters

- **first_row** (*int*) – The first row of the range. (All zero indexed.)
- **first_col** (*int*) – The first column of the range.
- **last_row** (*int*) – The last row of the range.
- **last_col** (*int*) – The last col of the range.
- **formula** (*string*) – Array formula to write to cell.
- **cell_format** (*Format*) – Optional Format object.

The `write_array_formula()` method write an array formula to a cell range. In Excel an array formula is a formula that performs a calculation on a set of values. It can return a single value or a range of values.

An array formula is indicated by a pair of braces around the formula: `{=SUM(A1:B1*A2:B2)}`. If the array formula returns a single value then the `first_` and `last_` parameters should be the same:

```
worksheet.write_array_formula('A1:A1', '{=SUM(B1:C1*B2:C2)}')
```

It this case however it is easier to just use the `write_formula()` or `write()` methods:

```
# Same as above but more concise.
worksheet.write('A1', '{=SUM(B1:C1*B2:C2)}')
worksheet.write_formula('A1', '{=SUM(B1:C1*B2:C2)}')
```

For array formulas that return a range of values you must specify the range that the return values will be written to:

```
worksheet.write_array_formula('A1:A3', '{=TREND(C1:C3,B1:B3)}')
worksheet.write_array_formula(0, 0, 2, 0, '{=TREND(C1:C3,B1:B3)}')
```

As shown above, both row-column and A1 style notation are support. See [Working with Cell Notation](#) for more details.

The `cell_format` parameter is used to apply formatting to the cell. This parameter is optional but when present it should be a valid *Format* object.

If required, it is also possible to specify the calculated value of the formula. This is occasionally necessary when working with non-Excel applications that don't calculate the value of the formula. The calculated value is added at the end of the argument list:

```
worksheet.write_array_formula('A1:A3', '{=TREND(C1:C3,B1:B3)}', format, 105)
```

In addition, some early versions of Excel 2007 don't calculate the values of array formulas when they aren't supplied. Installing the latest Office Service Pack should fix this issue.

See also *Example: Array formulas*.

7.6 worksheet.write_blank()

write_blank(row, col, blank[, cell_format])

Write a blank worksheet cell.

Parameters

- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **blank** – None or empty string. The value is ignored.
- **cell_format** (*Format*) – Optional Format object.

Write a blank cell specified by row and column:

```
worksheet.write_blank(0, 0, None, format)
```

This method is used to add formatting to a cell which doesn't contain a string or number value.

Excel differentiates between an “Empty” cell and a “Blank” cell. An “Empty” cell is a cell which doesn't contain data whilst a “Blank” cell is a cell which doesn't contain data but does contain formatting. Excel stores “Blank” cells but ignores “Empty” cells.

As such, if you write an empty cell without formatting it is ignored:

```
worksheet.write('A1', None, format) # write_blank()
worksheet.write('A2', None)         # Ignored
```

This seemingly uninteresting fact means that you can write arrays of data without special treatment for None or empty string values.

As shown above, both row-column and A1 style notation are supported. See *Working with Cell Notation* for more details.

7.7 worksheet.write_datetime()

write_datetime(row, col, datetime[, cell_format])

Write a date or time to a worksheet cell.

Parameters

- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **datetime** (*datetime*) – A `datetime.datetime`, `.date` or `.time` object.
- **cell_format** (*Format*) – Optional `Format` object.

The `write_datetime()` method can be used to write a date or time to the cell specified by row and column:

```
worksheet.write_datetime(0, 0, datetime, date_format)
```

The `datetime` should be a `datetime.datetime`, `datetime.date` or `datetime.time` object. The `datetime` class is part of the standard Python libraries.

There are many way to create `datetime` objects, for example the `datetime.datetime.strptime()` method:

```
date_time = datetime.datetime.strptime('2013-01-23', '%Y-%m-%d')
```

See the `datetime` documentation for other date/time creation methods.

A date/time should always have a `cell_format` of type *Format*, otherwise it will appear as a number:

```
date_format = workbook.add_format({'num_format': 'd mmmm yyyy'})  
worksheet.write_datetime('A1', date_time, date_format)
```

See *Working with Dates and Time* for more details.

7.8 worksheet.write_url()

write_url(row, col, url[, cell_format[, string[, tip]]])

Write a hyperlink to a worksheet cell.

Parameters

- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **url** (*string*) – Hyperlink url.
- **cell_format** (*Format*) – Optional `Format` object.

- **string** (*string*) – An optional display string for the hyperlink.
- **tip** (*string*) – An optional tooltip.

The `write_url()` method is used to write a hyperlink in a worksheet cell. The url is comprised of two elements: the displayed string and the non-displayed link. The displayed string is the same as the link unless an alternative string is specified.

Both row-column and A1 style notation are support. See [Working with Cell Notation](#) for more details.

The `cell_format` parameter is used to apply formatting to the cell. This parameter is optional, however, without a format the link won't look like a link. The suggested *Format* is:

```
link_format = workbook.add_format({'color': 'blue', 'underline': 1})
```

There are four web style URI's supported: `http://`, `https://`, `ftp://` and `mailto::`

```
worksheet.write_url('A1', 'ftp://www.python.org/', link_format)
worksheet.write_url('A2', 'http://www.python.org/', link_format)
worksheet.write_url('A3', 'https://www.python.org/', link_format)
worksheet.write_url('A4', 'mailto:jmcnamaracpan.org', link_format)
```

All of these URI types are recognised by the `write()` method, so the following are equivalent:

```
worksheet.write_url('A2', 'http://www.python.org/', link_format)
worksheet.write      ('A2', 'http://www.python.org/', link_format)  # Same.
```

You can display an alternative string using the `string` parameter:

```
worksheet.write_url('A1', 'http://www.python.org', link_format, 'Python')
```

If you wish to have some other cell data such as a number or a formula you can overwrite the cell using another call to `write_*`():

```
worksheet.write_url('A1', 'http://www.python.org/', link_format)

# Overwrite the URL string with a formula. The cell is still a link.
worksheet.write_formula('A1', '=1+1', link_format)
```

There are two local URIs supported: `internal:` and `external:`. These are used for hyperlinks to internal worksheet references or external workbook and worksheet references:

```
worksheet.write_url('A1', 'internal:Sheet2!A1', link_format)
worksheet.write_url('A2', 'internal:Sheet2!A1', link_format)
worksheet.write_url('A3', 'internal:Sheet2!A1:B2', link_format)
worksheet.write_url('A4', 'internal:'Sales Data'!A1", link_format)
worksheet.write_url('A5', r'external:c:\temp\foo.xlsx', link_format)
worksheet.write_url('A6', r'external:c:\foo.xlsx#Sheet2!A1', link_format)
worksheet.write_url('A7', r'external:..\foo.xlsx', link_format)
worksheet.write_url('A8', r'external:..\foo.xlsx#Sheet2!A1', link_format)
worksheet.write_url('A9', r'external:\\NET\share\foo.xlsx', link_format)
```

Worksheet references are typically of the form `Sheet1!A1`. You can also link to a worksheet range using the standard Excel notation: `Sheet1!A1:B2`.

In external links the workbook and worksheet name must be separated by the # character: `external:Workbook.xlsx#Sheet1!A1`.

You can also link to a named range in the target worksheet. For example say you have a named range called `my_name` in the workbook `c:\temp\foo.xlsx` you could link to it as follows:

```
worksheet.write_url('A14', r'external:c:\temp\foo.xlsx#my_name')
```

Excel requires that worksheet names containing spaces or non alphanumeric characters are single quoted as follows `'Sales Data'!A1`.

Links to network files are also supported. Network files normally begin with two back slashes as follows `\\NETWORK\etc`. In order to generate this in a single or double quoted string you will have to escape the backslashes, `'\\\\NETWORK\\etc'` or use a raw string `r'\\NETWORK\etc'`.

Alternatively, you can avoid most of these quoting problems by using forward slashes. These are translated internally to backslashes:

```
worksheet.write_url('A14', "external:c:/temp/foo.xlsx")
worksheet.write_url('A15', 'external://NETWORK/share/foo.xlsx')
```

See also [Example: Adding hyperlinks](#).

Note: XlsxWriter will escape the following characters in URLs as required by Excel: `\s " < > \[] ' ^ { }` unless the URL already contains `%xx` style escapes. In which case it is assumed that the URL was escaped correctly by the user and will be passed directly to Excel.

7.9 worksheet.write_rich_string()

write_rich_string(row, col, *string_parts[, cell_format])

Write a “rich” string with multiple formats to a worksheet cell.

Parameters

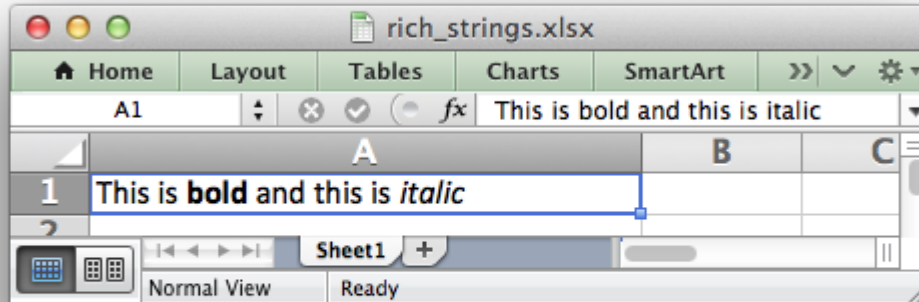
- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **string_parts** – String and format pairs.
- **cell_format** (*Format*) – Optional Format object.

The `write_rich_string()` method is used to write strings with multiple formats. For example to write the string “This is **bold** and this is *italic*” you would use the following:

```
bold = workbook.add_format({'bold': True})
italic = workbook.add_format({'italic': True})

worksheet.write_rich_string('A1',
                             'This is ',
                             bold, 'bold',
```

```
' and this is ',
italic, 'italic')
```



The basic rule is to break the string into fragments and put a `Format` object before the fragment that you want to format. For example:

```
# Unformatted string.
'This is an example string'

# Break it into fragments.
'This is an ', 'example', ' string'

# Add formatting before the fragments you want formatted.
'This is an ', format, 'example', ' string'

# In XlsxWriter.
worksheet.write_rich_string('A1',
                             'This is an ', format, 'example', ' string')
```

String fragments that don't have a format are given a default format. So for example when writing the string "Some **bold** text" you would use the first example below but it would be equivalent to the second:

```
# Some bold format and a default format.
bold    = workbook.add_format({'bold': True})
default = workbook.add_format()

# With default formatting:
worksheet.write_rich_string('A1',
                             'Some ',
                             bold, 'bold',
                             ' text')

# Or more explicitly:
worksheet.write_rich_string('A1',
                             default, 'Some ',
```

```
bold,      'bold',
default,   'text')
```

In Excel only the font properties of the format such as font name, style, size, underline, color and effects are applied to the string fragments in a rich string. Other features such as border, background, text wrap and alignment must be applied to the cell.

The `write_rich_string()` method allows you to do this by using the last argument as a cell format (if it is a format object). The following example centers a rich string in the cell:

```
bold    = workbook.add_format({'bold': True})
center  = workbook.add_format({'align': 'center'})

worksheet.write_rich_string('A5',
                             'Some ',
                             bold, 'bold text',
                             'centered',
                             center)
```

See also [Example: Writing “Rich” strings with multiple formats](#).

7.10 worksheet.write_row()

write_row(row, col, data[, cell_format])

Write a row of data starting from (row, col).

Parameters

- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **data** – Cell data to write. Variable types.
- **cell_format** (*Format*) – Optional Format object.

The `write_row()` method can be used to write a list of data in one go. This is useful for converting the results of a database query into an Excel worksheet. The `write()` method is called for each element of the data. For example:

```
# Some sample data.
data = ('Foo', 'Bar', 'Baz')

# Write the data to a sequence of cells.
worksheet.write_row('A1', data)

# The above example is equivalent to:
worksheet.write('A1', data[0])
worksheet.write('B1', data[1])
worksheet.write('C1', data[2])
```

7.11 worksheet.write_column()

write_column(*row*, *col*, *data*[, *cell_format*])

Write a column of data starting from (*row*, *col*).

Parameters

- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **data** – Cell data to write. Variable types.
- **cell_format** (*Format*) – Optional Format object.

The `write_column()` method can be used to write a list of data in one go. This is useful for converting the results of a database query into an Excel worksheet. The `write()` method is called for each element of the data. For example:

```
# Some sample data.
data = ('Foo', 'Bar', 'Baz')

# Write the data to a sequence of cells.
worksheet.write_row('A1', data)

# The above example is equivalent to:
worksheet.write('A1', data[0])
worksheet.write('A2', data[1])
worksheet.write('A3', data[2])
```

7.12 worksheet.set_row()

set_row(*row*, *height*, *cell_format*, *options*)

Set properties for a row of cells.

Parameters

- **row** (*int*) – The worksheet row (zero indexed).
- **height** (*int*) – The row height.
- **cell_format** (*Format*) – Optional Format object.
- **options** (*dict*) – Optional row parameters: hidden, level, collapsed.

The `set_row()` method is used to change the default properties of a row. The most common use for this method is to change the height of a row:

```
worksheet.set_row(0, 20) # Set the height of Row 1 to 20.
```

The other common use for `set_row()` is to set the *Format* for all cells in the row:

```
cell_format = workbook.add_format({'bold': True})

worksheet.set_row(0, 20, cell_format)
```

If you wish to set the format of a row without changing the height you can pass `None` as the height parameter or use the default row height of 15:

```
worksheet.set_row(1, None, cell_format)
worksheet.set_row(1, 15, cell_format) # Same as this.
```

The `cell_format` parameter will be applied to any cells in the row that don't have a format. As with Excel it is overridden by an explicit cell format. For example:

```
worksheet.set_row(0, None, format1) # Row 1 has format1.

worksheet.write('A1', 'Hello') # Cell A1 defaults to format1.
worksheet.write('B1', 'Hello', format2) # Cell B1 keeps format2.
```

The options parameter is a dictionary with the following possible keys:

- 'hidden'
- 'level'
- 'collapsed'

Options can be set as follows:

```
worksheet.set_row(0, 20, cell_format, {'hidden': 1})

# Or use defaults for other properties and set the options only.
worksheet.set_row(0, None, None, {'hidden': 1})
```

The 'hidden' option is used to hide a row. This can be used, for example, to hide intermediary steps in a complicated calculation:

```
worksheet.set_row(0, 20, cell_format, {'hidden': 1})
```

The 'level' parameter is used to set the outline level of the row. Outlines are described in "Working with Outlines and Grouping". Adjacent rows with the same outline level are grouped together into a single outline. (**Note:** This feature is not implemented yet).

The following example sets an outline level of 1 for some rows:

```
worksheet.set_row(0, None, None, {'level': 1})
worksheet.set_row(1, None, None, {'level': 1})
worksheet.set_row(2, None, None, {'level': 1})
```

Note: Excel allows up to 7 outline levels. The 'level' parameter should be in the range `0 <= level <= 7`.

The 'hidden' parameter can also be used to hide collapsed outlined rows when used in conjunction with the 'level' parameter:

```
worksheet.set_row(1, None, None, {'hidden': 1, 'level': 1})
worksheet.set_row(2, None, None, {'hidden': 1, 'level': 1})
```

The 'collapsed' parameter is used in collapsed outlines to indicate which row has the collapsed '+' symbol:

```
worksheet.set_row(3, None, None, {'collapsed': 1})
```

7.13 worksheet.set_column()

set_column(*first_col*, *last_col*, *width*, *cell_format*, *hidden*, *level*, *collapsed*)
Set properties for one or more columns of cells.

Parameters

- **first_col** (*int*) – First column (zero-indexed).
- **last_col** (*int*) – Last column (zero-indexed). Can be same as firstcol.
- **width** (*int*) – The width of the column(s).
- **cell_format** (*Format*) – Optional Format object.
- **options** (*dict*) – Optional parameters: hidden, level, collapsed.

The `set_column()` method can be used to change the default properties of a single column or a range of columns:

```
worksheet.set_column(1, 3, 30) # Width of columns B:D set to 30.
```

If `set_column()` is applied to a single column the value of `first_col` and `last_col` should be the same:

```
worksheet.set_column(1, 1, 30) # Width of column B set to 30.
```

It is also possible, and generally clearer, to specify a column range using the form of A1 notation used for columns. See [Working with Cell Notation](#) for more details.

Examples:

```
worksheet.set_column(0, 0, 20) # Column A width set to 20.
worksheet.set_column(1, 3, 30) # Columns B-D width set to 30.
worksheet.set_column('E:E', 20) # Column E width set to 20.
worksheet.set_column('F:H', 30) # Columns F-H width set to 30.
```

The width corresponds to the column width value that is specified in Excel. It is approximately equal to the length of a string in the default font of Calibri 11. Unfortunately, there is no way to specify “AutoFit” for a column in the Excel file format. This feature is only available at runtime from within Excel. It is possible to simulate “AutoFit” by tracking the width of the data in the column as you write it.

As usual the `cell_format` *Format* parameter is optional. If you wish to set the format without changing the width you can pass `None` as the width parameter:

```
cell_format = workbook.add_format({'bold': True})

worksheet.set_column(0, 0, None, cell_format)
```

The `cell_format` parameter will be applied to any cells in the column that don't have a format. For example:

```
worksheet.set_column('A:A', None, format1) # Col 1 has format1.

worksheet.write('A1', 'Hello')             # Cell A1 defaults to format1.
worksheet.write('A2', 'Hello', format2)    # Cell A2 keeps format2.
```

A row format takes precedence over a default column format:

```
worksheet.set_row(0, None, format1)         # Set format for row 1.
worksheet.set_column('A:A', None, format2)  # Set format for col 1.

worksheet.write('A1', 'Hello')             # Defaults to format1
worksheet.write('A2', 'Hello')             # Defaults to format2
```

The `options` parameter is a dictionary with the following possible keys:

- 'hidden'
- 'level'
- 'collapsed'

Options can be set as follows:

```
worksheet.set_column('D:D', 20, cell_format, {'hidden': 1})

# Or use defaults for other properties and set the options only.
worksheet.set_column('E:E', None, None, {'hidden': 1})
```

The 'hidden' option is used to hide a column. This can be used, for example, to hide intermediary steps in a complicated calculation:

```
worksheet.set_column('D:D', 20, cell_format, {'hidden': 1})
```

The 'level' parameter is used to set the outline level of the column. Outlines are described in "Working with Outlines and Grouping". Adjacent columns with the same outline level are grouped together into a single outline. (**Note:** This feature is not implemented yet).

The following example sets an outline level of 1 for columns B to G:

```
worksheet.set_column('B:G', None, None, {'level': 1})
```

Note: Excel allows up to 7 outline levels. The 'level' parameter should be in the range `0 <= level <= 7`.

The 'hidden' parameter can also be used to hide collapsed outlined columns when used in conjunction with the 'level' parameter:

```
worksheet.set_column('B:G', None, None, {'hidden': 1, 'level': 1})
```

The 'collapsed' parameter is used in collapsed outlines to indicate which column has the collapsed '+' symbol:

```
worksheet.set_column('H:H', None, None, {'collapsed': 1})
```

7.14 worksheet.insert_image()

insert_image(*row*, *col*, *image*[, *options*])

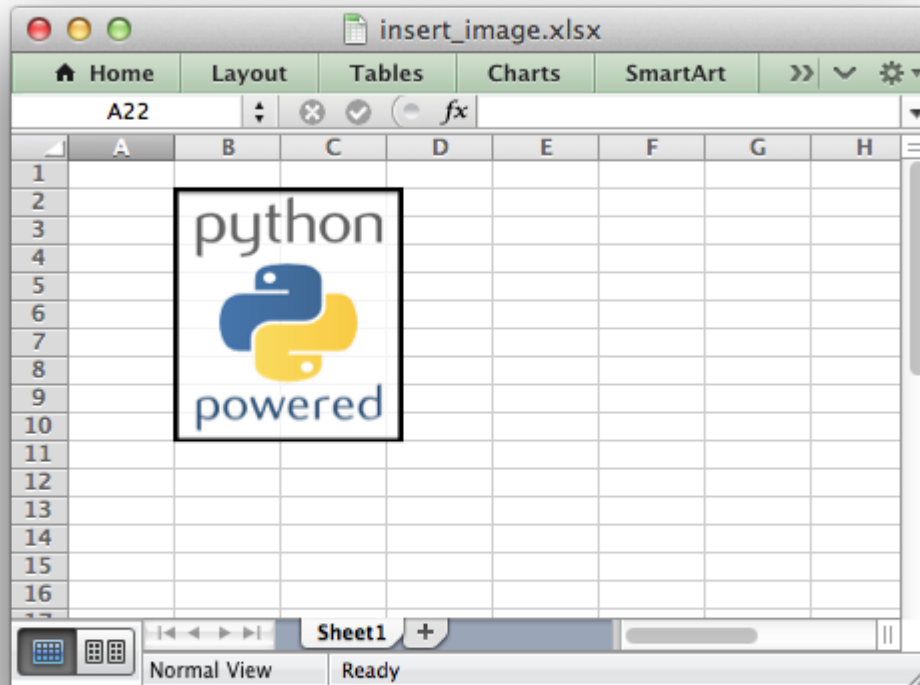
Write a string to a worksheet cell.

Parameters

- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **image** (*string*) – Image filename (with path if required).
- **options** (*dict*) – Optional parameters to position and scale the image.

This method can be used to insert a image into a worksheet. The image can be in PNG, JPEG or BMP format:

```
worksheet.insert_image('B2', 'python.png')
```

A file path can be specified with the image name:

```
worksheet1.insert_image('B10', '../images/python.png')
worksheet2.insert_image('B20', r'c:\images\python.png')
```

The `insert_image()` method takes optional parameters in a dictionary to position and scale the image. The available parameters with their default values are:

```
{
    'x_offset': 0,
    'y_offset': 0,
    'x_scale': 1,
    'y_scale': 1,
}
```

The offset values are in pixels:

```
worksheet1.insert_image('B2', 'python.png', {'x_offset': 15, 'y_offset': 10})
```

The offsets can be greater than the width or height of the underlying cell. This can be occasionally useful if you wish to align two or more images relative to the same cell.

The `x_scale` and `y_scale` parameters can be used to scale the image horizontally and vertically:

```
worksheet.insert_image('B3', 'python.png', {'x_scale': 0.5, 'y_scale': 0.5})
```

Currently only 96 dpi images are supported without modification. If you need to insert images with other dpi values you can use the scale parameters.

Note: You must call `set_row()` or `set_column()` before `insert_image()` if you wish to change the default dimensions of any of the rows or columns that the image occupies. The height of a row can also change if you use a font that is larger than the default or have text wrapping turned on. This in turn will affect the scaling of your image. To avoid this you should explicitly set the height of the row using `set_row()` if it contains a font size that will change the row height.

Inserting images into headers or a footers isn't supported.

BMP images are only supported for backward compatibility. In general it is best to avoid BMP images since they aren't compressed. If used, BMP images must be 24 bit, true colour, bitmaps.

See also [Example: Inserting images into a worksheet](#).

7.15 worksheet.data_validation()

data_validation(*first_row*, *first_col*, *last_row*, *last_col*, *options*)

Write a conditional format to range of cells.

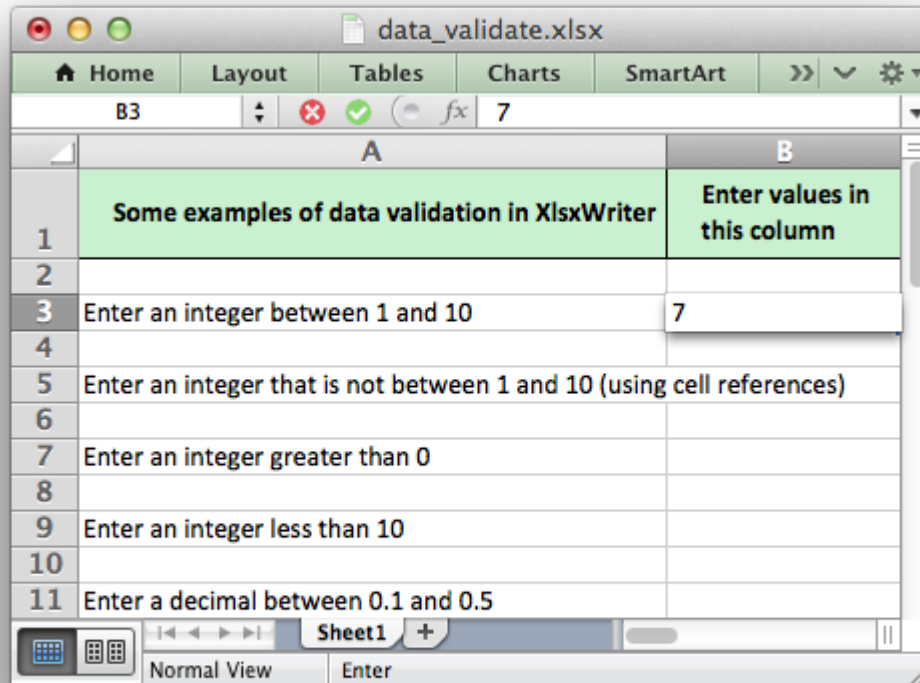
Parameters

- **first_row** (*int*) – The first row of the range. (All zero indexed.)
- **first_col** (*int*) – The first column of the range.
- **last_row** (*int*) – The last row of the range.
- **last_col** (*int*) – The last col of the range.
- **options** (*dict*) – Data validation options.

The `data_validation()` method is used to construct an Excel data validation or to limit the user input to a dropdown list of values:

```
worksheet.data_validation('B3', {'validate': 'integer',
                                'criteria': 'between',
                                'minimum': 1,
                                'maximum': 10})

worksheet.data_validation('B13', {'validate': 'list',
                                'source': ['open', 'high', 'close']})
```



The data validation can be applied to a single cell or a range of cells. As usual you can use A1 or Row/Column notation, see [Working with Cell Notation](#).

With Row/Column notation you must specify all four cells in the range: (*first_row*, *first_col*, *last_row*, *last_col*). If you need to refer to a single cell set the *last_* values equal to the *first_* values. With A1 notation you can refer to a single cell or a range of cells:

```
worksheet.data_validation(0, 0, 4, 1, {...})
worksheet.data_validation('B1',    {...})
worksheet.data_validation('C1:E5',  {...})
```

The options parameter in `data_validation()` must be a dictionary containing the parameters that describe the type and style of the data validation. There are a lot of available options which are described in detail in a separate section: [Working with Data Validation](#). See also [Example: Data Validation and Drop Down Lists](#).

7.16 worksheet.conditional_format()

conditional_format(*first_row*, *first_col*, *last_row*, *last_col*, *options*)

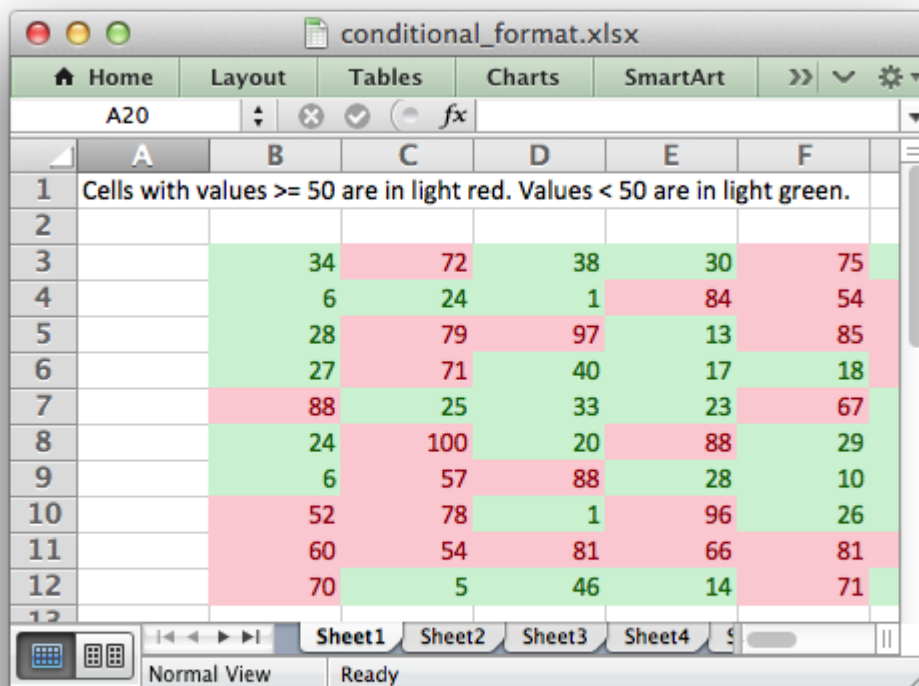
Write a conditional format to range of cells.

Parameters

- **first_row** (*int*) – The first row of the range. (All zero indexed.)
- **first_col** (*int*) – The first column of the range.
- **last_row** (*int*) – The last row of the range.
- **last_col** (*int*) – The last col of the range.
- **options** (*dict*) – Conditional formatting options.

The `conditional_format()` method is used to add formatting to a cell or range of cells based on user defined criteria:

```
worksheet.conditional_format('B3:K12', {'type': 'cell',
                                         'criteria': '>=',
                                         'value': 50,
                                         'format': format1})
```



The conditional format can be applied to a single cell or a range of cells. As usual you can use A1 or Row/Column notation, see [Working with Cell Notation](#).

With Row/Column notation you must specify all four cells in the range: (`first_row`, `first_col`, `last_row`, `last_col`). If you need to refer to a single cell set the `last_val` values equal to the `first` values. With A1 notation you can refer to a single cell or a range of cells:

```
worksheet.conditional_format(0, 0, 4, 1, {...})
worksheet.conditional_format('B1',      {...})
worksheet.conditional_format('C1:E5',   {...})
```

The options parameter in `conditional_format()` must be a dictionary containing the parameters that describe the type and style of the conditional format. There are a lot of available options which are described in detail in a separate section: [Working with Conditional Formatting](#). See also [Example: Conditional Formatting](#).

7.17 worksheet.write_comment()

write_comment(row, col, comment[, options])

Write a comment to a worksheet cell.

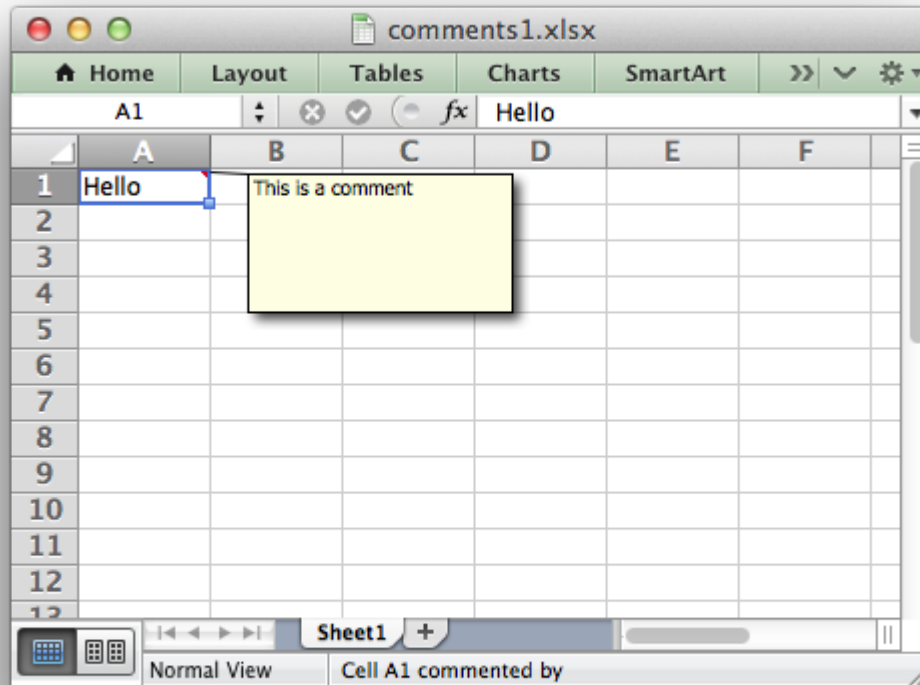
Parameters

- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **comment** (*string*) – String to write to cell.
- **options** (*dict*) – Comment formatting options..

The `write_comment()` method is used to add a comment to a cell. A comment is indicated in Excel by a small red triangle in the upper right-hand corner of the cell. Moving the cursor over the red triangle will reveal the comment.

The following example shows how to add a comment to a cell:

```
worksheet.write('A1', 'Hello')
worksheet.write_comment('A1', 'This is a comment')
```



As usual you can replace the row and col parameters with an A1 cell reference. See [Working with Cell Notation](#) for more details.

The properties of the cell comment can be modified by passing an optional dictionary of key/value pairs to control the format of the comment. For example:

```
worksheet.write_comment('C3', 'Hello', {'x_scale': 1.2, 'y_scale': 0.8})
```

Most of these options are quite specific and in general the default comment behaviour will be all that you need. However, should you need greater control over the format of the cell comment the following options are available:

```
author
visible
x_scale
width
y_scale
height
color
start_cell
start_row
start_col
x_offset
y_offset
```

For more details see [Working with Cell Comments](#) and [Example: Adding Cell Comments to Worksheets \(Advanced\)](#).

7.18 worksheet.show_comments()

show_comments()

Make any comments in the worksheet visible.

This method is used to make all cell comments visible when a worksheet is opened:

```
worksheet.show_comments()
```

Individual comments can be made visible using the `visible` parameter of the `write_comment` method (see above):

```
worksheet.write_comment('C3', 'Hello', {'visible': True})
```

If all of the cell comments have been made visible you can hide individual comments as follows:

```
worksheet.show_comments()  
worksheet.write_comment('C3', 'Hello', {'visible': False})
```

For more details see [Working with Cell Comments](#) and [Example: Adding Cell Comments to Worksheets \(Advanced\)](#).

7.19 worksheet.set_comments_author()

set_comments_author(author)

Set the default author of the cell comments.

Parameters `author` (*string*) – Comment author.

This method is used to set the default author of all cell comments:

```
worksheet.set_comments_author('John Smith')
```

Individual comment authors can be set using the `author` parameter of the `write_comment` method (see above).

If no author is specified the default comment author name is an empty string.

For more details see [Working with Cell Comments](#) and [Example: Adding Cell Comments to Worksheets \(Advanced\)](#).

7.20 worksheet.get_name()

get_name()

Retrieve the worksheet name.

The `get_name()` method is used to retrieve the name of a worksheet. This is something useful for debugging or logging:

```
for worksheet in workbook.worksheets():  
    print worksheet.get_name()
```

There is no `set_name()` method. The only safe way to set the worksheet name is via the `add_worksheet()` method.

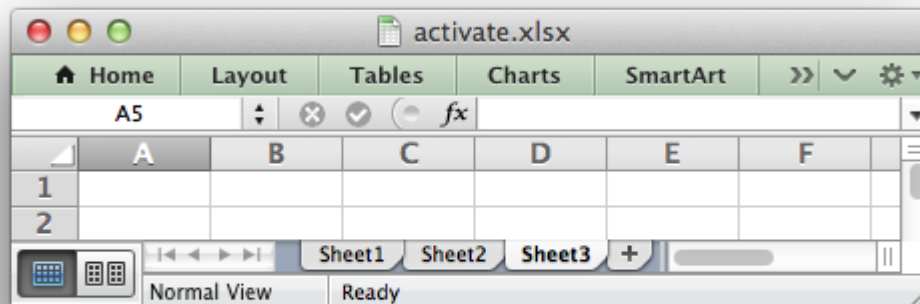
7.21 worksheet.activate()

activate()

Make a worksheet the active, i.e., visible worksheet.

The `activate()` method is used to specify which worksheet is initially visible in a multi-sheet workbook:

```
worksheet1 = workbook.add_worksheet()  
worksheet2 = workbook.add_worksheet()  
worksheet3 = workbook.add_worksheet()  
  
worksheet3.activate()
```



More than one worksheet can be selected via the `select()` method, see below, however only one worksheet can be active.

The default active worksheet is the first worksheet.

7.22 worksheet.select()

select()

Set a worksheet tab as selected.

The `select()` method is used to indicate that a worksheet is selected in a multi-sheet workbook:

```
worksheet1.activate()  
worksheet2.select()  
worksheet3.select()
```

A selected worksheet has its tab highlighted. Selecting worksheets is a way of grouping them together so that, for example, several worksheets could be printed in one go. A worksheet that has been activated via the `activate()` method will also appear as selected.

7.23 worksheet.hide()

hide()

Hide the current worksheet.

The `hide()` method is used to hide a worksheet:

```
worksheet2.hide()
```

You may wish to hide a worksheet in order to avoid confusing a user with intermediate data or calculations.

A hidden worksheet can not be activated or selected so this method is mutually exclusive with the `activate()` and `select()` methods. In addition, since the first worksheet will default to being the active worksheet, you cannot hide the first worksheet without activating another sheet:

```
worksheet2.activate()  
worksheet1.hide()
```

7.24 worksheet.set_first_sheet()

set_first_sheet()

Set current worksheet as the first visible sheet tab.

The `activate()` method determines which worksheet is initially selected. However, if there are a large number of worksheets the selected worksheet may not appear on the screen. To avoid this you can select which is the leftmost visible worksheet tab using `set_first_sheet()`:

```
for i in range(1, 21):  
    workbook.add_worksheet()  
  
worksheet19.set_first_sheet() # First visible worksheet tab.  
worksheet20.activate()       # First visible worksheet.
```

This method is not required very often. The default value is the first worksheet.

7.25 worksheet.merge_range()

merge_range (*first_row*, *first_col*, *last_row*, *last_col*, *cell_format*)

Merge a range of cells.

Parameters

- **first_row** (*int*) – The first row of the range. (All zero indexed.)
- **first_col** (*int*) – The first column of the range.
- **last_row** (*int*) – The last row of the range.
- **last_col** (*int*) – The last col of the range.
- **data** – Cell data to write. Variable types.
- **cell_format** (*Format*) – Optional Format object.

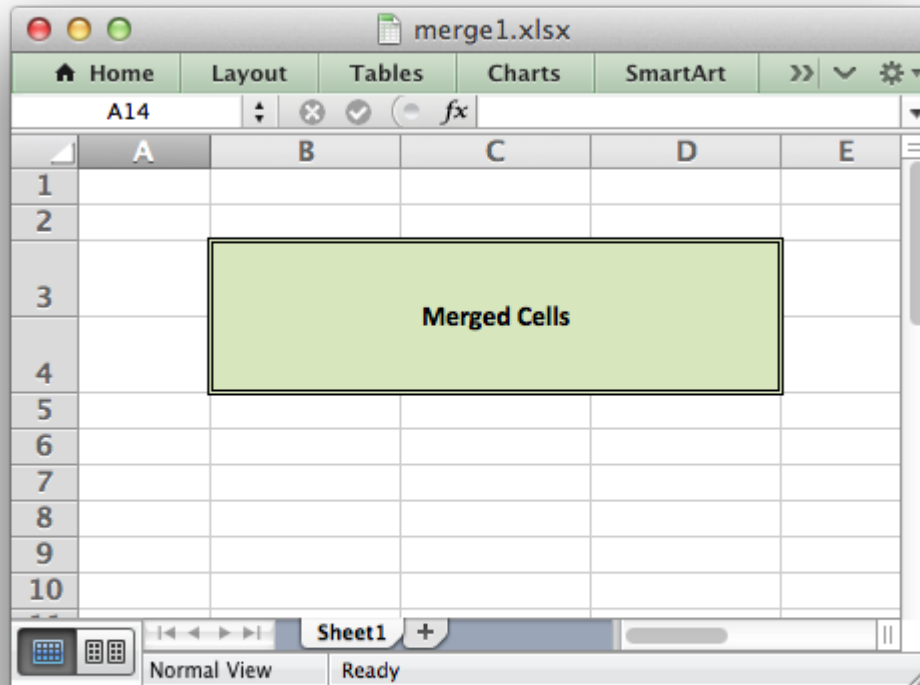
The `merge_range()` method allows cells to be merged together so that they act as a single area.

Excel generally merges and centers cells at same time. To get similar behaviour with XlsxWriter you need to apply a *Format*:

```
merge_format = workbook.add_format({'align': 'center'})  
  
worksheet.merge_range('B3:D4', 'Merged Cells', merge_format)
```

It is possible to apply other formatting to the merged cells as well:

```
merge_format = workbook.add_format({  
    'bold':      True,  
    'border':    6,  
    'align':     'center',  
    'valign':    'vcenter',  
    'fg_color':  '#D7E4BC',  
})  
  
worksheet.merge_range('B3:D4', 'Merged Cells', merge_format)
```



The `merge_range()` method writes its data argument using `write()`. Therefore it will handle numbers, strings and formulas as usual. If this doesn't handle your data correctly then you can overwrite the first cell with a call to one of the other `write_*()` methods using the same *Format* as in the merged cells.

See [Example: Merging Cells](#) for more details.

7.26 worksheet.autofilter()

autofilter(*first_row*, *first_col*, *last_row*, *last_col*)

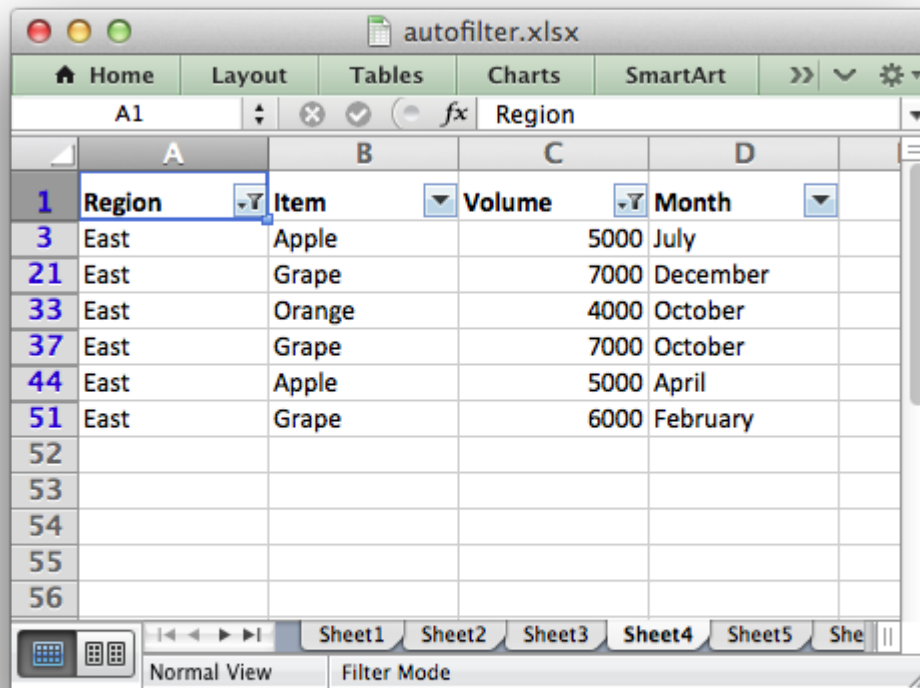
Set the autofilter area in the worksheet.

Parameters

- **first_row** (*int*) – The first row of the range. (All zero indexed.)
- **first_col** (*int*) – The first column of the range.
- **last_row** (*int*) – The last row of the range.
- **last_col** (*int*) – The last col of the range.

The `autofilter()` method allows an autofilter to be added to a worksheet. An autofilter is a way of adding drop down lists to the headers of a 2D range of worksheet data. This allows users

to filter the data based on simple criteria so that some data is shown and some is hidden.



To add an autofilter to a worksheet:

```
worksheet.autofilter('A1:D11')
worksheet.autofilter(0, 0, 10, 3) # Same as above.
```

Filter conditions can be applied using the `filter_column()` or `filter_column_list()` methods.

See [Working with Autofilters](#) for more details.

7.27 worksheet.filter_column()

filter_column(*col*, *criteria*)
Set the column filter criteria.

Parameters

- **col** (*int*) – Filter column (zero-indexed).
- **criteria** (*string*) – Filter criteria.

The `filter_column` method can be used to filter columns in a autofilter range based on simple conditions.

The conditions for the filter are specified using simple expressions:

```
worksheet.filter_column('A', 'x > 2000')
worksheet.filter_column('B', 'x > 2000 and x < 5000')
```

The `col` parameter can either be a zero indexed column number or a string column name.

It isn't sufficient to just specify the filter condition. You must also hide any rows that don't match the filter condition. See [Working with Autofilters](#) for more details.

7.28 worksheet.filter_column_list()

filter_column_list(*col*, *filters*)

Set the column filter criteria in Excel 2007 list style.

Parameters

- **col** (*int*) – Filter column (zero-indexed).
- **filters** (*list*) – List of filter criteria to match.

The `filter_column_list()` method can be used to represent filters with multiple selected criteria:

```
worksheet.filter_column_list('A', 'March', 'April', 'May')
```

The `col` parameter can either be a zero indexed column number or a string column name.

One or more criteria can be selected:

```
worksheet.filter_column_list('A', 'March')
worksheet.filter_column_list('C', 100, 110, 120, 130)
```

It isn't sufficient to just specify filters. You must also hide any rows that don't match the filter condition. See [Working with Autofilters](#) for more details.

7.29 worksheet.set_zoom()

set_zoom(*zoom*)

Set the worksheet zoom factor.

Parameters **zoom** (*int*) – Worksheet zoom factor.

Set the worksheet zoom factor in the range `10 <= zoom <= 400`:

```
worksheet1.set_zoom(50)
worksheet2.set_zoom(75)
worksheet3.set_zoom(300)
worksheet4.set_zoom(400)
```

The default zoom factor is 100. It isn't possible to set the zoom to "Selection" because it is calculated by Excel at run-time.

Note, `set_zoom()` does not affect the scale of the printed page. For that you should use `set_print_scale()`.

7.30 `worksheet.right_to_left()`

`right_to_left()`

Display the worksheet cells from right to left for some versions of Excel.

The `right_to_left()` method is used to change the default direction of the worksheet from left-to-right, with the A1 cell in the top left, to right-to-left, with the A1 cell in the top right.

```
worksheet.right_to_left()
```

This is useful when creating Arabic, Hebrew or other near or far eastern worksheets that use right-to-left as the default direction.

7.31 `worksheet.hide_zero()`

`hide_zero()`

Hide zero values in worksheet cells.

The `hide_zero()` method is used to hide any zero values that appear in cells:

```
worksheet.hide_zero()
```

7.32 `worksheet.set_tab_color()`

`set_tab_color()`

Set the colour of the worksheet tab.

Parameters `color` (*string*) – The tab color.

The `set_tab_color()` method is used to change the colour of the worksheet tab:

```
worksheet1.set_tab_color('red')
worksheet2.set_tab_color('#FF9900') # Orange
```

The colour can be a Html style #RRGGBB string or a limited number named colours, see [Format Colors](#).

See [Example: Setting Worksheet Tab Colours](#) for more details.

7.33 worksheet.protect()

protect()

Set the colour of the worksheet tab.

Parameters

- **password** (*string*) – A worksheet password.
- **options** (*dict*) – A dictionary of worksheet options to protect.

The `protect()` method is used to protect a worksheet from modification:

```
worksheet.protect()
```

The `protect()` method also has the effect of enabling a cell's locked and hidden properties if they have been set. A *locked* cell cannot be edited and this property is on by default for all cells. A *hidden* cell will display the results of a formula but not the formula itself. These properties can be set using the `set_locked()` and `set_hidden()` format methods.

You can optionally add a password to the worksheet protection:

```
worksheet.protect('abc123')
```

Passing the empty string "" is the same as turning on protection without a password.

You can specify which worksheet elements you wish to protect by passing a dictionary in the `options` argument with any or all of the following keys:

```
# Default values shown.
options = {
    'objects': 0,
    'scenarios': 0,
    'format_cells': 0,
    'format_columns': 0,
    'format_rows': 0,
    'insert_columns': 0,
    'insert_rows': 0,
    'insert_hyperlinks': 0,
    'delete_columns': 0,
    'delete_rows': 0,
    'select_locked_cells': 1,
    'sort': 0,
    'autofilter': 0,
    'pivot_tables': 0,
    'select_unlocked_cells': 1,
}
```

The default boolean values are shown above. Individual elements can be protected as follows:

```
worksheet.protect('abc123', { 'insert_rows': 1 })
```

See also the `set_locked()` and `set_hidden()` format methods and [Example: Enabling Cell protection in Worksheets](#).

Note: Worksheet level passwords in Excel offer very weak protection. They not encrypt your data and are very easy to deactivate. Full workbook encryption is not supported by XlsxWriter since it requires a completely different file format and would take several man months to implement.

THE WORKSHEET CLASS (PAGE SETUP)

Page set-up methods affect the way that a worksheet looks when it is printed. They control features such as paper size, orientation, page headers and margins.

These methods are really just standard *worksheet* methods. They are documented separately for the sake of clarity.

8.1 `worksheet.set_landscape()`

`set_landscape()`

Set the page orientation as landscape.

This method is used to set the orientation of a worksheet's printed page to landscape:

```
worksheet.set_landscape()
```

8.2 `worksheet.set_portrait()`

`set_portrait()`

Set the page orientation as portrait.

This method is used to set the orientation of a worksheet's printed page to portrait. The default worksheet orientation is portrait, so you won't generally need to call this method:

```
worksheet.set_portrait()
```

8.3 `worksheet.set_page_view()`

`set_page_view()`

Set the page view mode.

This method is used to display the worksheet in “Page View/Layout” mode:

```
worksheet.set_page_view()
```

8.4 worksheet.set_paper()

set_paper(*index*)

Set the paper type.

Parameters *index* (*int*) – The Excel paper format index.

This method is used to set the paper format for the printed output of a worksheet. The following paper styles are available:

Index	Paper format	Paper size
0	Printer default	
1	Letter	8 1/2 x 11 in
2	Letter Small	8 1/2 x 11 in
3	Tabloid	11 x 17 in
4	Ledger	17 x 11 in
5	Legal	8 1/2 x 14 in
6	Statement	5 1/2 x 8 1/2 in
7	Executive	7 1/4 x 10 1/2 in
8	A3	297 x 420 mm
9	A4	210 x 297 mm
10	A4 Small	210 x 297 mm
11	A5	148 x 210 mm
12	B4	250 x 354 mm
13	B5	182 x 257 mm
14	Folio	8 1/2 x 13 in
15	Quarto	215 x 275 mm
16		10x14 in
17		11x17 in
18	Note	8 1/2 x 11 in
19	Envelope 9	3 7/8 x 8 7/8
20	Envelope 10	4 1/8 x 9 1/2
21	Envelope 11	4 1/2 x 10 3/8
22	Envelope 12	4 3/4 x 11
23	Envelope 14	5 x 11 1/2
24	C size sheet	
25	D size sheet	
26	E size sheet	
27	Envelope DL	110 x 220 mm
28	Envelope C3	324 x 458 mm
29	Envelope C4	229 x 324 mm
Continued on next page		

Table 8.1 – continued from previous page

Index	Paper format	Paper size
30	Envelope C5	162 x 229 mm
31	Envelope C6	114 x 162 mm
32	Envelope C65	114 x 229 mm
33	Envelope B4	250 x 353 mm
34	Envelope B5	176 x 250 mm
35	Envelope B6	176 x 125 mm
36	Envelope	110 x 230 mm
37	Monarch	3.875 x 7.5 in
38	Envelope	3 5/8 x 6 1/2 in
39	Fanfold	14 7/8 x 11 in
40	German Std Fanfold	8 1/2 x 12 in
41	German Legal Fanfold	8 1/2 x 13 in

Note, it is likely that not all of these paper types will be available to the end user since it will depend on the paper formats that the user's printer supports. Therefore, it is best to stick to standard paper types:

```
worksheet.set_paper(1) # US Letter
worksheet.set_paper(9) # A4
```

If you do not specify a paper type the worksheet will print using the printer's default paper style.

8.5 worksheet.center_horizontally()

center_horizontally()

Center the printed page horizontally.

Center the worksheet data horizontally between the margins on the printed page:

```
worksheet.center_horizontally()
```

8.6 worksheet.center_vertically()

center_vertically()

Center the printed page vertically.

Center the worksheet data vertically between the margins on the printed page:

```
worksheet.center_vertically()
```

8.7 worksheet.worksheet.set_margins()

set_margins ([*left*=0.7,] *right*=0.7,] *top*=0.75,] *bottom*=0.75]]])

Set the worksheet margins for the printed page.

Parameters

- **left** (*float*) – Left margin in inches. Default 0.7.
- **right** (*float*) – Right margin in inches. Default 0.7.
- **top** (*float*) – Top margin in inches. Default 0.75.
- **bottom** (*float*) – Bottom margin in inches. Default 0.75.

The `set_margins()` method is used to set the margins of the worksheet when it is printed. The units are in inches. All parameters are optional and have default values corresponding to the default Excel values.

8.8 worksheet.set_header()

set_header ([*header*="",] *margin*=0.3]]])

Set the printed page header caption and optional margin.

Parameters

- **header** (*string*) – Header string with Excel control characters.
- **margin** (*float*) – Header margin in inches. Default 0.3.

Headers and footers are generated using a string which is a combination of plain text and control characters.

The available control character are:

Control	Category	Description
&L	Justification	Left
&C		Center
&R		Right
&P	Information	Page number
&N		Total number of pages
&D		Date
&T		Time
&F		File name
&A		Worksheet name
&Z		Workbook path
&fontsize	Font	Font size
&"font,style"		Font name and style
&U		Single underline
&E		Double underline
&S		Strikethrough
&X		Superscript
&Y		Subscript
&&	Miscellaneous	Literal ampersand &

Text in headers and footers can be justified (aligned) to the left, center and right by prefixing the text with the control characters &L, &C and &R.

For example (with ASCII art representation of the results):

```
worksheet.set_header('&LHello')
```

```
|-----|
| Hello |
|-----|
```

```
$worksheet->set_header('&CHello');
```

|

Hello

```
$worksheet->set_header('&RHello');
```

|
|
|
|

Hello

For simple text, if you do not specify any justification the text will be centred. However, you must prefix the text with &C if you specify a font name or any other formatting:

```
worksheet.set_header('Hello')
```

Hello

You can have text in each of the justification regions:

```
worksheet.set_header('&LCiao&CBello&RCielo')
```

Ciao	Bello	Cielo

The information control characters act as variables that Excel will update as the workbook or worksheet changes. Times and dates are in the users default format:

```
worksheet.set_header('&CPage &P of &N')
```

Page 1 of 6

```
worksheet.set_header('&CUpdated at &T')
```

Updated at 12:30 PM

You can specify the font size of a section of the text by prefixing it with the control character &n where n is the font size:

```
worksheet1.set_header('&C&30Hello Big')  
worksheet2.set_header('&C&10Hello Small')
```

You can specify the font of a section of the text by prefixing it with the control sequence &"font,style" where fontname is a font name such as "Courier New" or "Times New Roman" and style is one of the standard Windows font descriptions: "Regular", "Italic", "Bold" or "Bold Italic":

```
worksheet1.set_header('&C&"Courier New,Italic"Hello')  
worksheet2.set_header('&C&"Courier New,Bold Italic"Hello')  
worksheet3.set_header('&C&"Times New Roman,Regular"Hello')
```

It is possible to combine all of these features together to create sophisticated headers and footers. As an aid to setting up complicated headers and footers you can record a page set-up as a macro in Excel and look at the format strings that VBA produces. Remember however that VBA uses two double quotes "" to indicate a single double quote. For the last example above the equivalent

VBA code looks like this:

```
.LeftHeader = ""
.CenterHeader = "&"Times New Roman,Regular""Hello"
.RightHeader = ""
```

To include a single literal ampersand & in a header or footer you should use a double ampersand &&:

```
worksheet1.set_header('&&Curiouser and Curiouser - Attorneys at Law')
```

As stated above the margin parameter is optional. As with the other margins the value should be in inches. The default header and footer margin is 0.3 inch. The header and footer margin size can be set as follows:

```
worksheet.set_header('&CHello', 0.75)
```

The header and footer margins are independent of the top and bottom margins.

Note, the header or footer string must be less than 255 characters. Strings longer than this will not be written and an exception will be thrown.

See also [Example: Adding Headers and Footers to Worksheets](#).

8.9 worksheet.set_footer()

set_footer ([*footer*="",] *margin*=0.3])

Set the printed page footer caption and optional margin.

Parameters

- **footer** (*string*) – Footer string with Excel control characters.
- **margin** (*float*) – Footer margin in inches. Default 0.3.

The syntax of the `set_footer()` method is the same as `set_header()`.

8.10 worksheet.repeat_rows()

repeat_rows (*first_row*[, *last_row*])

Set the number of rows to repeat at the top of each printed page.

Parameters

- **first_row** (*int*) – First row of repeat range.
- **last_row** (*int*) – Last row of repeat range. Optional.

For large Excel documents it is often desirable to have the first row or rows of the worksheet print out at the top of each page.

This can be achieved by using the `repeat_rows()` method. The parameters `first_row` and `last_row` are zero based. The `last_row` parameter is optional if you only wish to specify one row:

```
worksheet1.repeat_rows(0)      # Repeat the first row.
worksheet2.repeat_rows(0, 1)   # Repeat the first two rows.
```

8.11 worksheet.repeat_columns()

repeat_columns(*first_col*[, *last_col*])

Set the columns to repeat at the left hand side of each printed page.

Parameters

- **first_col** (*int*) – First column of repeat range.
- **last_col** (*int*) – Last column of repeat range. Optional.

For large Excel documents it is often desirable to have the first column or columns of the worksheet print out at the left hand side of each page.

This can be achieved by using the `repeat_columns()` method. The parameters `first_column` and `last_column` are zero based. The `last_column` parameter is optional if you only wish to specify one column. You can also specify the columns using A1 column notation, see [Working with Cell Notation](#) for more details.:

```
worksheet1.repeat_columns(0)      # Repeat the first column.
worksheet2.repeat_columns(0, 1)   # Repeat the first two columns.
worksheet3.repeat_columns('A:A') # Repeat the first column.
worksheet4.repeat_columns('A:B') # Repeat the first two columns.
```

8.12 worksheet.hide_gridlines()

hide_gridlines([*option=1*])

Set the option to hide gridlines on the screen and the printed page.

Parameters *option* (*int*) – Hide gridline options. See below.

This method is used to hide the gridlines on the screen and printed page. Gridlines are the lines that divide the cells on a worksheet. Screen and printed gridlines are turned on by default in an Excel worksheet.

If you have defined your own cell borders you may wish to hide the default gridlines:

```
worksheet.hide_gridlines()
```

The following values of *option* are valid:

- 0. Don't hide gridlines.

1. Hide printed gridlines only.
2. Hide screen and printed gridlines.

If you don't supply an argument the default option is 1, i.e. only the printed gridlines are hidden.

8.13 `worksheet.print_row_col_headers()`

`print_row_col_headers()`

Set the option to print the row and column headers on the printed page.

When you print a worksheet from Excel you get the data selected in the print area. By default the Excel row and column headers (the row numbers on the left and the column letters at the top) aren't printed.

The `print_row_col_headers()` method sets the printer option to print these headers:

```
worksheet.print_row_col_headers()
```

8.14 `worksheet.print_area()`

`print_area(first_row, first_col, last_row, last_col)`

Set the print area in the current worksheet.

Parameters

- **first_row** (*integer*) – The first row of the range. (All zero indexed.)
- **first_col** (*integer*) – The first column of the range.
- **last_row** (*integer*) – The last row of the range.
- **last_col** (*integer*) – The last col of the range.
- **formula** – Array formula to write to cell.

This method is used to specify the area of the worksheet that will be printed.

All four parameters must be specified. You can also use A1 notation, see [Working with Cell Notation](#):

```
worksheet1.print_area('A1:H20')    # Cells A1 to H20.
worksheet2.print_area(0, 0, 19, 7) # The same as above.
worksheet3.print_area('A:H')       # Columns A to H if rows have data.
```

8.15 `worksheet.print_across()`

`print_across()`

Set the order in which pages are printed.

The `print_across` method is used to change the default print direction. This is referred to by Excel as the sheet “page order”:

```
worksheet.print_across()
```

The default page order is shown below for a worksheet that extends over 4 pages. The order is called “down then across”:

```
[1] [3]
[2] [4]
```

However, by using the `print_across` method the print order will be changed to “across then down”:

```
[1] [2]
[3] [4]
```

8.16 `worksheet.fit_to_pages()`

`fit_to_pages` (*width*, *height*)

Fit the printed area to a specific number of pages both vertically and horizontally.

Parameters

- **width** (*int*) – Number of pages horizontally.
- **height** (*int*) – Number of pages vertically.

The `fit_to_pages()` method is used to fit the printed area to a specific number of pages both vertically and horizontally. If the printed area exceeds the specified number of pages it will be scaled down to fit. This ensures that the printed area will always appear on the specified number of pages even if the page size or margins change:

```
worksheet1.fit_to_pages(1, 1) # Fit to 1x1 pages.
worksheet2.fit_to_pages(2, 1) # Fit to 2x1 pages.
worksheet3.fit_to_pages(1, 2) # Fit to 1x2 pages.
```

The print area can be defined using the `print_area()` method as described above.

A common requirement is to fit the printed output to *n* pages wide but have the height be as long as necessary. To achieve this set the height to zero:

```
worksheet1.fit_to_pages(1, 0) # 1 page wide and as long as necessary.
```

Note: Although it is valid to use both `fit_to_pages()` and `set_print_scale()` on the same worksheet only one of these options can be active at a time. The last method call made will set the active option.

Note: The `fit_to_pages()` will override any manual page breaks that are defined in the worksheet.

Note: When using `fit_to_pages()` it may also be required to set the printer paper size using `set_paper()` or else Excel will default to “US Letter”.

8.17 worksheet.set_start_page()

set_start_page()

Set the start page number when printing.

Parameters `start_page` (*int*) – Starting page number.

The `set_start_page()` method is used to set the number of the starting page when the worksheet is printed out:

```
worksheet.set_start_page(2)
```

8.18 worksheet.set_print_scale()

set_print_scale()

Set the scale factor for the printed page.

Parameters `scale` (*int*) – Print scale of worksheet to be printed.

Set the scale factor of the printed page. Scale factors in the range `10 <= $scale <= 400` are valid:

```
worksheet1.set_print_scale(50)
worksheet2.set_print_scale(75)
worksheet3.set_print_scale(300)
worksheet4.set_print_scale(400)
```

The default scale factor is 100. Note, `set_print_scale()` does not affect the scale of the visible page in Excel. For that you should use `set_zoom()`.

Note also that although it is valid to use both `fit_to_pages()` and `set_print_scale()` on the same worksheet only one of these options can be active at a time. The last method call made will set the active option.

8.19 worksheet.set_h_pagebreaks()

set_h_pagebreaks (*breaks*)

Set the horizontal page breaks on a worksheet.

Parameters `breaks` (*list*) – List of pagebreak rows.

The `set_h_pagebreaks()` method adds horizontal page breaks to a worksheet. A page break causes all the data that follows it to be printed on the next page. Horizontal page breaks act between rows. To create a page break between rows 20 and 21 you must specify the break at row 21. However in zero index notation this is actually row 20. So you can pretend for a small while that you are using 1 index notation:

```
worksheet1.set_h_pagebreaks([20]) # Break between row 20 and 21.
```

The `set_v_pagebreaks()` method takes a list of page breaks:

```
worksheet2.set_v_pagebreaks([20, 40, 60, 80, 100])
```

Note: Note: If you specify the “fit to page” option via the `fit_to_pages()` method it will override all manual page breaks.

There is a silent limitation of 1023 horizontal page breaks per worksheet in line with an Excel internal limitation.

8.20 worksheet.set_v_pagebreaks()

`set_v_pagebreaks(breaks)`

Set the vertical page breaks on a worksheet.

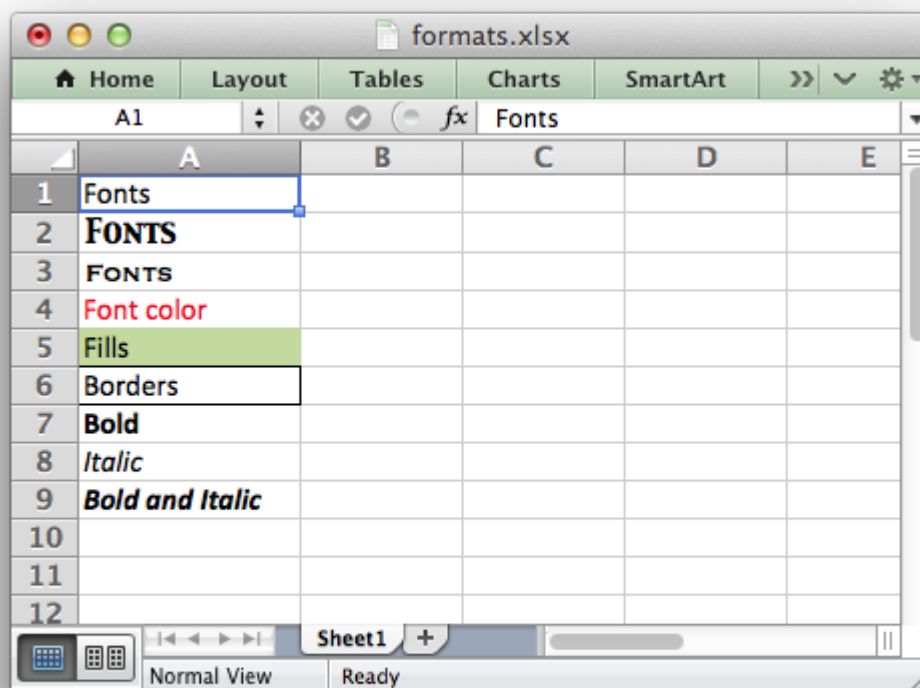
Parameters `breaks` (*list*) – List of pagebreak columns.

The `set_v_pagebreaks()` method is the same as the above `set_h_pagebreaks()` method except it adds page breaks between columns.

THE FORMAT CLASS

This section describes the methods and properties that are available for formatting cells in Excel.

The properties of a cell that can be formatted include: fonts, colours, patterns, borders, alignment and number formatting.



9.1 `format.set_font_name()`

`set_font_name(fontname)`

Set the font used in the cell.

Parameters `fontname` (*string*) – Cell font.

Specify the font used in the cell format:

```
cell_format.set_font_name('Times New Roman')
```

Excel can only display fonts that are installed on the system that it is running on. Therefore it is best to use the fonts that come as standard such as 'Calibri', 'Times New Roman' and 'Courier New'.

The default font for an unformatted cell in Excel 2007+ is 'Calibri'.

9.2 `format.set_font_size()`

`set_font_size(size)`

Set the size of the font used in the cell.

Parameters `size` (*int*) – The cell font size.

Set the font size of the cell format:

```
format = workbook.add_format()
format.set_font_size(30)
```

Excel adjusts the height of a row to accommodate the largest font size in the row. You can also explicitly specify the height of a row using the `set_row()` worksheet method.

9.3 `format.set_font_color()`

`set_font_color(color)`

Set the color of the font used in the cell.

Parameters `color` (*string*) – The cell font color.

Set the font colour:

```
format = workbook.add_format()

format.set_font_color('red')

worksheet.write(0, 0, 'wheelbarrow', format)
```

The color can be a Html style #RRGGBB string or a limited number of named colors, see [Format Colors](#).

Note: The `set_font_color()` method is used to set the colour of the font in a cell. To set the colour of a cell use the `set_bg_color()` and `set_pattern()` methods.

9.4 `format.set_bold()`

`set_bold()`

Turn on bold for the format font.

Set the bold property of the font:

```
format.set_bold()
```

9.5 `format.set_italic()`

`set_italic()`

Turn on italic for the format font.

Set the italic property of the font:

```
format.set_italic()
```

9.6 `format.set_underline()`

`set_underline()`

Turn on underline for the format.

Parameters `style (int)` – Underline style.

Set the underline property of the format:

```
format.set_underline()
```

The available underline styles are:

- 1 = Single underline (the default)
- 2 = Double underline
- 33 = Single accounting underline
- 34 = Double accounting underline

9.7 `format.set_font_strikeout()`

`set_font_strikeout()`

Set the strikeout property of the font.

9.8 format.set_font_script()

set_font_script()

Set the superscript/subscript property of the font.

The available options are:

- 1 = Superscript
- 2 = Subscript

9.9 format.set_num_format()

set_num_format(*format_string*)

Set the number format for a cell.

Parameters *format_string* (*string*) – The cell number format.

This method is used to define the numerical format of a number in Excel. It controls whether a number is displayed as an integer, a floating point number, a date, a currency value or some other user defined format.

The numerical format of a cell can be specified by using a format string or an index to one of Excel's built-in formats:

```
format1 = workbook.add_format()
format2 = workbook.add_format()

format1.set_num_format('d mmm yyyy') # Format string.
format2.set_num_format(0x0F)         # Format index.
```

Format strings can control any aspect of number formatting allowed by Excel:

```
format01.set_num_format('0.000')
worksheet.write(1, 0, 3.1415926, format01)      # -> 3.142

format02.set_num_format('#,##0')
worksheet.write(2, 0, 1234.56, format02)        # -> 1,235

format03.set_num_format('#,##0.00')
worksheet.write(3, 0, 1234.56, format03)        # -> 1,234.56

format04.set_num_format('0.00')
worksheet.write(4, 0, 49.99, format04)          # -> 49.99

format05.set_num_format('mm/dd/yy')
worksheet.write(5, 0, 36892.521, format05)      # -> 01/01/01

format06.set_num_format('mmm d yyyy')
worksheet.write(6, 0, 36892.521, format06)      # -> Jan 1 2001

format07.set_num_format('d mmmm yyyy')
```



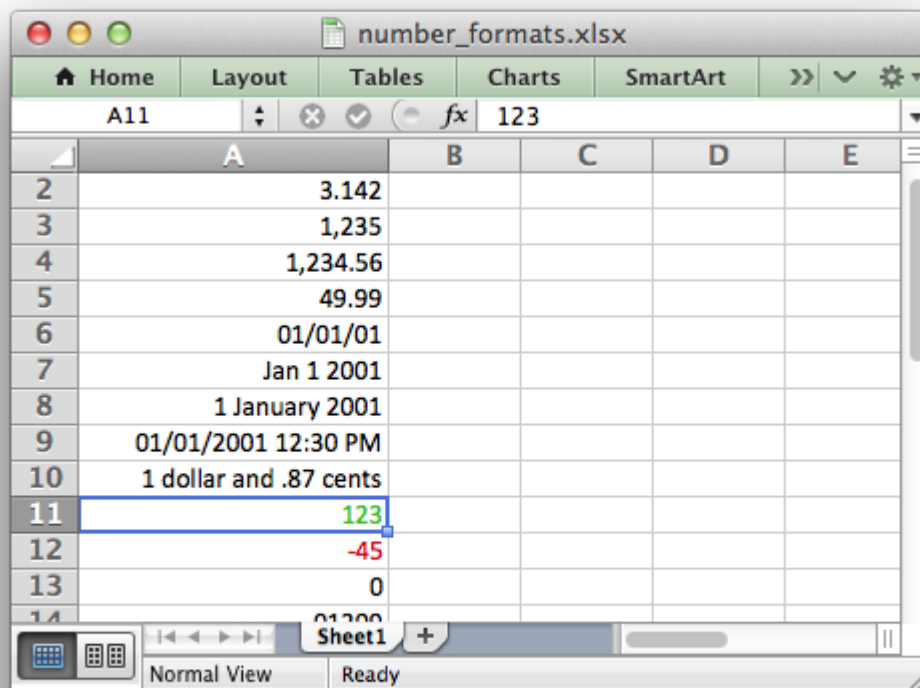
```
worksheet.write(7, 0, 36892.521, format07)           # -> 1 January 2001

format08.set_num_format('dd/mm/yyyy hh:mm AM/PM')
worksheet.write(8, 0, 36892.521, format08)           # -> 01/01/2001 12:30 AM

format09.set_num_format('0 "dollar and" .00 "cents"')
worksheet.write(9, 0, 1.87, format09)                # -> 1 dollar and .87 cents

# Conditional numerical formatting.
format10.set_num_format('[Green]General;[Red]-General;General')
worksheet.write(10, 0, 123, format10)                # > 0 Green
worksheet.write(11, 0, -45, format10)                # < 0 Red
worksheet.write(12, 0, 0, format10)                  # = 0 Default colour

# Zip code.
format11.set_num_format('000000')
worksheet.write(13, 0, 1209, format11)
```



The number system used for dates is described in [Working with Dates and Time](#).

The colour format should have one of the following values:

[Black] [Blue] [Cyan] [Green] [Magenta] [Red] [White] [Yellow]

For more information refer to the [Microsoft documentation on cell formats](#).

Excel's built-in formats are shown in the following table:

Index	Index	Format String
0	0x00	General
1	0x01	0
2	0x02	0.00
3	0x03	#,##0
4	0x04	#,##0.00
5	0x05	(\$#,##0_);(\$#,##0)
6	0x06	(\$#,##0_);[Red](\$#,##0)
7	0x07	(\$#,##0.00_);(\$#,##0.00)
8	0x08	(\$#,##0.00_);[Red](\$#,##0.00)
9	0x09	0%
10	0x0a	0.00%
11	0x0b	0.00E+00
12	0x0c	# ?/?
13	0x0d	# ??/??
14	0x0e	m/d/yy
15	0x0f	d-mmm-yy
16	0x10	d-mmm
17	0x11	mmm-yy
18	0x12	h:mm AM/PM
19	0x13	h:mm:ss AM/PM
20	0x14	h:mm
21	0x15	h:mm:ss
22	0x16	m/d/yy h:mm
...
37	0x25	(#,##0_);(#,##0)
38	0x26	(#,##0_);[Red](\$#,##0)
39	0x27	(#,##0.00_);(#,##0.00)
40	0x28	(#,##0.00_);[Red](\$#,##0.00)
41	0x29	_(* #,##0_);_(* (#,##0);_(* " - ");_(@)
42	0x2a	_(\$* #,##0_);_(\$* (#,##0);_(\$* " - ");_(@)
43	0x2b	_(* #,##0.00_);_(* (#,##0.00);_(* " - " ??);_(@)
44	0x2c	_(\$* #,##0.00_);_(\$* (#,##0.00);_(\$* " - " ??);_(@)
45	0x2d	mm:ss
46	0x2e	[h]:mm:ss
47	0x2f	mm:ss.0
48	0x30	##0.0E+0
49	0x31	@

Note: Numeric formats 23 to 36 are not documented by Microsoft and may differ in international versions.

Note: The dollar sign appears as the defined local currency symbol.

9.10 `format.set_locked()`

`set_locked(state)`

Set the cell locked state.

Parameters `state` (*bool*) – Turn cell locking on or off. Defaults to True.

This property can be used to prevent modification of a cells contents. Following Excel's convention, cell locking is turned on by default. However, it only has an effect if the worksheet has been protected using the worksheet `protect()` method:

```
locked = workbook.add_format()
locked.set_locked(True)

unlocked = workbook.add_format()
locked.set_locked(False)

# Enable worksheet protection
worksheet.protect()

# This cell cannot be edited.
worksheet.write('A1', '=1+2', locked)

# This cell can be edited.
worksheet.write('A2', '=1+2', unlocked)
```

9.11 `format.set_hidden()`

`set_hidden()`

Hide formulas in a cell.

This property is used to hide a formula while still displaying its result. This is generally used to hide complex calculations from end users who are only interested in the result. It only has an effect if the worksheet has been protected using the worksheet `protect()` method:

```
hidden = workbook.add_format()
hidden.set_hidden()

# Enable worksheet protection
worksheet.protect()

# The formula in this cell isn't visible
worksheet.write('A1', '=1+2', hidden)
```

9.12 format.set_align()

set_align() (*alignment*)

Set the alignment for data in the cell.

Parameters **alignment** (*string*) – The vertical and or horizontal alignment direction.

This method is used to set the horizontal and vertical text alignment within a cell. The following are the available horizontal alignments:

Horizontal alignment
center
right
fill
justify
center_across

The following are the available vertical alignments:

Vertical alignment
top
vcenter
bottom
vjustify

As in Excel, vertical and horizontal alignments can be combined:

```
format = workbook.add_format()

format.set_align('center')
format.set_align('vcenter')

worksheet.set_row(0, 30)
worksheet.write(0, 0, 'Some Text', format)
```

Text can be aligned across two or more adjacent cells using the 'center_across' property. However, for genuine merged cells it is better to use the `merge_range()` worksheet method.

The 'vjustify' (vertical justify) option can be used to provide automatic text wrapping in a cell. The height of the cell will be adjusted to accommodate the wrapped text. To specify where the text wraps use the `set_text_wrap()` method.

9.13 format.set_center_across()

set_center_across()

Centre text across adjacent cells.

Text can be aligned across two or more adjacent cells using the `set_center_across()` method. This is an alias for the `set_align('center_across')` method call.

Only one cell should contain the text, the other cells should be blank:

```
format = workbook.add_format()
format.set_center_across()

worksheet.write(1, 1, 'Center across selection', format)
worksheet.write_blank(1, 2, format)
```

For actual merged cells it is better to use the `merge_range()` worksheet method.

9.14 `format.set_text_wrap()`

`set_text_wrap()`

Wrap text in a cell.

Turn text wrapping on for text in a cell:

```
format = workbook.add_format()
format.set_text_wrap()

worksheet.write(0, 0, "Some long text to wrap in a cell", format)
```

If you wish to control where the text is wrapped you can add newline characters to the string:

```
format = workbook.add_format()
format.set_text_wrap()

worksheet.write(0, 0, "It's\na bum\nwrap", format)
```

Excel will adjust the height of the row to accommodate the wrapped text. A similar effect can be obtained without newlines using the `set_align('vjustify')` method.

9.15 `format.set_rotation()`

`set_rotation(angle)`

Set the rotation of the text in a cell.

Parameters *angle* (*int*) – Rotation angle in the range -90 to 90 and 270.

Set the rotation of the text in a cell. The rotation can be any angle in the range -90 to 90 degrees:

```
format = workbook.add_format()
format.set_rotation(30)

worksheet.write(0, 0, 'This text is rotated', format)
```

The angle 270 is also supported. This indicates text where the letters run from top to bottom.

9.16 format.set_indent()

set_indent() (*level*)

Set the cell text indentation level.

Parameters *level* (*int*) – Indentation level.

This method can be used to indent text in a cell. The argument, which should be an integer, is taken as the level of indentation:

```
format = workbook.add_format()
format.set_indent(2)

worksheet.write(0, 0, 'This text is indented', format)
```

Indentation is a horizontal alignment property. It will override any other horizontal properties but it can be used in conjunction with vertical properties.

9.17 format.set_shrink()

set_shrink()

Turn on the text “shrink to fit” for a cell.

This method can be used to shrink text so that it fits in a cell:

```
format = workbook.add_format()
format.set_shrink()

worksheet.write(0, 0, 'Honey, I shrunk the text!', format)
```

9.18 format.set_text_justlast()

set_text_justlast()

Turn on the justify last text property.

Only applies to Far Eastern versions of Excel.

9.19 format.set_pattern()

set_pattern() (*index*)

Parameters *index* (*int*) – Pattern index. 0 - 18.

Set the background pattern of a cell.

The most common pattern is 1 which is a solid fill of the background color.

9.20 format.set_bg_color()

`set_bg_color(color)`

Set the color of the background pattern in a cell.

Parameters `color` (*string*) – The cell font color.

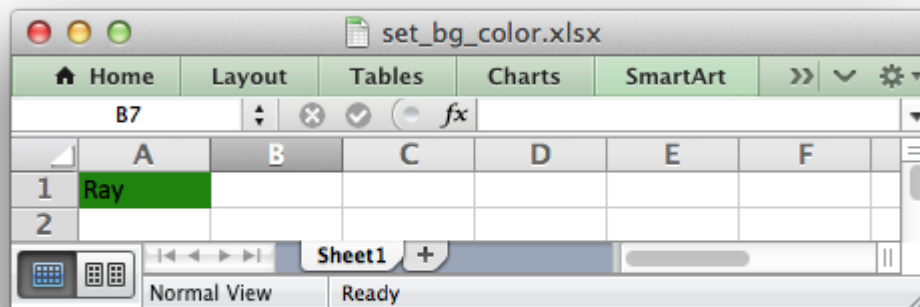
The `set_bg_color()` method can be used to set the background colour of a pattern. Patterns are defined via the `set_pattern()` method. If a pattern hasn't been defined then a solid fill pattern is used as the default.

Here is an example of how to set up a solid fill in a cell:

```
format = workbook.add_format()

format.set_pattern(1) # This is optional when using a solid fill.
format.set_bg_color('green')

worksheet.write('A1', 'Ray', format)
```



The color can be a Html style #RRGGBB string or a limited number of named colors, see [Format Colors](#).

9.21 format.set_fg_color()

`set_fg_color(color)`

Set the color of the foreground pattern in a cell.

Parameters `color` (*string*) – The cell font color.

The `set_fg_color()` method can be used to set the foreground colour of a pattern.

The color can be a Html style #RRGGBB string or a limited number of named colors, see [Format Colors](#).

9.22 format.set_border()

set_border(*style*)

Set the cell border style.

Parameters *style* (*int*) – Border style index. Default is 1.

Individual border elements can be configured using the following methods with the same parameters:

- `set_bottom()`
- `set_top()`
- `set_left()`
- `set_right()`

A cell border is comprised of a border on the bottom, top, left and right. These can be set to the same value using `set_border()` or individually using the relevant method calls shown above.

The following shows the border styles sorted by XlsxWriter index number:

Index	Name	Weight	Style
0	None	0	
1	Continuous	1	-----
2	Continuous	2	-----
3	Dash	1	- - - - -
4	Dot	1
5	Continuous	3	-----
6	Double	3	=====
7	Continuous	0	-----
8	Dash	2	- - - - -
9	Dash Dot	1	- . - . - .
10	Dash Dot	2	- . - . - .
11	Dash Dot Dot	1	- . . - . .
12	Dash Dot Dot	2	- . . - . .
13	SlantDash Dot	2	/ - . / - .

The following shows the borders in the order shown in the Excel Dialog:

Index	Style	Index	Style
0	None	12	- . . - . .
7	-----	13	/ - . / - .
4	10	- . - . - .
11	- . . - . .	8	- - - - -
9	- . - . - .	2	-----
3	- - - - -	5	-----
1	-----	6	=====

9.23 `format.set_bottom()`

`set_bottom(style)`

Set the cell bottom border style.

Parameters `style` (*int*) – Border style index. Default is 1.

Set the cell bottom border style. See `set_border()` for details on the border styles.

9.24 `format.set_top()`

`set_top(style)`

Set the cell top border style.

Parameters `style` (*int*) – Border style index. Default is 1.

Set the cell top border style. See `set_border()` for details on the border styles.

9.25 `format.set_left()`

`set_left(style)`

Set the cell left border style.

Parameters `style` (*int*) – Border style index. Default is 1.

Set the cell left border style. See `set_border()` for details on the border styles.

9.26 `format.set_right()`

`set_right(style)`

Set the cell right border style.

Parameters `style` (*int*) – Border style index. Default is 1.

Set the cell right border style. See `set_border()` for details on the border styles.

9.27 `format.set_border_color()`

`set_border_color(color)`

Set the color of the cell border.

Parameters `color` (*string*) – The cell border color.

Individual border elements can be configured using the following methods with the same parameters:

- `set_bottom_color()`

- `set_top_color()`
- `set_left_color()`
- `set_right_color()`

Set the colour of the cell borders. A cell border is comprised of a border on the bottom, top, left and right. These can be set to the same colour using `set_border_color()` or individually using the relevant method calls shown above.

The color can be a Html style `#RRGGBB` string or a limited number of named colors, see [Format Colors](#).

9.28 `format.set_bottom_color()`

`set_bottom_color(color)`

Set the color of the bottom cell border.

Parameters `color` (*string*) – The cell border color.

See `set_border_color()` for details on the border colors.

9.29 `format.set_top_color()`

`set_top_color(color)`

Set the color of the top cell border.

Parameters `color` (*string*) – The cell border color.

See `set_border_color()` for details on the border colors.

9.30 `format.set_left_color()`

`set_left_color(color)`

Set the color of the left cell border.

Parameters `color` (*string*) – The cell border color.

See `set_border_color()` for details on the border colors.

9.31 `format.set_right_color()`

`set_right_color(color)`

Set the color of the right cell border.

Parameters `color` (*string*) – The cell border color.

See `set_border_color()` for details on the border colors.

WORKING WITH CELL NOTATION

XlsxWriter supports two forms of notation to designate the position of cells: **Row-column** notation and **A1** notation.

Row-column notation uses a zero based index for both row and column while A1 notation uses the standard Excel alphanumeric sequence of column letter and 1-based row. For example:

```
(0, 0)      # Row-column notation.
('A1')     # The same cell in A1 notation.

(6, 2)     # Row-column notation.
('C7')     # The same cell in A1 notation.
```

Row-column notation is useful if you are referring to cells programmatically:

```
for row in range(0, 5):
    worksheet.write(row, 0, 'Hello')
```

A1 notation is useful for setting up a worksheet manually and for working with formulas:

```
worksheet.write('H1', 200)
worksheet.write('H2', '=H1+1')
```

In general when using the XlsxWriter module you can use A1 notation anywhere you can use row-column notation:

```
# These are equivalent.
worksheet.write(0, 7, 200)
worksheet.write('H1', 200)
```

The XlsxWriter utility contains several helper functions for dealing with A1 notation, for example:

```
from utility import xl_cell_to_rowcol, xl_rowcol_to_cell

(row, col) = xl_cell_to_rowcol('C2') # -> (1, 2)
string     = xl_rowcol_to_cell(1, 2)  # -> C2
```

Note: In Excel it is also possible to use R1C1 notation. This is not supported by XlsxWriter.

WORKING WITH FORMATS

The methods and properties used to add formatting to a cell are shown in *The Format Class*. This section provides some additional information about working with formats.

11.1 Creating and using a Format object

Cell formatting is defined through a *Format object*. Format objects are created by calling the `workbook.add_format()` method as follows:

```
format1 = workbook.add_format()      # Set properties later.
format2 = workbook.add_format(props) # Set properties at creation.
```

Once a Format object has been constructed and its properties have been set it can be passed as an argument to the worksheet write methods as follows:

```
worksheet.write      (0, 0, 'Foo', format)
worksheet.write_string(1, 0, 'Bar', format)
worksheet.write_number(2, 0, 3,      format)
worksheet.write_blank (3, 0, '',      format)
```

Formats can also be passed to the worksheet `set_row()` and `set_column()` methods to define the default property for a row or column:

```
worksheet.set_row(0, 18, format)
worksheet.set_column('A:D', 20, format)
```

11.2 Format methods and Format properties

The following table shows the Excel format categories, the formatting properties that can be applied and the equivalent object method:

Category	Description	Property	Method Name
Font	Font type	'font_name'	<code>set_font_name()</code>
Continued on next page			

Table 11.1 – continued from previous page

Category	Description	Property	Method Name
	Font size	'font_size'	set_font_size()
	Font color	'font_color'	set_font_color()
	Bold	'bold'	set_bold()
	Italic	'italic'	set_italic()
	Underline	'underline'	set_underline()
	Strikeout	'font_strikeout'	set_font_strikeout()
	Super/Subscript	'font_script'	set_font_script()
Number	Numeric format	'num_format'	set_num_format()
Protection	Lock cells	'locked'	set_locked()
	Hide formulas	'hidden'	set_hidden()
Alignment	Horizontal align	'align'	set_align()
	Vertical align	'valign'	set_align()
	Rotation	'rotation'	set_rotation()
	Text wrap	'text_wrap'	set_text_wrap()
	Justify last	'text_justlast'	set_text_justlast()
	Center across	'center_across'	set_center_across()
	Indentation	'indent'	set_indent()
	Shrink to fit	'shrink'	set_shrink()
Pattern	Cell pattern	'pattern'	set_pattern()
	Background color	'bg_color'	set_bg_color()
	Foreground color	'fg_color'	set_fg_color()
Border	Cell border	'border'	set_border()
	Bottom border	'bottom'	set_bottom()
	Top border	'top'	set_top()
	Left border	'left'	set_left()
	Right border	'right'	set_right()
	Border color	'border_color'	set_border_color()
	Bottom color	'bottom_color'	set_bottom_color()
	Top color	'top_color'	set_top_color()
	Left color	'left_color'	set_left_color()
	Right color	'right_color'	set_right_color()

There are two ways of setting Format properties: by using the object interface or by setting the property as a dictionary of key/value pairs in the constructor. For example, a typical use of the object interface would be as follows:

```
format = workbook.add_format()
format.set_bold()
format.set_font_color('red')
```

By comparison the properties can be set by passing a dictionary of properties to the *add_format()* constructor:

```
format = workbook.add_format({'bold': True, 'font_color': 'red'})
```

The object method interface is mainly provided for backward compatibility with *Ex-*

`cel::Writer::XLSX`. The key/value interface has proved to be more flexible in real world programs and is the recommended method for setting format properties.

11.3 Format Colors

Format property colors are specified using a Html style #RRGGBB index:

```
cell_format.set_font_color('#FF0000')
```

For backward compatibility with `Excel::Writer::XLSX` a limited number of color names are supported:

```
cell_format.set_font_color('red')
```

The color names and corresponding #RRGGBB indices are shown below:

Color name	RGB color code
black	#000000
blue	#0000FF
brown	#800000
cyan	#00FFFF
gray	#808080
green	#008000
lime	#00FF00
magenta	#FF00FF
navy	#000080
orange	#FF6600
pink	#FF00FF
purple	#800080
red	#FF0000
silver	#C0C0C0
white	#FFFFFF
yellow	#FFFF00

11.4 Format Defaults

The default Excel 2007+ cell format is Calibri 11 with all other properties off.

In general a format method call without an argument will turn a property on, for example:

```
format1 = workbook.add_format()

format1.set_bold()    # Turns bold on.
format1.set_bold(1)  # Also turns bold on.
```

Since most properties are already off by default it isn't generally required to turn them off. However, it is possible if required:

```
format1.set_bold(0); # Turns bold off.
```

11.5 Modifying Formats

Each unique cell format in an XlsxWriter spreadsheet must have a corresponding Format object. It isn't possible to use a Format with a `write()` method and then redefine it for use at a later stage. This is because a Format is applied to a cell not in its current state but in its final state. Consider the following example:

```
format = workbook.add_format({'bold': True, 'font_color': 'red'})
worksheet.write('A1', 'Cell A1', format)

# Later...
format.set_font_color('green')
worksheet.write('B1', 'Cell B1', format)
```

Cell A1 is assigned a format which initially has the font set to the colour red. However, the colour is subsequently set to green. When Excel displays Cell A1 it will display the final state of the Format which in this case will be the colour green.

WORKING WITH DATES AND TIME

Dates and times in Excel are represented by real numbers, for example “Jan 1 2013 12:00 PM” is represented by the number 41275.5.

The integer part of the number stores the number of days since the epoch and the fractional part stores the percentage of the day.

A date or time in Excel is just like any other number. To display the number as a date you must apply an Excel number format to it. Here are some examples:

```
from xlswriter.workbook import Workbook

workbook = Workbook('date_examples.xlsx')
worksheet = workbook.add_worksheet()

# Widen column A for extra visibility.
worksheet.set_column('A:A', 30)

# A number to convert to a date.
number = 41333.5

# Write it as a number without formatting.
worksheet.write('A1', number)           # 41333.5

format2 = workbook.add_format({'num_format': 'dd/mm/yy'})
worksheet.write('A2', number, format2)  # 28/02/13

format3 = workbook.add_format({'num_format': 'mm/dd/yy'})
worksheet.write('A3', number, format3)  # 02/28/13

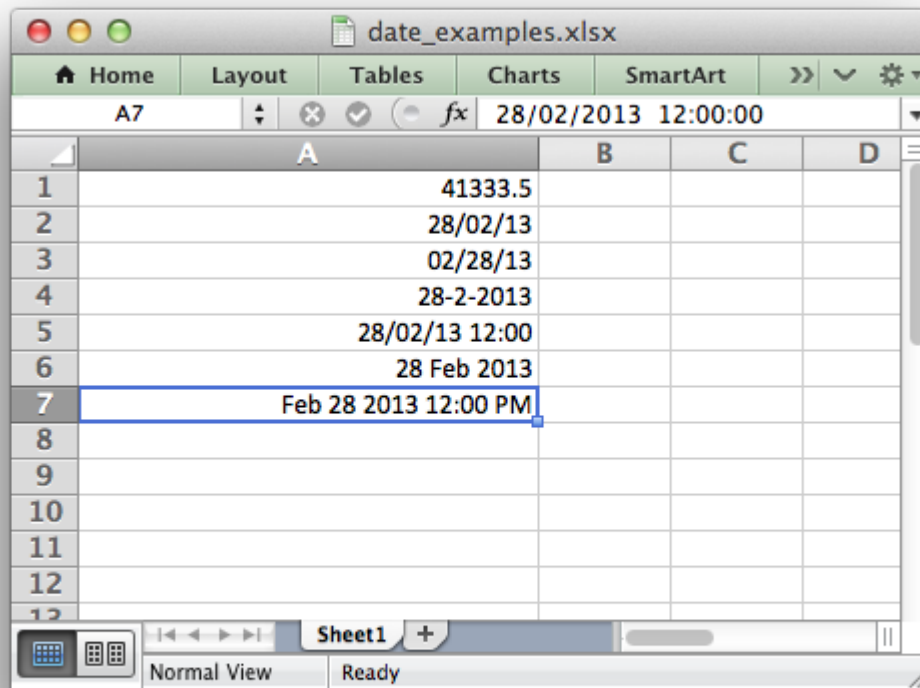
format4 = workbook.add_format({'num_format': 'd-m-yyyy'})
worksheet.write('A4', number, format4)  # 28-2-2013

format5 = workbook.add_format({'num_format': 'dd/mm/yy hh:mm'})
worksheet.write('A5', number, format5)  # 28/02/13 12:00

format6 = workbook.add_format({'num_format': 'd mmm yyyy'})
worksheet.write('A6', number, format6)  # 28 Feb 2013

format7 = workbook.add_format({'num_format': 'mmm d yyyy hh:mm AM/PM'})
```

```
worksheet.write('A7', number, format7)          # Feb 28 2008 12:00 PM
workbook.close()
```



To make working with dates and times a little easier the XlsxWriter module provides a `write_datetime()` method to write dates in standard library `datetime` format.

Specifically it supports datetime objects of type `datetime.datetime`, `datetime.date` and `datetime.time`.

There are many way to create datetime objects, for example the `datetime.datetime.strptime()` method:

```
date_time = datetime.datetime.strptime('2013-01-23', '%Y-%m-%d')
```

See the `datetime` documentation for other date/time creation methods.

As explained above you also need to create and apply a number format to format the date/time:

```
date_format = workbook.add_format({'num_format': 'd mmmm yyyy'})
worksheet.write_datetime('A1', date_time, date_format)
```

```
# Displays "23 January 2013"
```

Here is a longer example that displays the same date in a several different formats:

```

from datetime import datetime
from xlsxwriter.workbook import Workbook

# Create a workbook and add a worksheet.
workbook = Workbook('datetimes.xlsx')
worksheet = workbook.add_worksheet()
bold = workbook.add_format({'bold': True})

# Expand the first columns so that the date is visible.
worksheet.set_column('A:B', 30)

# Write the column headers.
worksheet.write('A1', 'Formatted date', bold)
worksheet.write('B1', 'Format', bold)

# Create a datetime object to use in the examples.
date_time = datetime.strptime('2013-01-23 12:30:05.123',
                              '%Y-%m-%d %H:%M:%S.%f')

# Examples date and time formats.
date_formats = (
    'dd/mm/yy',
    'mm/dd/yy',
    'dd m yy',
    'd mm yy',
    'd mmm yy',
    'd mmmm yy',
    'd mmmm yyy',
    'd mmmm yyyy',
    'dd/mm/yy hh:mm',
    'dd/mm/yy hh:mm:ss',
    'dd/mm/yy hh:mm:ss.000',
    'hh:mm',
    'hh:mm:ss',
    'hh:mm:ss.000',
)

# Start from first row after headers.
row = 1

# Write the same date and time using each of the above formats.
for date_format_str in date_formats:

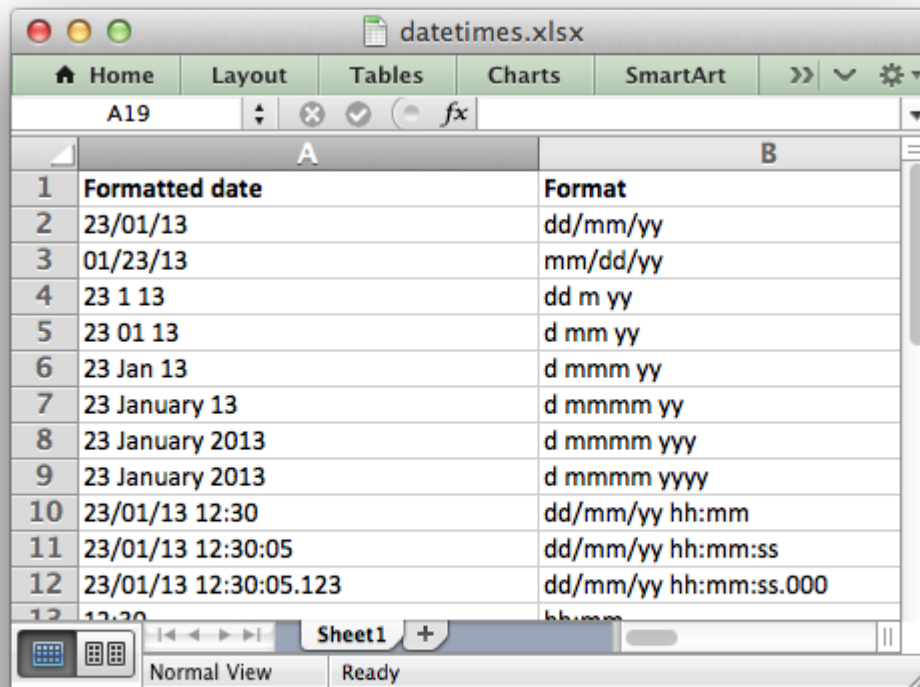
    # Create a format for the date or time.
    date_format = workbook.add_format({'num_format': date_format_str,
                                       'align': 'left'})

    # Write the same date using different formats.
    worksheet.write_datetime(row, 0, date_time, date_format)

    # Also write the format string for comparison.
    worksheet.write_string(row, 1, date_format_str)

```

```
row += 1
```

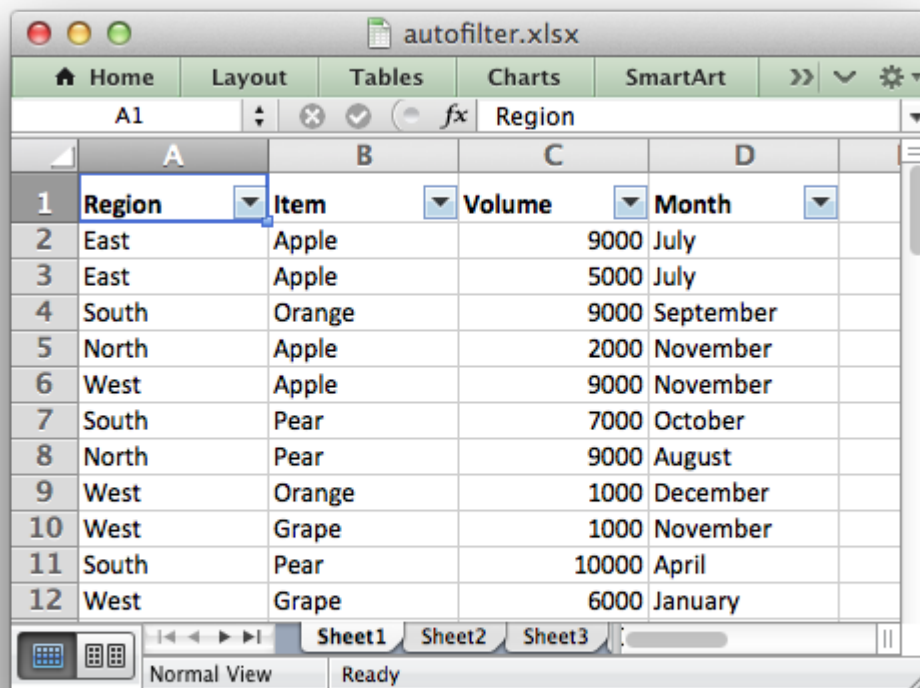


The screenshot shows an Excel spreadsheet with two columns: 'Formatted date' and 'Format'. The rows contain various date and time strings along with their corresponding Excel format codes. The spreadsheet is titled 'datetimes.xlsx' and is on 'Sheet1'.

	Formatted date	Format
1	Formatted date	Format
2	23/01/13	dd/mm/yy
3	01/23/13	mm/dd/yy
4	23 1 13	dd m yy
5	23 01 13	d mm yy
6	23 Jan 13	d mmm yy
7	23 January 13	d mmmm yy
8	23 January 2013	d mmmm yyy
9	23 January 2013	d mmmm yyyy
10	23/01/13 12:30	dd/mm/yy hh:mm
11	23/01/13 12:30:05	dd/mm/yy hh:mm:ss
12	23/01/13 12:30:05.123	dd/mm/yy hh:mm:ss.000
13	12:30	hh:mm

WORKING WITH AUTOFILTERS

An autofilter in Excel is a way of filtering a 2D range of data based on some simple criteria.



13.1 Applying an autofilter

The first step is to apply an autofilter to a cell range in a worksheet using the `autofilter()` method:

```
worksheet.autofilter('A1:D11')
```

As usual you can also use *Row-Column* notation:

```
worksheet.autofilter(0, 0, 10, 3) # Same as above.
```

13.2 Filter data in an autofilter

The `autofilter()` defines the cell range that the filter applies to and creates drop-down selectors in the heading row. In order to filter out data it is necessary to apply some criteria to the columns using either the `filter_column()` or `filter_column_list()` methods.

The `filter_column` method is used to filter columns in a autofilter range based on simple criteria:

```
worksheet.filter_column('A', 'x > 2000')
worksheet.filter_column('B', 'x > 2000 and x < 5000')
```

It isn't sufficient to just specify the filter condition. You must also hide any rows that don't match the filter condition. Rows are hidden using the `set_row()` `hidden` parameter. XlsxWriter cannot filter rows automatically since it isn't part of the file format.

The following is an example of how you might filter a data range to match an autofilter criteria:

```
# Set the autofilter.
worksheet.autofilter('A1:D51')

# Add the filter criteria. The placeholder "Region" in the filter is
# ignored and can be any string that adds clarity to the expression.
worksheet.filter_column(0, 'Region == East')

# Hide the rows that don't match the filter criteria.
row = 1
for row_data in (data):
    region = row_data[0]

    # Check for rows that match the filter.
    if region == 'East':
        # Row matches the filter, display the row as normal.
        pass
    else:
        # We need to hide rows that don't match the filter.
        worksheet.set_row(row, options={'hidden': True})

    worksheet.write_row(row, 0, row_data)

# Move on to the next worksheet row.
row += 1
```

13.3 Setting a filter criteria for a column

The `filter_column()` method can be used to filter columns in a autofilter range based on simple conditions:

```
worksheet.filter_column('A', 'x > 2000')
```

The column parameter can either be a zero indexed column number or a string column name.

The following operators are available for setting the filter criteria:

Operator	Synonyms
<code>==</code>	<code>eq</code> <code>=~ =</code>
<code>!=</code>	<code><></code> <code>ne</code>
<code>></code>	
<code><</code>	
<code>>=</code>	
<code><=</code>	
<code>and</code>	<code>&&</code>
<code>or</code>	<code> </code>

The operator synonyms are just syntactic sugar to make you more comfortable using the expressions. It is important to remember that the expressions will be interpreted by Excel and not by Python.

An expression can comprise a single statement or two statements separated by the `and` and `or` operators. For example:

```
'x < 2000'
'x > 2000'
'x == 2000'
'x > 2000 and x < 5000'
'x == 2000 or x == 5000'
```

Filtering of blank or non-blank data can be achieved by using a value of `Blanks` or `NonBlanks` in the expression:

```
'x == Blanks'
'x == NonBlanks'
```

Excel also allows some simple string matching operations:

```
'x == b*'      # begins with b
'x != b*'      # doesn't begin with b
'x == *b'      # ends with b
'x != *b'      # doesn't end with b
'x == *b*'     # contains b
'x != *b*'     # doesn't contains b
```

You can also use `'*'` to match any character or number and `'?'` to match any single character or number. No other regular expression quantifier is supported by Excel's filters. Excel's regular expression characters can be escaped using `'~'`.

The placeholder variable `x` in the above examples can be replaced by any simple string. The actual placeholder name is ignored internally so the following are all equivalent:

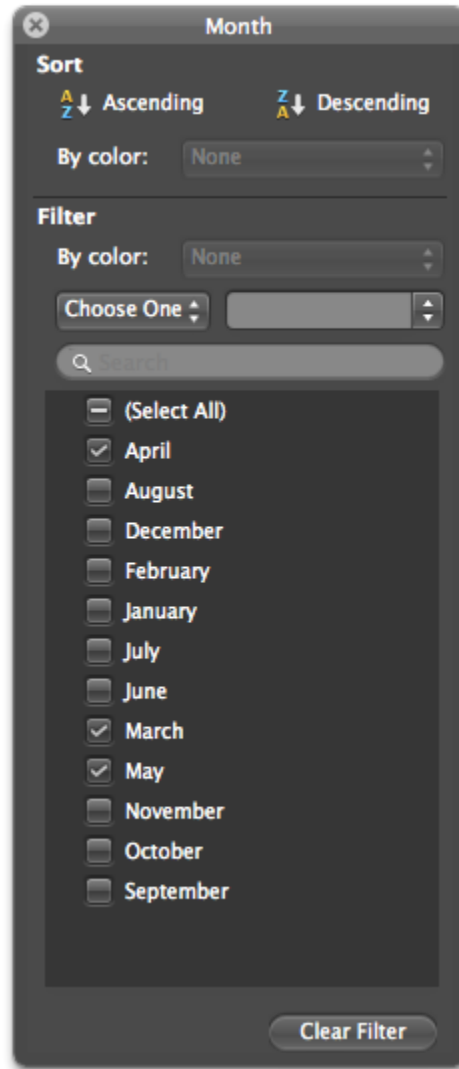
```
'x'      < 2000'  
'col'    < 2000'  
'Price'  < 2000'
```

A filter condition can only be applied to a column in a range specified by the `autofilter()` method.

13.4 Setting a column list filter

Prior to Excel 2007 it was only possible to have either 1 or 2 filter conditions such as the ones shown above in the `filter_column()` method.

Excel 2007 introduced a new list style filter where it is possible to specify 1 or more ‘or’ style criteria. For example if your column contained data for the months of the year you could filter the data based on certain months:



The `filter_column_list()` method can be used to represent these types of filters:

```
worksheet.filter_column_list('A', 'March', 'April', 'May')
```

One or more criteria can be selected:

```
worksheet.filter_column_list('A', 'March')
worksheet.filter_column_list('B', 100, 110, 120, 130)
```

13.5 Example

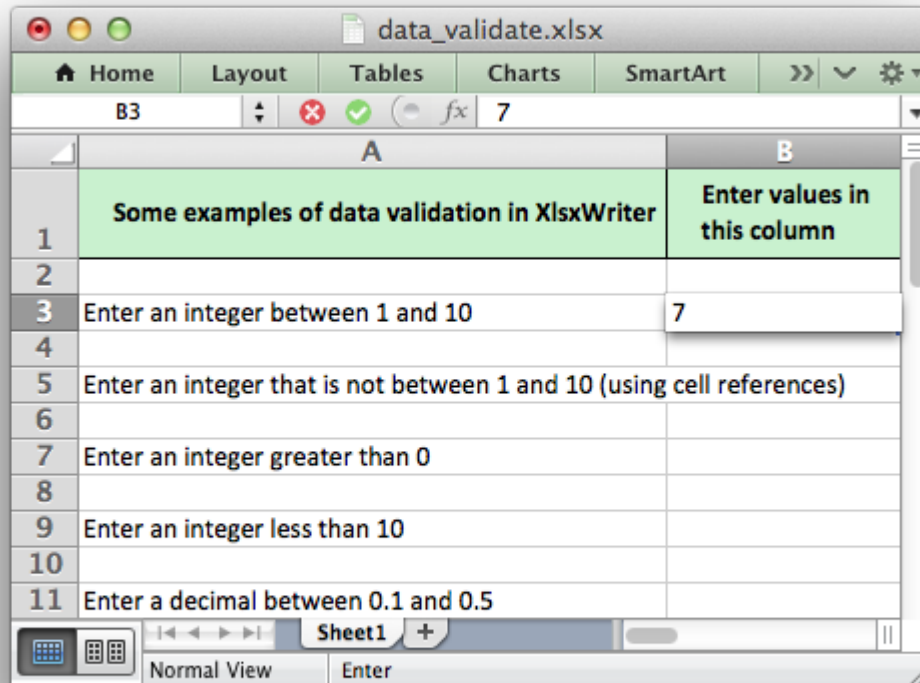
See [Example: Applying Autofilters](#) for a full example of all these features.

WORKING WITH DATA VALIDATION

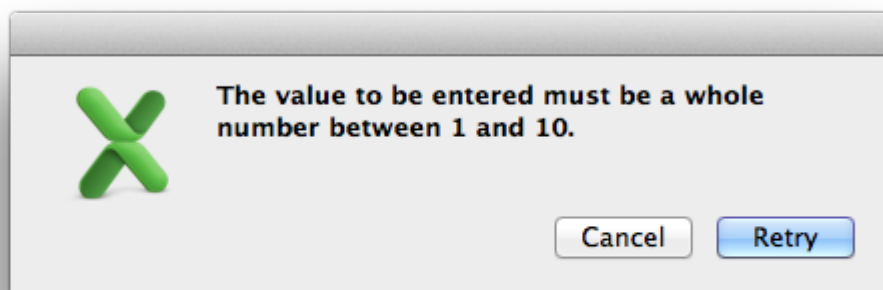
Data validation is a feature of Excel which allows you to restrict the data that a users enters in a cell and to display associated help and warning messages. It also allows you to restrict input to values in a drop down list.

A typical use case might be to restrict data in a cell to integer values in a certain range, to provide a help message to indicate the required value and to issue a warning if the input data doesn't meet the stated criteria. In XlsxWriter we could do that as follows:

```
worksheet.data_validation('B25', {'validate': 'integer',  
                                  'criteria': 'between',  
                                  'minimum': 1,  
                                  'maximum': 100,  
                                  'input_title': 'Enter an integer:',  
                                  'input_message': 'between 1 and 100'})
```



If the user inputs a value that doesn't match the specified criteria an error message is displayed:



For more information on data validation see the Microsoft support article "Description and examples of data validation in Excel": <http://support.microsoft.com/kb/211485>.

The following sections describe how to use the `data_validation()` method and its various options.

14.1 data_validation()

The `data_validation()` method is used to construct an Excel data validation.

The data validation can be applied to a single cell or a range of cells. As usual you can use A1 or Row/Column notation, see [Working with Cell Notation](#).

With Row/Column notation you must specify all four cells in the range: (`first_row`, `first_col`, `last_row`, `last_col`). If you need to refer to a single cell set the *last_* values equal to the *first_* values. With A1 notation you can refer to a single cell or a range of cells:

```
worksheet.data_validation(0, 0, 4, 1, {...})
worksheet.data_validation('B1',      {...})
worksheet.data_validation('C1:E5',    {...})
```

The options parameter in `data_validation()` must be a dictionary containing the parameters that describe the type and style of the data validation. The main parameters are:

validate		
criteria		
value	minimum	source
maximum		
ignore_blank		
dropdown		
input_title		
input_message		
show_input		
error_title		
error_message		
error_type		
show_error		

These parameters are explained in the following sections. Most of the parameters are optional, however, you will generally require the three main options `validate`, `criteria` and `value`:

```
worksheet.data_validation('A1', {'validate': 'integer',
                                'criteria': '>',
                                'value': 100})
```

14.1.1 validate

The `validate` parameter is used to set the type of data that you wish to validate:

```
worksheet.data_validation('A1', {'validate': 'integer',
                                'criteria': '>',
                                'value': 100})
```

It is always required and it has no default value. Allowable values are:

integer
decimal
list
date
time
length
custom
any

- **integer**: restricts the cell to integer values. Excel refers to this as ‘whole number’.
- **decimal**: restricts the cell to decimal values.
- **list**: restricts the cell to a set of user specified values. These can be passed in a Python list or as an Excel cell range. Excel requires that range references are restricted to cells on the same worksheet.
- **date**: restricts the cell to date values specified as a datetime object as shown in [Working with Dates and Time](#).
- **time**: restricts the cell to time values specified as a datetime object as shown in [Working with Dates and Time](#).
- **length**: restricts the cell data based on an integer string length. Excel refers to this as ‘Text length’.
- **custom**: restricts the cell based on an external Excel formula that returns a TRUE/FALSE value.
- **any**: is used to specify that the type of data is unrestricted. This is the same as not applying a data validation. It is only provided for completeness and isn’t used very often in the context of Excel::Writer::XLSX.

14.1.2 criteria

The `criteria` parameter is used to set the criteria by which the data in the cell is validated. It is almost always required except for the `list` and `custom` validate options. It has no default value:

```
worksheet.data_validation('A1', {'validate': 'integer',
                                'criteria': '>',
                                'value': 100})
```

Allowable values are:

between		
not between		
equal to	==	=
not equal to	not =	<>
greater than	>	
less than	<	
greater than or equal to	>=	
less than or equal to	<=	

You can either use Excel's textual description strings, in the first column above, or the more common symbolic alternatives. The following are equivalent:

```
worksheet.data_validation('A1', {'validate': 'integer',
                                'criteria': '>',
                                'value': 100})

worksheet.data_validation('A1', {'validate': 'integer',
                                'criteria': 'greater than',
                                'value': 100})
```

The list and custom validate options don't require a criteria. If you specify one it will be ignored:

```
worksheet.data_validation('B13', {'validate': 'list',
                                'source': ['open', 'high', 'close']})

worksheet.data_validation('B23', {'validate': 'custom',
                                'value': '=AND(F5=50,G5=60)'})
```

14.1.3 value, minimum, source

The value parameter is used to set the limiting value to which the criteria is applied. It is always required and it has no default value. You can also use the synonyms minimum or source to make the validation a little clearer and closer to Excel's description of the parameter:

```
# Use 'value'
worksheet.data_validation('A1', {'validate': 'integer',
                                'criteria': 'greater than',
                                'value': 100})

# Use 'minimum'
worksheet.data_validation('B11', {'validate': 'decimal',
                                'criteria': 'between',
                                'minimum': 0.1,
                                'maximum': 0.5})

# Use 'source'
worksheet.data_validation('B10', {'validate': 'list',
                                'source': '=$E$4:$G$4'})
```

14.1.4 maximum

The maximum parameter is used to set the upper limiting value when the criteria is either 'between' or 'not between':

```
worksheet.data_validation('B11', {'validate': 'decimal',
                                'criteria': 'between',
                                'minimum': 0.1,
                                'maximum': 0.5})
```

14.1.5 ignore_blank

The `ignore_blank` parameter is used to toggle on and off the 'Ignore blank' option in the Excel data validation dialog. When the option is on the data validation is not applied to blank data in the cell. It is on by default:

```
worksheet.data_validation('B5', {'validate': 'integer',  
                                'criteria': 'between',  
                                'minimum': 1,  
                                'maximum': 10,  
                                'ignore_blank': 0,  
                                })
```

14.1.6 dropdown

The `dropdown` parameter is used to toggle on and off the 'In-cell dropdown' option in the Excel data validation dialog. When the option is on a dropdown list will be shown for list validations. It is on by default.

14.1.7 input_title

The `input_title` parameter is used to set the title of the input message that is displayed when a cell is entered. It has no default value and is only displayed if the input message is displayed. See the `input_message` parameter below.

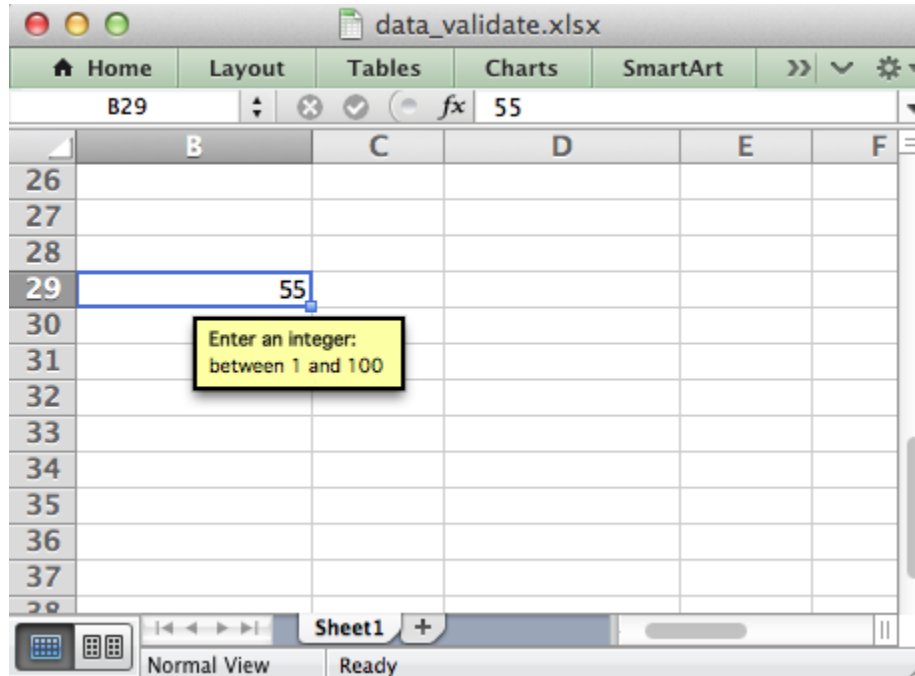
The maximum title length is 32 characters.

14.1.8 input_message

The `input_message` parameter is used to set the input message that is displayed when a cell is entered. It has no default value:

```
worksheet.data_validation('B25', {'validate': 'integer',  
                                'criteria': 'between',  
                                'minimum': 1,  
                                'maximum': 100,  
                                'input_title': 'Enter an integer:',  
                                'input_message': 'between 1 and 100'})
```

The input message generated from the above example is:



The message can be split over several lines using newlines. The maximum message length is 255 characters.

14.1.9 show_input

The `show_input` parameter is used to toggle on and off the ‘Show input message when cell is selected’ option in the Excel data validation dialog. When the option is off an input message is not displayed even if it has been set using `input_message`. It is on by default.

14.1.10 error_title

The `error_title` parameter is used to set the title of the error message that is displayed when the data validation criteria is not met. The default error title is ‘Microsoft Excel’. The maximum title length is 32 characters.

14.1.11 error_message

The `error_message` parameter is used to set the error message that is displayed when a cell is entered. The default error message is “The value you entered is not valid. A user has restricted values that can be entered into the cell.”. A non-default error message can be displayed as follows:

```
worksheet.data_validation('B27', {'validate': 'integer',
                                   'criteria': 'between',
                                   'minimum': 1,
                                   'maximum': 100,
                                   'input_title': 'Enter an integer:',
                                   'input_message': 'between 1 and 100',
```

```
'error_title': 'Input value not valid!',  
'error_message': 'Sorry.')
```

The message can be split over several lines using newlines. The maximum message length is 255 characters.

14.1.12 error_type

The `error_type` parameter is used to specify the type of error dialog that is displayed. There are 3 options:

```
'stop'  
'warning'  
'information'
```

The default is 'stop'.

14.1.13 show_error

The `show_error` parameter is used to toggle on and off the 'Show error alert after invalid data is entered' option in the Excel data validation dialog. When the option is off an error message is not displayed even if it has been set using `error_message`. It is on by default.

14.2 Data Validation Examples

Example 1. Limiting input to an integer greater than a fixed value:

```
worksheet.data_validation('A1', {'validate': 'integer',  
                                'criteria': '>',  
                                'value': 0,  
                                })
```

Example 2. Limiting input to an integer greater than a fixed value where the value is referenced from a cell:

```
worksheet.data_validation('A2', {'validate': 'integer',  
                                'criteria': '>',  
                                'value': '=E3',  
                                })
```

Example 3. Limiting input to a decimal in a fixed range:

```
worksheet.data_validation('A3', {'validate': 'decimal',  
                                'criteria': 'between',  
                                'minimum': 0.1,  
                                'maximum': 0.5,  
                                })
```

Example 4. Limiting input to a value in a dropdown list:

```
worksheet.data_validation('A4', {'validate': 'list',
                                'source': ['open', 'high', 'close'],
                                })
```

Example 5. Limiting input to a value in a dropdown list where the list is specified as a cell range:

```
worksheet.data_validation('A5', {'validate': 'list',
                                'source': '=$E$4:$G$4',
                                })
```

Example 6. Limiting input to a date in a fixed range:

```
from datetime import date

worksheet.data_validation('A6', {'validate': 'date',
                                'criteria': 'between',
                                'minimum': date(2013, 1, 1),
                                'maximum': date(2013, 12, 12),
                                })
```

Example 7. Displaying a message when the cell is selected:

```
worksheet.data_validation('A7', {'validate': 'integer',
                                'criteria': 'between',
                                'minimum': 1,
                                'maximum': 100,
                                'input_title': 'Enter an integer:',
                                'input_message': 'between 1 and 100',
                                })
```

See also [Example: Data Validation and Drop Down Lists](#).

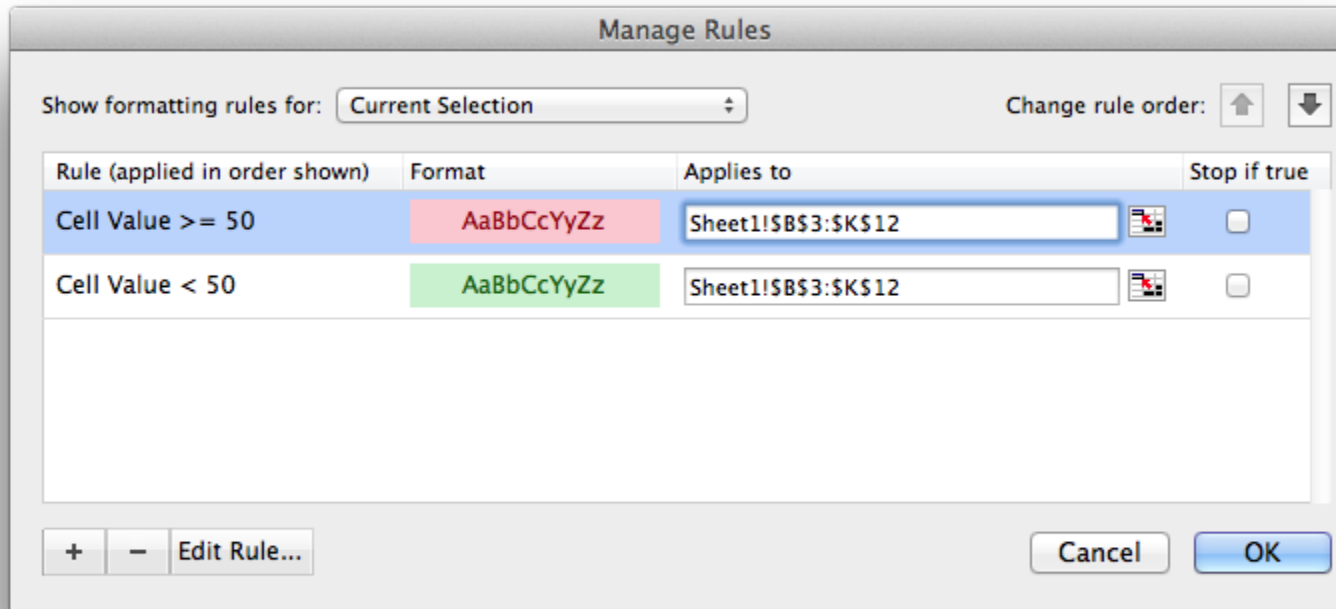
WORKING WITH CONDITIONAL FORMATTING

Conditional formatting is a feature of Excel which allows you to apply a format to a cell or a range of cells based on certain criteria.

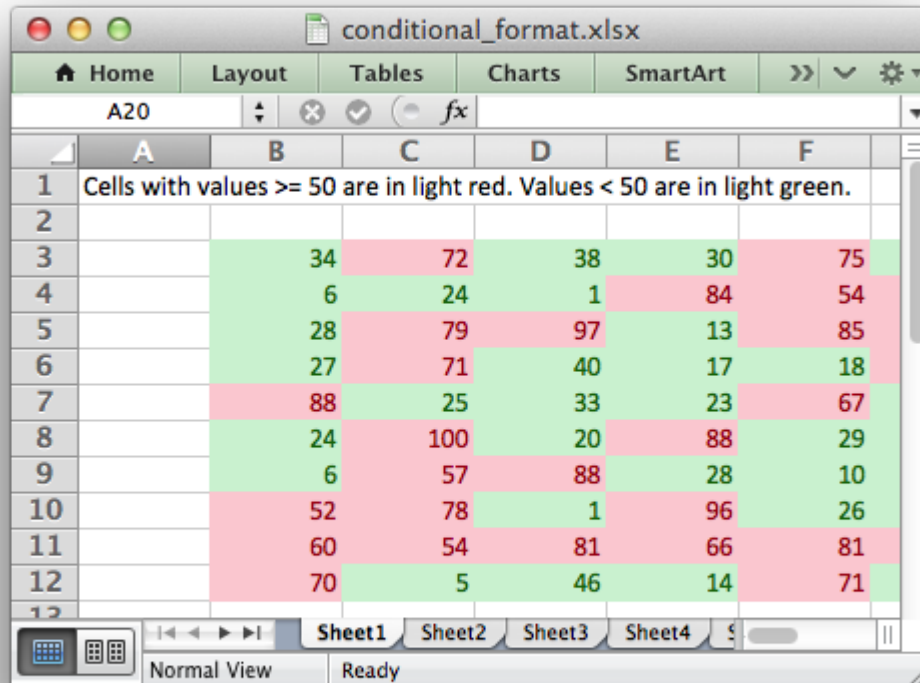
For example the following rules are used to highlight cells in the *conditional_format.py* example:

```
worksheet.conditional_format('B3:K12', {'type':      'cell',  
                                         'criteria': '>=',  
                                         'value':    50,  
                                         'format':   format1})  
  
worksheet.conditional_format('B3:K12', {'type':      'cell',  
                                         'criteria': '<',  
                                         'value':    50,  
                                         'format':   format2})
```

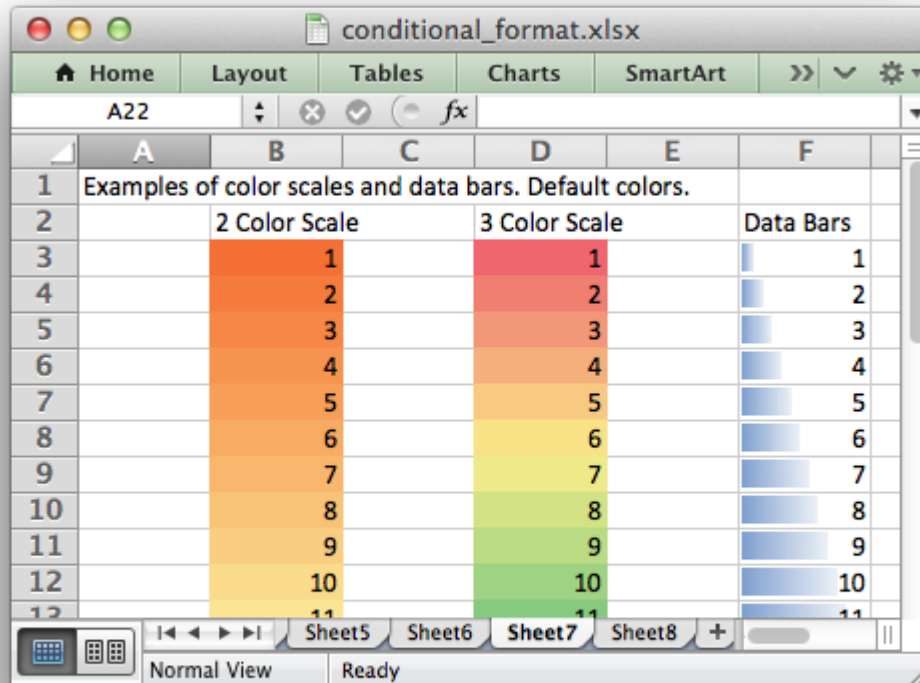
Which gives criteria like this:



And output which looks like this:



It is also possible to create color scales and data bars:



15.1 The conditional_format() method

The `conditional_format()` worksheet method is used to apply formatting based on user defined criteria to an XlsxWriter file.

The conditional format can be applied to a single cell or a range of cells. As usual you can use A1 or Row/Column notation (*Working with Cell Notation*).

With Row/Column notation you must specify all four cells in the range: (first_row, first_col, last_row, last_col). If you need to refer to a single cell set the last_* values equal to the first_* values. With A1 notation you can refer to a single cell or a range of cells:

```
worksheet.conditional_format(0, 0, 4, 1, {...})
worksheet.conditional_format('B1',      {...})
worksheet.conditional_format('C1:E5',    {...})
```

The options parameter in `conditional_format()` must be a dictionary containing the parameters that describe the type and style of the conditional format. The main parameters are:

- type
- format

- criteria
- value
- minimum
- maximum

Other, less commonly used parameters are:

- min_type
- mid_type
- max_type
- min_value
- mid_value
- max_value
- min_color
- mid_color
- max_color
- bar_color
- multi_range

15.2 Conditional Format Options

The conditional format options that can be used with `conditional_format()` are explained in the following sections.

15.2.1 type

The type option is a required parameter and it has no default value. Allowable type values and their associated parameters are:

Type	Parameters
cell	criteria
	value
	minimum
	maximum
date	criteria
	value
	minimum
	maximum
time_period	criteria
Continued on next page	

Table 15.1 – continued from previous page

Type	Parameters
text	criteria
	value
average	criteria
duplicate	(none)
unique	(none)
top	criteria
	value
bottom	criteria
	value
blanks	(none)
no_blanks	(none)
errors	(none)
no_errors	(none)
2_color_scale	min_type
	max_type
	min_value
	max_value
	min_color
	max_color
3_color_scale	min_type
	mid_type
	max_type
	min_value
	mid_value
	max_value
	min_color
	mid_color
	max_color
data_bar	min_type
	max_type
	min_value
	max_value
	bar_color
formula	criteria

All conditional formatting types have an associated *Format* parameter, see below.

15.2.2 type: cell

This is the most common conditional formatting type. It is used when a format is applied to a cell based on a simple criterion.

For example using a single cell and the greater than criteria:

```
worksheet.conditional_format('A1', {'type': 'cell',
                                     'criteria': 'greater than',
                                     'value': 5,
                                     'format': red_format})
```

Or, using a range and the between criteria:

```
worksheet.conditional_format('C1:C4', {'type': 'cell',
                                     'criteria': 'between',
                                     'minimum': 20,
                                     'maximum': 30,
                                     'format': green_format})
```

Other types are shown below, after the other main options.

15.2.3 criteria:

The `criteria` parameter is used to set the criteria by which the cell data will be evaluated. It has no default value. The most common criteria as applied to `{'type': 'cell'}` are:

between		
not between		
equal to	==	=
not equal to	!=	<>
greater than	>	
less than	<	
greater than or equal to	>=	
less than or equal to	<=	

You can either use Excel's textual description strings, in the first column above, or the more common symbolic alternatives shown in the other columns.

Additional criteria which are specific to other conditional format types are shown in the relevant sections below.

15.2.4 value:

The `value` is generally used along with the `criteria` parameter to set the rule by which the cell data will be evaluated:

```
worksheet.conditional_format('A1', {'type': 'cell',
                                     'criteria': 'greater than',
                                     'value': 5,
                                     'format': red_format})
```

The `value` property can also be an cell reference:

```
worksheet.conditional_format('A1', {'type': 'cell',
                                     'criteria': 'greater than',
```

```
'value':      '$C$1',
'format':     red_format})
```

15.2.5 format:

The format parameter is used to specify the format that will be applied to the cell when the conditional formatting criterion is met. The format is created using the `add_format()` method in the same way as cell formats:

```
format1 = workbook.add_format({'bold': 1, 'italic': 1})

worksheet.conditional_format('A1', {'type':      'cell',
                                     'criteria': '>',
                                     'value':     5,
                                     'format':    format1})
```

Note: In Excel, a conditional format is superimposed over the existing cell format and not all cell format properties can be modified. Properties that cannot be modified are font name, font size, superscript and subscript and diagonal borders.

Excel specifies some default formats to be used with conditional formatting. These can be replicated using the following XlsxWriter formats:

```
# Light red fill with dark red text.
format1 = workbook.add_format({'bg_color':    '#FFC7CE',
                                'font_color':  '#9C0006'})

# Light yellow fill with dark yellow text.
format2 = workbook.add_format({'bg_color':    '#FFEB9C',
                                'font_color':  '#9C6500'})

# Green fill with dark green text.
format3 = workbook.add_format({'bg_color':    '#C6EFCE',
                                'font_color':  '#006100'})
```

See also [Working with Formats](#).

15.2.6 minimum:

The minimum parameter is used to set the lower limiting value when the criteria is either 'between' or 'not between':

```
worksheet.conditional_format('A1', {'type':      'cell',
                                     'criteria': 'between',
                                     'minimum':   2,
                                     'maximum':   6,
                                     'format':    format1,
                                     })
```

15.2.7 maximum:

The maximum parameter is used to set the upper limiting value when the criteria is either 'between' or 'not between'. See the previous example.

15.2.8 type: date

The date type is similar the cell type and uses the same criteria and values. However, the value, minimum and maximum properties are specified as a datetime object as shown in [Working with Dates and Time](#):

```
date = datetime.datetime.strptime('2011-01-01', "%Y-%m-%d")

worksheet.conditional_format('A1:A4', {'type':      'date',
                                       'criteria': 'greater than',
                                       'value':     date,
                                       'format':    format1})
```

15.2.9 type: time_period

The time_period type is used to specify Excel's "Dates Occurring" style conditional format:

```
worksheet.conditional_format('A1:A4', {'type':      'time_period',
                                       'criteria': 'yesterday',
                                       'format':    format1})
```

The period is set in the criteria and can have one of the following values:

```
'criteria': 'yesterday',
'criteria': 'today',
'criteria': 'last 7 days',
'criteria': 'last week',
'criteria': 'this week',
'criteria': 'continue week',
'criteria': 'last month',
'criteria': 'this month',
'criteria': 'continue month'
```

15.2.10 type: text

The text type is used to specify Excel's "Specific Text" style conditional format. It is used to do simple string matching using the criteria and value parameters:

```
worksheet.conditional_format('A1:A4', {'type':      'text',
                                       'criteria': 'containing',
                                       'value':     'foo',
                                       'format':    format1})
```

The criteria can have one of the following values:

```
'criteria': 'containing',  
'criteria': 'not containing',  
'criteria': 'begins with',  
'criteria': 'ends with',
```

The value parameter should be a string or single character.

15.2.11 type: average

The average type is used to specify Excel's “Average” style conditional format:

```
worksheet.conditional_format('A1:A4', {'type': 'average',  
                                       'criteria': 'above',  
                                       'format': format1})
```

The type of average for the conditional format range is specified by the criteria:

```
'criteria': 'above',  
'criteria': 'below',  
'criteria': 'equal or above',  
'criteria': 'equal or below',  
'criteria': '1 std dev above',  
'criteria': '1 std dev below',  
'criteria': '2 std dev above',  
'criteria': '2 std dev below',  
'criteria': '3 std dev above',  
'criteria': '3 std dev below',
```

15.2.12 type: duplicate

The duplicate type is used to highlight duplicate cells in a range:

```
worksheet.conditional_format('A1:A4', {'type': 'duplicate',  
                                       'format': format1})
```

15.2.13 type: unique

The unique type is used to highlight unique cells in a range:

```
worksheet.conditional_format('A1:A4', {'type': 'unique',  
                                       'format': format1})
```

15.2.14 type: top

The top type is used to specify the top n values by number or percentage in a range:

```
worksheet.conditional_format('A1:A4', {'type': 'top',
                                         'value': 10,
                                         'format': format1})
```

The criteria can be used to indicate that a percentage condition is required:

```
worksheet.conditional_format('A1:A4', {'type': 'top',
                                         'value': 10,
                                         'criteria': '%',
                                         'format': format1})
```

15.2.15 type: bottom

The bottom type is used to specify the bottom n values by number or percentage in a range.

It takes the same parameters as top, see above.

15.2.16 type: blanks

The blanks type is used to highlight blank cells in a range:

```
worksheet.conditional_format('A1:A4', {'type': 'blanks',
                                         'format': format1})
```

15.2.17 type: no_blanks

The no_blanks type is used to highlight non blank cells in a range:

```
worksheet.conditional_format('A1:A4', {'type': 'no_blanks',
                                         'format': format1})
```

15.2.18 type: errors

The errors type is used to highlight error cells in a range:

```
worksheet.conditional_format('A1:A4', {'type': 'errors',
                                         'format': format1})
```

15.2.19 type: no_errors

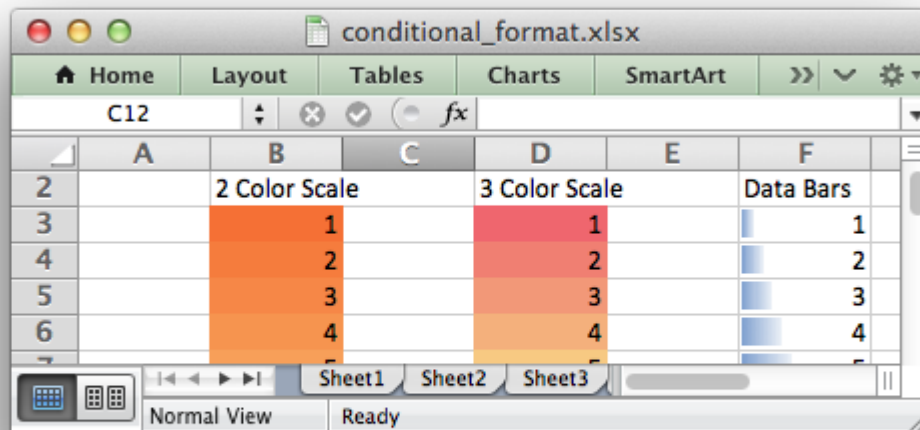
The no_errors type is used to highlight non error cells in a range:

```
worksheet.conditional_format('A1:A4', {'type': 'no_errors',
                                         'format': format1})
```

15.2.20 type: 2_color_scale

The 2_color_scale type is used to specify Excel's "2 Color Scale" style conditional format:

```
worksheet.conditional_format('A1:A12', {'type': '2_color_scale'})
```



This conditional type can be modified with min_type, max_type, min_value, min_value, min_color and max_color, see below.

15.2.21 type: 3_color_scale

The 3_color_scale type is used to specify Excel's "3 Color Scale" style conditional format:

```
worksheet.conditional_format('A1:A12', {'type': '3_color_scale'})
```

This conditional type can be modified with min_type, mid_type, max_type, min_value, mid_value, min_value, min_color, mid_color and max_color, see below.

15.2.22 type: data_bar

The data_bar type is used to specify Excel's "Data Bar" style conditional format:

```
worksheet.conditional_format('A1:A12', {'type': 'data_bar'})
```

This conditional type can be modified with min_type, max_type, min_value, min_value and bar_color, see below.

15.2.23 type: formula

The formula type is used to specify a conditional format based on a user defined formula:

```
worksheet.conditional_format('A1:A4', {'type':      'formula',
                                         'criteria': '=A1>5',
                                         'format':    format1})
```

The formula is specified in the criteria.

15.2.24 min_type:

The min_type and max_type properties are available when the conditional formatting type is 2_color_scale, 3_color_scale or data_bar. The mid_type is available for 3_color_scale. The properties are used as follows:

```
worksheet.conditional_format('A1:A12', {'type':      '2_color_scale',
                                         'min_type':  'percent',
                                         'max_type':  'percent'})
```

The available min/mid/max types are:

```
num
percent
percentile
formula
```

15.2.25 mid_type:

Used for 3_color_scale. Same as min_type, see above.

15.2.26 max_type:

Same as min_type, see above.

15.2.27 min_value:

The min_value and max_value properties are available when the conditional formatting type is 2_color_scale, 3_color_scale or data_bar. The mid_value is available for 3_color_scale. The properties are used as follows:

```
worksheet.conditional_format('A1:A12', {'type':      '2_color_scale',
                                         'min_value': 10,
                                         'max_value': 90})
```

15.2.28 mid_value:

Used for 3_color_scale. Same as min_value, see above.

15.2.29 max_value:

Same as min_value, see above.

15.2.30 min_color:

The min_color and max_color properties are available when the conditional formatting type is 2_color_scale, 3_color_scale or data_bar. The mid_color is available for 3_color_scale. The properties are used as follows:

```
worksheet.conditional_format('A1:A12', {'type': '2_color_scale',  
                                         'min_color': '#C5D9F1',  
                                         'max_color': '#538ED5'})
```

The colour can be a Html style #RRGGBB string or a limited number named colours, see [Format Colors](#).

15.2.31 mid_color:

Used for 3_color_scale. Same as min_color, see above.

15.2.32 max_color:

Same as min_color, see above.

15.2.33 bar_color:

Used for data_bar. Same as min_color, see above.

15.2.34 multi_range:

The multi_range option is used to extend a conditional format over non-contiguous ranges.

It is possible to apply the conditional format to different cell ranges in a worksheet using multiple calls to conditional_format(). However, as a minor optimisation it is also possible in Excel to apply the same conditional format to different non-contiguous cell ranges.

This is replicated in conditional_format() using the multi_range option. The range must contain the primary range for the conditional format and any others separated by spaces.

For example to apply one conditional format to two ranges, 'B3:K6' and 'B9:K12':

```
worksheet.conditional_format('B3:K6', {'type': 'cell',
                                       'criteria': '>=',
                                       'value': 50,
                                       'format': format1,
                                       'multi_range': 'B3:K6 B9:K12'})
```

15.3 Conditional Formatting Examples

Highlight cells greater than an integer value:

```
worksheet.conditional_format('A1:F10', {'type': 'cell',
                                       'criteria': 'greater than',
                                       'value': 5,
                                       'format': format1})
```

Highlight cells greater than a value in a reference cell:

```
worksheet.conditional_format('A1:F10', {'type': 'cell',
                                       'criteria': 'greater than',
                                       'value': 'H1',
                                       'format': format1})
```

Highlight cells more recent (greater) than a certain date:

```
date = datetime.datetime.strptime('2011-01-01', "%Y-%m-%d")

worksheet.conditional_format('A1:F10', {'type': 'date',
                                       'criteria': 'greater than',
                                       'value': date,
                                       'format': format1})
```

Highlight cells with a date in the last seven days:

```
worksheet.conditional_format('A1:F10', {'type': 'time_period',
                                       'criteria': 'last 7 days',
                                       'format': format1})
```

Highlight cells with strings starting with the letter b:

```
worksheet.conditional_format('A1:F10', {'type': 'text',
                                       'criteria': 'begins with',
                                       'value': 'b',
                                       'format': format1})
```

Highlight cells that are 1 standard deviation above the average for the range:

```
worksheet.conditional_format('A1:F10', {'type': 'average',
                                       'format': format1})
```

Highlight duplicate cells in a range:

```
worksheet.conditional_format('A1:F10', {'type': 'duplicate',  
                                         'format': format1})
```

Highlight unique cells in a range:

```
worksheet.conditional_format('A1:F10', {'type': 'unique',  
                                         'format': format1})
```

Highlight the top 10 cells:

```
worksheet.conditional_format('A1:F10', {'type': 'top',  
                                         'value': 10,  
                                         'format': format1})
```

Highlight blank cells:

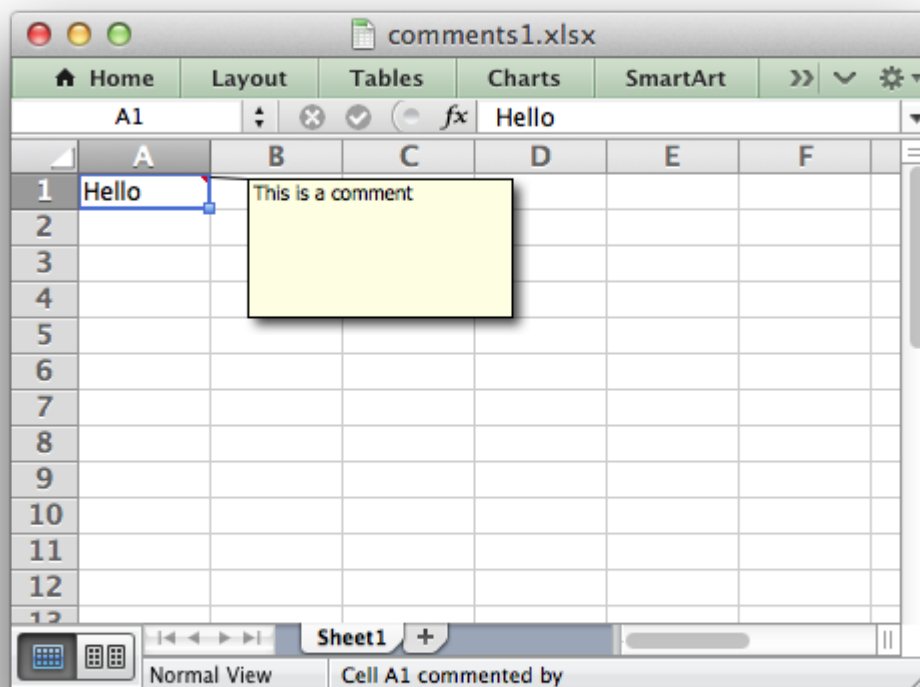
```
worksheet.conditional_format('A1:F10', {'type': 'blanks',  
                                         'format': format1})
```

See also [Example: Conditional Formatting](#).

WORKING WITH CELL COMMENTS

Cell comments are a way of adding notation to cells in Excel. For example:

```
worksheet.write('A1', 'Hello')  
worksheet.write_comment('A1', 'This is a comment')
```



16.1 Setting Comment Properties

The properties of the cell comment can be modified by passing an optional dictionary of key/value pairs to control the format of the comment. For example:

```
worksheet.write_comment('C3', 'Hello', {'x_scale': 1.2, 'y_scale': 0.8})
```

The following options are available:

```
author
visible
x_scale
width
y_scale
height
color
start_cell
start_row
start_col
x_offset
y_offset
```

The options are explained in detail below:

- **author:** This option is used to indicate who is the author of the cell comment. Excel displays the author of the comment in the status bar at the bottom of the worksheet. This is usually of interest in corporate environments where several people might review and provide comments to a workbook:

```
worksheet.write_comment('C3', 'Atonement', {'author': 'Ian McEwan'})
```

The default author for all cell comments in a worksheet can be set using the `set_comments_author()` method:

```
worksheet.set_comments_author('John Smith')
```

- **visible:** This option is used to make a cell comment visible when the worksheet is opened. The default behaviour in Excel is that comments are initially hidden. However, it is also possible in Excel to make individual comments or all comments visible. In XlsxWriter individual comments can be made visible as follows:

```
worksheet.write_comment('C3', 'Hello', {'visible', True})
```

It is possible to make all comments in a worksheet visible using the `show_comments()` worksheet method. Alternatively, if all of the cell comments have been made visible you can hide individual comments:

```
worksheet.write_comment('C3', 'Hello', {'visible', False})
```

- **x_scale:** This option is used to set the width of the cell comment box as a factor of the default width:

```
worksheet.write_comment('C3', 'Hello', {'x_scale': 2 })
worksheet.write_comment('C4', 'Hello', {'x_scale': 4.2})
```

- **width:** This option is used to set the width of the cell comment box explicitly in pixels:

```
worksheet.write_comment('C3', 'Hello', {'width': 200})
```

- **y_scale:** This option is used to set the height of the cell comment box as a factor of the default height:

```
worksheet.write_comment('C3', 'Hello', {'y_scale': 2 })
worksheet.write_comment('C4', 'Hello', {'y_scale': 4.2})
```

- **height:** This option is used to set the height of the cell comment box explicitly in pixels:

```
worksheet.write_comment('C3', 'Hello', {'height': 200})
```

- **color:** This option is used to set the background colour of cell comment box. You can use one of the named colours recognised by XlsxWriter or a Html color. See [Format Colors](#):

```
worksheet.write_comment('C3', 'Hello', {'color': 'green' })
worksheet.write_comment('C4', 'Hello', {'color': '#CCFFCC'})
```

- **start_cell:** This option is used to set the cell in which the comment will appear. By default Excel displays comments one cell to the right and one cell above the cell to which the comment relates. However, you can change this behaviour if you wish. In the following example the comment which would appear by default in cell D2 is moved to E2:

```
worksheet.write_comment('C3', 'Hello', {'start_cell': 'E2'})
```

- **start_row:** This option is used to set the row in which the comment will appear. See the **start_cell** option above. The row is zero indexed:

```
worksheet.write_comment('C3', 'Hello', {'start_row': 0})
```

- **start_col:** This option is used to set the column in which the comment will appear. See the **start_cell** option above. The column is zero indexed:

```
worksheet.write_comment('C3', 'Hello', {'start_col': 4})
```

- **x_offset:** This option is used to change the x offset, in pixels, of a comment within a cell:

```
worksheet.write_comment('C3', comment, {'x_offset': 30})
```

- **y_offset:** This option is used to change the y offset, in pixels, of a comment within a cell:

```
worksheet.write_comment('C3', comment, {'y_offset': 30})
```

You can apply as many of these options as you require. For a working example of these options in use see [Example: Adding Cell Comments to Worksheets \(Advanced\)](#).

Note: Excel only displays offset cell comments when they are displayed as `visible`. Excel does **not** display hidden cells as displaced when you mouse over them. Please note this when using options that adjust the position of the cell comment such as `start_cell`, `start_row`, `start_col`, `x_offset` and `y_offset`.

Note: Row height and comments. If you specify the height of a row that contains a comment then XlsxWriter will adjust the height of the comment to maintain the default or user specified dimensions. However, the height of a row can also be adjusted automatically by Excel if the text wrap property is set or large fonts are used in the cell. This means that the height of the row is unknown to the module at run time and thus the comment box is stretched with the row. Use the `set_row()` method to specify the row height explicitly and avoid this problem. See example 8 of [*Example: Adding Cell Comments to Worksheets \(Advanced\)*](#).

WORKING WITH MEMORY AND PERFORMANCE

The Python `XlsxWriter` module is based on the design of the Perl module `Excel::Writer::XLSX` which in turn is based on an older Perl module called `Spreadsheet::WriteExcel`.

`Spreadsheet::WriteExcel` was written to optimise speed and reduce memory usage. However, these design goals meant that it wasn't easy to implement features that many users requested such as writing formatting and data separately.

As a result `XlsxWriter` (and `Excel::Writer::XLSX`) takes a different design approach and holds a lot more data in memory so that it is functionally more flexible.

The effect of this is that `XlsxWriter` can consume a lot of memory. In addition the extended row and column ranges in Excel 2007+ mean that it is possible to run out of memory creating large files.

Fortunately, this memory usage can be reduced almost completely by using the `Workbook()` 'constant_memory' property:

```
workbook = Workbook(filename, {'constant_memory': True})
```

The optimisation works by flushing each row after a subsequent row is written. In this way the largest amount of data held in memory for a worksheet is the amount of data required to hold a single row of data.

Since each new row flushes the previous row, data must be written in sequential row order when 'constant_memory' mode is on:

```
# With 'constant_memory' you must write data in row by column order.
for row in range(0, row_max):
    for col in range(0, col_max):
        worksheet.write(row, col, some_data)

# With 'constant_memory' this would only write the first column of data.
for col in range(0, col_max):
    for row in range(0, row_max):
        worksheet.write(row, col, some_data)
```

Another optimisation that is used to reduce memory usage is that cell strings aren't stored in an Excel structure call "shared strings" and instead are written "in-line". This is a documented Excel feature that is supported by most spreadsheet applications. One known exception is Apple Numbers for Mac where the string data isn't displayed.

The trade-off when using 'constant_memory' mode is that you won't be able to take advantage of any new features that manipulate cell data after it is written. Currently there are no such features.

For larger files 'constant_memory' mode also gives an increase in execution speed, see below.

17.1 Performance Figures

The performance figures below show execution time and memory usage for worksheets of size N rows x 50 columns with a 50/50 mixture of strings and numbers. The figures are taken from an arbitrary, mid-range, machine. Specific figures will vary from machine to machine but the trends should be the same.

XlsxWriter in normal operation mode: the execution time and memory usage increase more or less linearly with the number of rows:

Rows	Columns	Time (s)	Memory (bytes)
200	50	0.72	2050552
400	50	1.45	4478272
800	50	2.90	8083072
1600	50	5.92	17799424
3200	50	11.83	32218624
6400	50	23.72	64792576
12800	50	47.85	128760832

XlsxWriter in constant_memory mode: the execution time still increases linearly with the number of rows but the memory usage remains small and constant:

Rows	Columns	Time (s)	Memory (bytes)
200	50	0.40	54248
400	50	0.80	54248
800	50	1.60	54248
1600	50	3.19	54248
3200	50	6.29	54248
6400	50	12.74	54248
12800	50	25.34	54248

In the constant_memory mode the performance is also increased. There will be further optimisation in both modes in later releases.

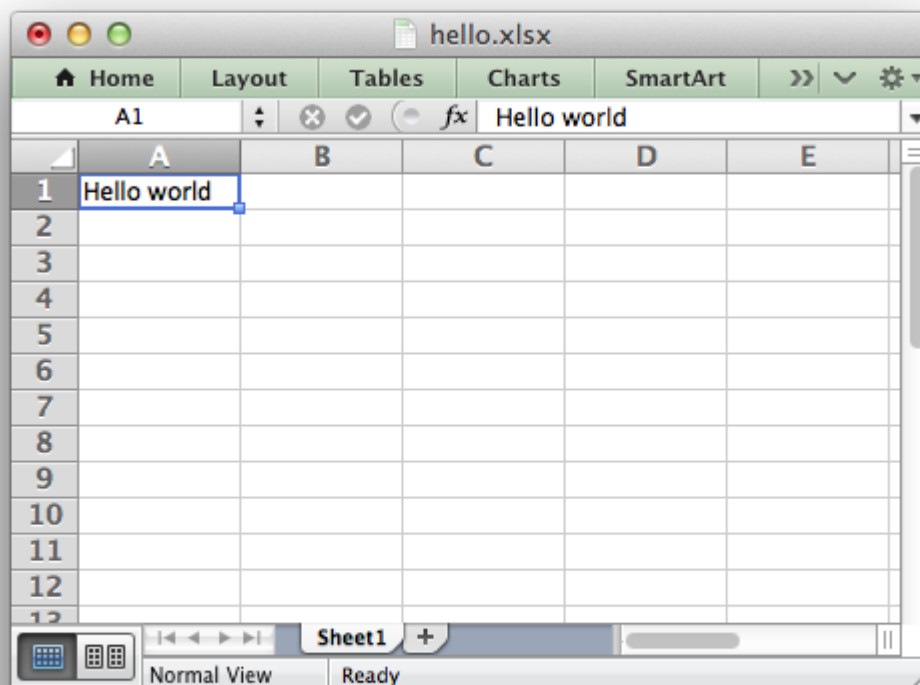
These figures were generated using programs in the dev/performance directory of the XlsxWriter source code.

EXAMPLES

The following are some of the examples included in the [examples](#) directory of the XlsxWriter distribution.

18.1 Example: Hello World

The simplest possible spreadsheet. This is a good place to start to see if the XlsxWriter module is installed correctly.



Code:

```
#####
#
# A hello world spreadsheet using the XlsxWriter Python module.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

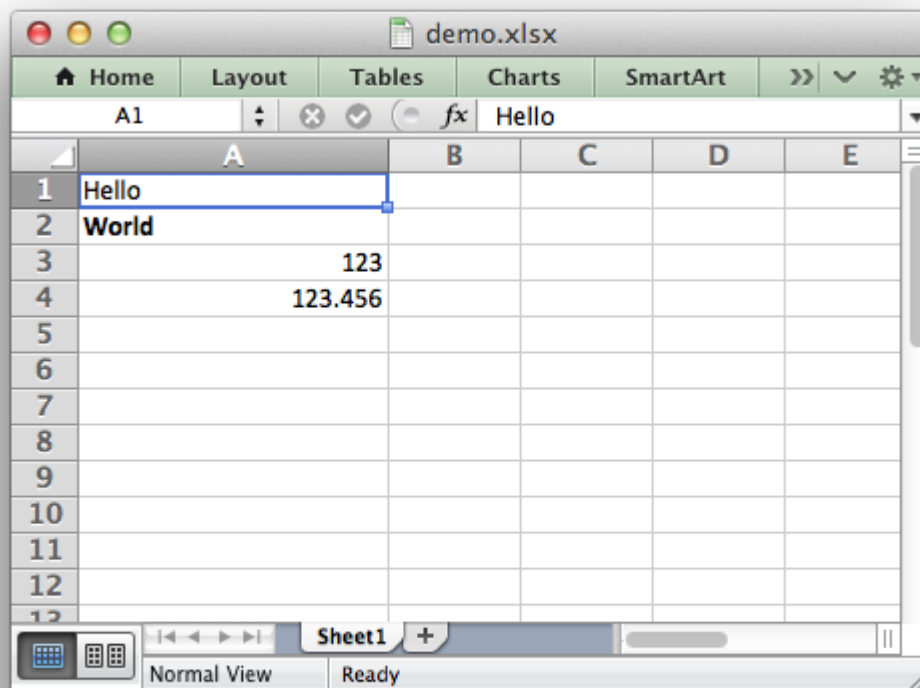
workbook = Workbook('hello_world.xlsx')
worksheet = workbook.add_worksheet()

worksheet.write('A1', 'Hello world')

workbook.close()
```

18.2 Example: Simple Feature Demonstration

This program is an example of writing some of the features of the XlsxWriter module.



Code:

```
#####
#
# A simple example of some of the features of the XlsxWriter Python module.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

# Create an new Excel file and add a worksheet.
workbook = Workbook('demo.xlsx')
worksheet = workbook.add_worksheet()

# Widen the first column to make the text clearer.
worksheet.set_column('A:A', 20)

# Add a bold format to use to highlight cells.
bold = workbook.add_format({'bold': 1})

# Write some simple text.
worksheet.write('A1', 'Hello')

# Text with formatting.
worksheet.write('A2', 'World', bold)

# Write some numbers, with row/column notation.
worksheet.write(2, 0, 123)
worksheet.write(3, 0, 123.456)

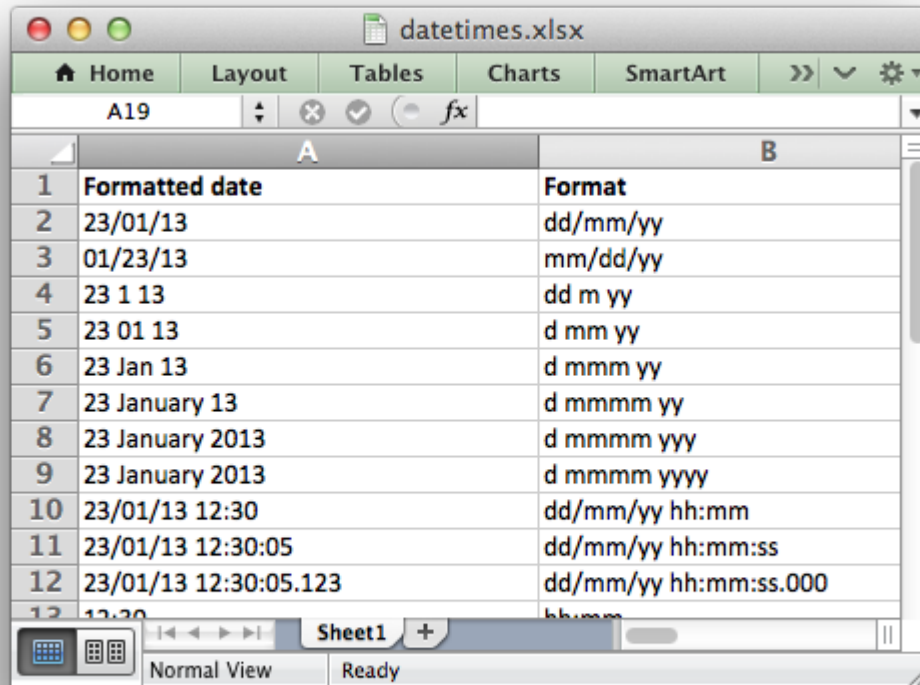
workbook.close()
```

Notes:

- This example includes the use of cell formatting via the *The Format Class*.
- Strings and numbers can be written with the same worksheet `write()` method.
- Data can be written to cells using Row-Column notation or 'A1' style notation, see *Working with Cell Notation*.

18.3 Example: Dates and Times in Excel

This program is an example of writing some of the features of the XlsxWriter module. See the *Working with Dates and Time* section for more details on this example.



	A	B
1	Formatted date	Format
2	23/01/13	dd/mm/yy
3	01/23/13	mm/dd/yy
4	23 1 13	dd m yy
5	23 01 13	d mm yy
6	23 Jan 13	d mmm yy
7	23 January 13	d mmmm yy
8	23 January 2013	d mmmm yyyy
9	23 January 2013	d mmmm yyyy
10	23/01/13 12:30	dd/mm/yy hh:mm
11	23/01/13 12:30:05	dd/mm/yy hh:mm:ss
12	23/01/13 12:30:05.123	dd/mm/yy hh:mm:ss.000

Code:

```
#####
#
# A simple program to write some dates and times to an Excel file
# using the XlsxWriter Python module.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
from datetime import datetime
from xlsxwriter.workbook import Workbook

# Create a workbook and add a worksheet.
workbook = Workbook('datetimes.xlsx')
worksheet = workbook.add_worksheet()
bold = workbook.add_format({'bold': True})

# Expand the first columns so that the date is visible.
worksheet.set_column('A:B', 30)

# Write the column headers.
worksheet.write('A1', 'Formatted date', bold)
worksheet.write('B1', 'Format', bold)

# Create a datetime object to use in the examples.
```

```
date_time = datetime.strptime('2013-01-23 12:30:05.123',
                              '%Y-%m-%d %H:%M:%S.%f')

# Examples date and time formats. In the output file compare how changing
# the format codes change the appearance of the date.
date_formats = (
    'dd/mm/yy',
    'mm/dd/yy',
    'dd m yy',
    'd mm yy',
    'd mmm yy',
    'd mmmm yy',
    'd mmmm yy',
    'd mmmm yyyy',
    'dd/mm/yy hh:mm',
    'dd/mm/yy hh:mm:ss',
    'dd/mm/yy hh:mm:ss.000',
    'hh:mm',
    'hh:mm:ss',
    'hh:mm:ss.000',
)

# Start from first row after headers.
row = 1

# Write the same date and time using each of the above formats.
for date_format_str in date_formats:

    # Create a format for the date or time.
    date_format = workbook.add_format({'num_format': date_format_str,
                                       'align': 'left'})

    # Write the same date using different formats.
    worksheet.write_datetime(row, 0, date_time, date_format)

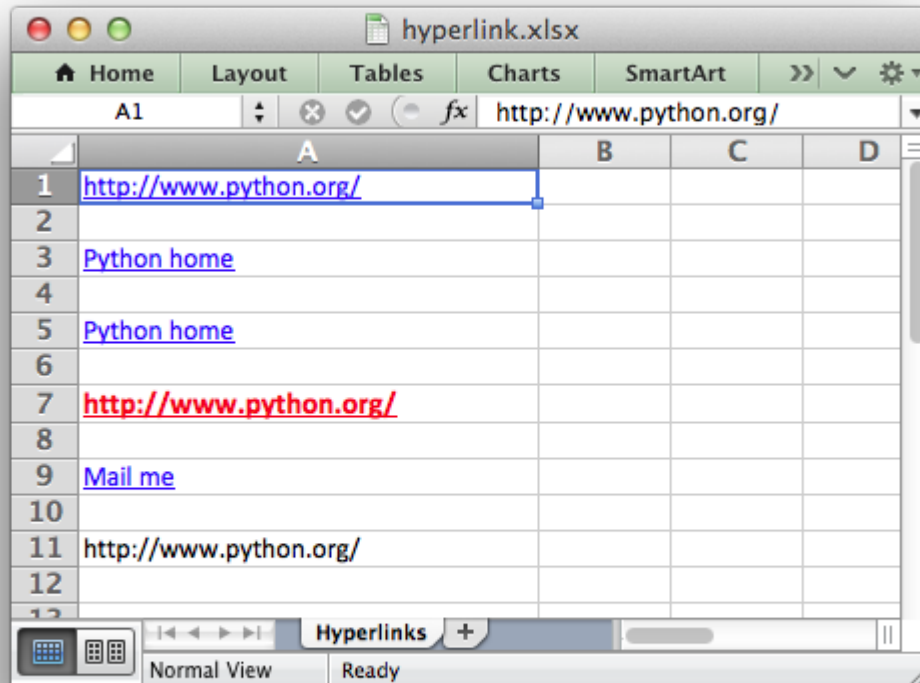
    # Also write the format string for comparison.
    worksheet.write_string(row, 1, date_format_str)

    row += 1

workbook.close()
```

18.4 Example: Adding hyperlinks

This program is an example of writing hyperlinks to a worksheet. See the `write_url()` method for more details.



Code:

```
#####
#
# Example of how to use the XlsxWriter module to write hyperlinks
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

# Create a new workbook and add a worksheet
workbook = Workbook('hyperlink.xlsx')
worksheet = workbook.add_worksheet('Hyperlinks')

# Format the first column
worksheet.set_column('A:A', 30)

# Add the standard url link format.
url_format = workbook.add_format({
    'color': 'blue',
    'underline': 1
})

# Add a sample alternative link format.
red_format = workbook.add_format({
```



```
        'color':      'red',
        'bold':       1,
        'underline':  1,
        'size':       12,
    })

    # Add an alternate description string to the URL.
    string = 'Python home'

    # Add a "tool tip" to the URL.
    tip = 'Get the latest Python news here.'

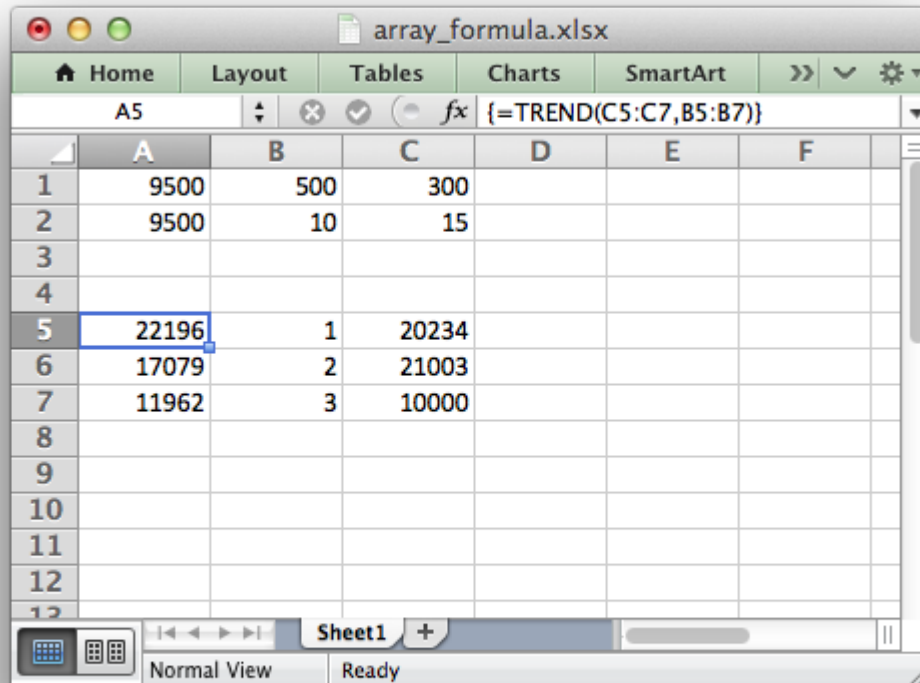
    # Write some hyperlinks
    worksheet.write('A1', 'http://www.python.org/', url_format)
    worksheet.write('A3', 'http://www.python.org/', url_format, string)
    worksheet.write('A5', 'http://www.python.org/', url_format, string, tip)
    worksheet.write('A7', 'http://www.python.org/', red_format)
    worksheet.write('A9', 'mailto:jmcnamaracpan.org', url_format, 'Mail me')

    # Write a URL that isn't a hyperlink
    worksheet.write_string('A11', 'http://www.python.org/')

    workbook.close()
```

18.5 Example: Array formulas

This program is an example of writing array formulas with one or more return values. See the `write_array_formula()` method for more details.



	A	B	C	D	E	F
1	9500	500	300			
2	9500	10	15			
3						
4						
5	22196	1	20234			
6	17079	2	21003			
7	11962	3	10000			
8						
9						
10						
11						
12						
13						

Code:

```
#####
#
# Example of how to use Python and the XlsxWriter module to write
# simple array formulas.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

# Create a new workbook and add a worksheet
workbook = Workbook('array_formula.xlsx')
worksheet = workbook.add_worksheet()

# Write some test data.
worksheet.write('B1', 500)
worksheet.write('B2', 10)
worksheet.write('B5', 1)
worksheet.write('B6', 2)
worksheet.write('B7', 3)
worksheet.write('C1', 300)
worksheet.write('C2', 15)
worksheet.write('C5', 20234)
worksheet.write('C6', 21003)
```

```

worksheet.write('C7', 10000)

# Write an array formula that returns a single value
worksheet.write('A1', '{=SUM(B1:C1*B2:C2)}')

# Same as above but more verbose.
worksheet.write_array_formula('A2:A2', '{=SUM(B1:C1*B2:C2)}')

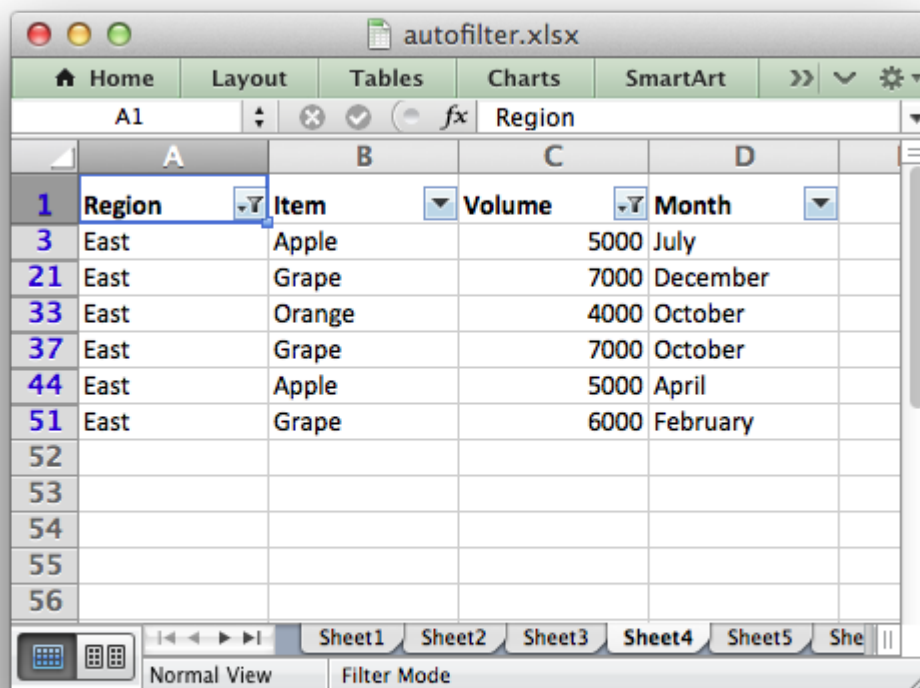
# Write an array formula that returns a range of values
worksheet.write_array_formula('A5:A7', '{=TREND(C5:C7,B5:B7)}')

workbook.close()

```

18.6 Example: Applying Autofilters

This program is an example of using autofilters in a worksheet. See [Working with Autofilters](#) for more details.



Code:

```
#####
#
# An example of how to create autofilters with XlsxWriter.
#
# An autofilter is a way of adding drop down lists to the headers of a 2D
# range of worksheet data. This allows users to filter the data based on
# simple criteria so that some data is shown and some is hidden.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

workbook = Workbook('autofilter.xlsx')

# Add a worksheet for each autofilter example.
worksheet1 = workbook.add_worksheet()
worksheet2 = workbook.add_worksheet()
worksheet3 = workbook.add_worksheet()
worksheet4 = workbook.add_worksheet()
worksheet5 = workbook.add_worksheet()
worksheet6 = workbook.add_worksheet()

# Add a bold format for the headers.
bold = workbook.add_format({'bold': 1})

# Open a text file with autofilter example data.
textfile = open('autofilter_data.txt')

# Read the headers from the first line of the input file.
headers = textfile.readline().strip("\n").split()

# Read the text file and store the field data.
data = []
for line in textfile:
    # Split the input data based on whitespace.
    row_data = line.strip("\n").split()
    data.append(row_data)

# Set up several sheets with the same data.
for worksheet in (workbook.worksheets()):
    # Make the columns wider.
    worksheet.set_column('A:D', 12)
    # Make the header row larger.
    worksheet.set_row(0, 20, bold)
    # Make the headers bold.
    worksheet.write_row('A1', headers)

#####
#
# Example 1. Autofilter without conditions.
```

```
#

# Set the autofilter.
worksheet1.autofilter('A1:D51')

row = 1
for row_data in (data):
    worksheet1.write_row(row, 0, row_data)

    # Move on to the next worksheet row.
    row += 1

#####
#
#
# Example 2. Autofilter with a filter condition in the first column.
#

# Autofilter range using Row-Column notation.
worksheet2.autofilter(0, 0, 50, 3)

# Add filter criteria. The placeholder "Region" in the filter is
# ignored and can be any string that adds clarity to the expression.
worksheet2.filter_column(0, 'Region == East')

# Hide the rows that don't match the filter criteria.
row = 1
for row_data in (data):
    region = row_data[0]

    # Check for rows that match the filter.
    if region == 'East':
        # Row matches the filter, no further action required.
        pass
    else:
        # We need to hide rows that don't match the filter.
        worksheet2.set_row(row, options={'hidden': True})

    worksheet2.write_row(row, 0, row_data)

    # Move on to the next worksheet row.
    row += 1

#####
#
#
# Example 3. Autofilter with a dual filter condition in one of the columns.
#

# Set the autofilter.
worksheet3.autofilter('A1:D51')
```

```
# Add filter criteria.
worksheet3.filter_column('A', 'x == East or x == South')

# Hide the rows that don't match the filter criteria.
row = 1
for row_data in (data):
    region = row_data[0]

    # Check for rows that match the filter.
    if region == 'East' or region == 'South':
        # Row matches the filter, no further action required.
        pass
    else:
        # We need to hide rows that don't match the filter.
        worksheet3.set_row(row, options={'hidden': True})

    worksheet3.write_row(row, 0, row_data)

    # Move on to the next worksheet row.
    row += 1

#####
#
#
# Example 4. Autofilter with filter conditions in two columns.
#

# Set the autofilter.
worksheet4.autofilter('A1:D51')

# Add filter criteria.
worksheet4.filter_column('A', 'x == East')
worksheet4.filter_column('C', 'x > 3000 and x < 8000')

# Hide the rows that don't match the filter criteria.
row = 1
for row_data in (data):
    region = row_data[0]
    volume = int(row_data[2])

    # Check for rows that match the filter.
    if region == 'East' and volume > 3000 and volume < 8000:
        # Row matches the filter, no further action required.
        pass
    else:
        # We need to hide rows that don't match the filter.
        worksheet4.set_row(row, options={'hidden': True})

    worksheet4.write_row(row, 0, row_data)

    # Move on to the next worksheet row.
    row += 1
```

```
#####
#
#
# Example 5. Autofilter with filter for blanks.
#
# Create a blank cell in our test data.

# Set the autofilter.
worksheet5.autofilter('A1:D51')

# Add filter criteria.
worksheet5.filter_column('A', 'x == Blanks')

# Simulate a blank cell in the data.
data[5][0] = ''

# Hide the rows that don't match the filter criteria.
row = 1
for row_data in (data):
    region = row_data[0]

    # Check for rows that match the filter.
    if region == '':
        # Row matches the filter, no further action required.
        pass
    else:
        # We need to hide rows that don't match the filter.
        worksheet5.set_row(row, options={'hidden': True})

    worksheet5.write_row(row, 0, row_data)

    # Move on to the next worksheet row.
    row += 1

#####
#
#
# Example 6. Autofilter with filter for non-blanks.
#

# Set the autofilter.
worksheet6.autofilter('A1:D51')

# Add filter criteria.
worksheet6.filter_column('A', 'x == NonBlanks')

# Hide the rows that don't match the filter criteria.
row = 1
for row_data in (data):
    region = row_data[0]
```

```
# Check for rows that match the filter.
if region != '':
    # Row matches the filter, no further action required.
    pass
else:
    # We need to hide rows that don't match the filter.
    worksheet6.set_row(row, options={'hidden': True})

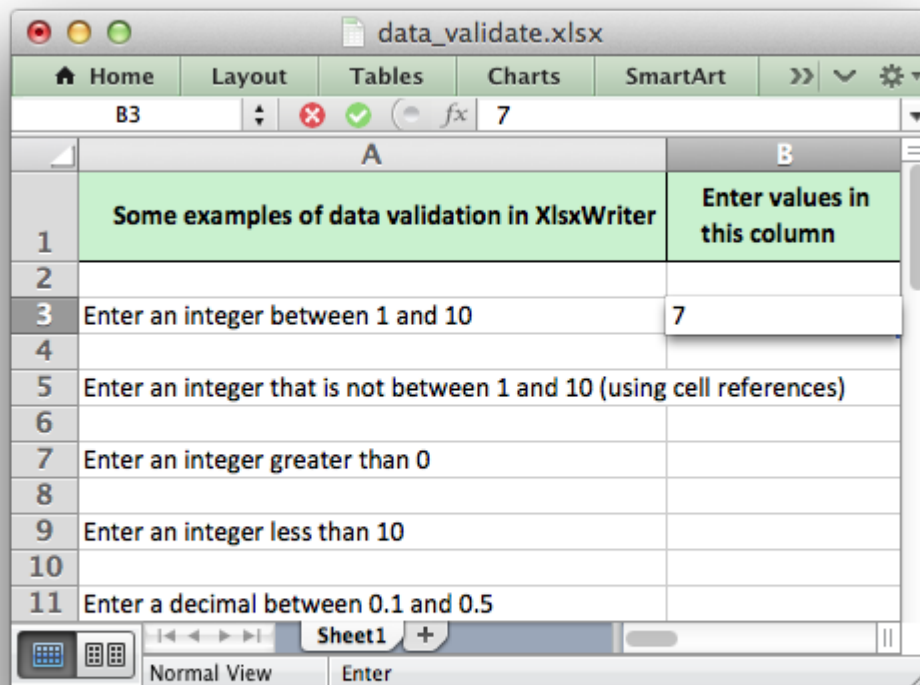
worksheet6.write_row(row, 0, row_data)

# Move on to the next worksheet row.
row += 1
```

```
workbook.close()
```

18.7 Example: Data Validation and Drop Down Lists

Example of how to add data validation and drop down lists to an XlsxWriter file. Data validation is a way of limiting user input to certain ranges or to allow a selection from a drop down list.



Code:


```
#####
#
# Example of how to add data validation and dropdown lists to an
# XlsxWriter file.
#
# Data validation is a feature of Excel which allows you to restrict
# the data that a user enters in a cell and to display help and
# warning messages. It also allows you to restrict input to values in
# a drop down list.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
from datetime import date, time
from xlsxwriter.workbook import Workbook

workbook = Workbook('data_validate.xlsx')
worksheet = workbook.add_worksheet()

# Add a format for the header cells.
header_format = workbook.add_format({
    'border': 1,
    'bg_color': '#C6EFCE',
    'bold': True,
    'text_wrap': True,
    'valign': 'vcenter',
    'indent': 1,
})

# Set up layout of the worksheet.
worksheet.set_column('A:A', 68)
worksheet.set_column('B:B', 15)
worksheet.set_column('D:D', 15)
worksheet.set_row(0, 36)

# Write the header cells and some data that will be used in the examples.
heading1 = 'Some examples of data validation in XlsxWriter'
heading2 = 'Enter values in this column'
heading3 = 'Sample Data'

worksheet.write('A1', heading1, header_format)
worksheet.write('B1', heading2, header_format)
worksheet.write('D1', heading3, header_format)

worksheet.write_row('D3', ['Integers', 1, 10])
worksheet.write_row('D4', ['List data', 'open', 'high', 'close'])
worksheet.write_row('D5', ['Formula', '=AND(F5=50,G5=60)', 50, 60])

# Example 1. Limiting input to an integer in a fixed range.
#
txt = 'Enter an integer between 1 and 10'

worksheet.write('A3', txt)
```

```
worksheet.data_validation('B3', {'validate': 'integer',
                                'criteria': 'between',
                                'minimum': 1,
                                'maximum': 10})

# Example 2. Limiting input to an integer outside a fixed range.
#
txt = 'Enter an integer that is not between 1 and 10 (using cell references)'

worksheet.write('A5', txt)
worksheet.data_validation('B5', {'validate': 'integer',
                                'criteria': 'not between',
                                'minimum': '=E3',
                                'maximum': '=F3'})

# Example 3. Limiting input to an integer greater than a fixed value.
#
txt = 'Enter an integer greater than 0'

worksheet.write('A7', txt)
worksheet.data_validation('B7', {'validate': 'integer',
                                'criteria': '>',
                                'value': 0})

# Example 4. Limiting input to an integer less than a fixed value.
#
txt = 'Enter an integer less than 10'

worksheet.write('A9', txt)
worksheet.data_validation('B9', {'validate': 'integer',
                                'criteria': '<',
                                'value': 10})

# Example 5. Limiting input to a decimal in a fixed range.
#
txt = 'Enter a decimal between 0.1 and 0.5'

worksheet.write('A11', txt)
worksheet.data_validation('B11', {'validate': 'decimal',
                                'criteria': 'between',
                                'minimum': 0.1,
                                'maximum': 0.5})

# Example 6. Limiting input to a value in a dropdown list.
#
txt = 'Select a value from a drop down list'
```

```
worksheet.write('A13', txt)
worksheet.data_validation('B13', {'validate': 'list',
                                  'source': ['open', 'high', 'close']})
```

```
# Example 7. Limiting input to a value in a dropdown list.
```

```
#
```

```
txt = 'Select a value from a drop down list (using a cell range)'
```

```
worksheet.write('A15', txt)
worksheet.data_validation('B10', {'validate': 'list',
                                  'source': '=$E$4:$G$4'})
```

```
# Example 8. Limiting input to a date in a fixed range.
```

```
#
```

```
txt = 'Enter a date between 1/1/2008 and 12/12/2008'
```

```
worksheet.write('A17', txt)
worksheet.data_validation('B17', {'validate': 'date',
                                  'criteria': 'between',
                                  'minimum': date(2013, 1, 1),
                                  'maximum': date(2013, 12, 12)})
```

```
# Example 9. Limiting input to a time in a fixed range.
```

```
#
```

```
txt = 'Enter a time between 6:00 and 12:00'
```

```
worksheet.write('A19', txt)
worksheet.data_validation('B19', {'validate': 'time',
                                  'criteria': 'between',
                                  'minimum': time(6, 0),
                                  'maximum': time(12, 0)})
```

```
# Example 10. Limiting input to a string greater than a fixed length.
```

```
#
```

```
txt = 'Enter a string longer than 3 characters'
```

```
worksheet.write('A21', txt)
worksheet.data_validation('B21', {'validate': 'length',
                                  'criteria': '>',
                                  'value': 3})
```

```
# Example 11. Limiting input based on a formula.
```

```
#
```

```
txt = 'Enter a value if the following is true "=AND(F5=50,G5=60)"'
```

```
worksheet.write('A23', txt)
worksheet.data_validation('B23', {'validate': 'custom',
                                  'value': '=AND(F5=50,G5=60)'})
```

```
# Example 12. Displaying and modifying data validation messages.
#
txt = 'Displays a message when you select the cell'

worksheet.write('A25', txt)
worksheet.data_validation('B25', {'validate': 'integer',
                                   'criteria': 'between',
                                   'minimum': 1,
                                   'maximum': 100,
                                   'input_title': 'Enter an integer:',
                                   'input_message': 'between 1 and 100'})

# Example 13. Displaying and modifying data validation messages.
#
txt = "Display a custom error message when integer isn't between 1 and 100"

worksheet.write('A27', txt)
worksheet.data_validation('B27', {'validate': 'integer',
                                   'criteria': 'between',
                                   'minimum': 1,
                                   'maximum': 100,
                                   'input_title': 'Enter an integer:',
                                   'input_message': 'between 1 and 100',
                                   'error_title': 'Input value is not valid!',
                                   'error_message':
                                   'It should be an integer between 1 and 100'})

# Example 14. Displaying and modifying data validation messages.
#
txt = "Display a custom info message when integer isn't between 1 and 100"

worksheet.write('A29', txt)
worksheet.data_validation('B29', {'validate': 'integer',
                                   'criteria': 'between',
                                   'minimum': 1,
                                   'maximum': 100,
                                   'input_title': 'Enter an integer:',
                                   'input_message': 'between 1 and 100',
                                   'error_title': 'Input value is not valid!',
                                   'error_message':
                                   'It should be an integer between 1 and 100',
                                   'error_type': 'information'})

workbook.close()
```

18.8 Example: Conditional Formatting

Example of how to add conditional formatting to an XlsxWriter file. Conditional formatting allows you to apply a format to a cell or a range of cells based on certain criteria.

	A	B	C	D	E	F
1	Cells with values >= 50 are in light red. Values < 50 are in light green.					
2						
3		34	72	38	30	75
4		6	24	1	84	54
5		28	79	97	13	85
6		27	71	40	17	18
7		88	25	33	23	67
8		24	100	20	88	29
9		6	57	88	28	10
10		52	78	1	96	26
11		60	54	81	66	81
12		70	5	46	14	71

Code:

```
#####
#
# Example of how to add conditional formatting to an XlsxWriter file.
#
# Conditional formatting allows you to apply a format to a cell or a
# range of cells based on certain criteria.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

workbook = Workbook('conditional_format.xlsx')
worksheet1 = workbook.add_worksheet()
worksheet2 = workbook.add_worksheet()
worksheet3 = workbook.add_worksheet()
worksheet4 = workbook.add_worksheet()
worksheet5 = workbook.add_worksheet()
```

```

worksheet6 = workbook.add_worksheet()
worksheet7 = workbook.add_worksheet()
worksheet8 = workbook.add_worksheet()

# Add a format. Light red fill with dark red text.
format1 = workbook.add_format({'bg_color': '#FFC7CE',
                                'font_color': '#9C0006'})

# Add a format. Green fill with dark green text.
format2 = workbook.add_format({'bg_color': '#C6EFCE',
                                'font_color': '#006100'})

# Some sample data to run the conditional formatting against.
data = [
    [34, 72, 38, 30, 75, 48, 75, 66, 84, 86],
    [6, 24, 1, 84, 54, 62, 60, 3, 26, 59],
    [28, 79, 97, 13, 85, 93, 93, 22, 5, 14],
    [27, 71, 40, 17, 18, 79, 90, 93, 29, 47],
    [88, 25, 33, 23, 67, 1, 59, 79, 47, 36],
    [24, 100, 20, 88, 29, 33, 38, 54, 54, 88],
    [6, 57, 88, 28, 10, 26, 37, 7, 41, 48],
    [52, 78, 1, 96, 26, 45, 47, 33, 96, 36],
    [60, 54, 81, 66, 81, 90, 80, 93, 12, 55],
    [70, 5, 46, 14, 71, 19, 66, 36, 41, 21],
]

#####
#
# Example 1.
#
caption = ('Cells with values >= 50 are in light red. '
          'Values < 50 are in light green.')

# Write the data.
worksheet1.write('A1', caption)

for row, row_data in enumerate(data):
    worksheet1.write_row(row + 2, 1, row_data)

# Write a conditional format over a range.
worksheet1.conditional_format('B3:K12', {'type': 'cell',
                                          'criteria': '>= ',
                                          'value': 50,
                                          'format': format1})

# Write another conditional format over the same range.
worksheet1.conditional_format('B3:K12', {'type': 'cell',
                                          'criteria': '< ',
                                          'value': 50,
                                          'format': format2})

```

```
#####
#
# Example 2.
#
caption = ('Values between 30 and 70 are in light red. '
          'Values outside that range are in light green.')

worksheet2.write('A1', caption)

for row, row_data in enumerate(data):
    worksheet2.write_row(row + 2, 1, row_data)

worksheet2.conditional_format('B3:K12', {'type': 'cell',
                                          'criteria': 'between',
                                          'minimum': 30,
                                          'maximum': 70,
                                          'format': format1})

worksheet2.conditional_format('B3:K12', {'type': 'cell',
                                          'criteria': 'not between',
                                          'minimum': 30,
                                          'maximum': 70,
                                          'format': format2})

#####
#
# Example 3.
#
caption = ('Duplicate values are in light red. '
          'Unique values are in light green.')

worksheet3.write('A1', caption)

for row, row_data in enumerate(data):
    worksheet3.write_row(row + 2, 1, row_data)

worksheet3.conditional_format('B3:K12', {'type': 'duplicate',
                                          'format': format1})

worksheet3.conditional_format('B3:K12', {'type': 'unique',
                                          'format': format2})

#####
#
# Example 4.
#
caption = ('Above average values are in light red. '
          'Below average values are in light green.')

worksheet4.write('A1', caption)
```

```

for row, row_data in enumerate(data):
    worksheet4.write_row(row + 2, 1, row_data)

worksheet4.conditional_format('B3:K12', {'type': 'average',
                                          'criteria': 'above',
                                          'format': format1})

worksheet4.conditional_format('B3:K12', {'type': 'average',
                                          'criteria': 'below',
                                          'format': format2})

#####
#
# Example 5.
#
caption = ('Top 10 values are in light red. '
          'Bottom 10 values are in light green.')

worksheet5.write('A1', caption)

for row, row_data in enumerate(data):
    worksheet5.write_row(row + 2, 1, row_data)

worksheet5.conditional_format('B3:K12', {'type': 'top',
                                          'value': '10',
                                          'format': format1})

worksheet5.conditional_format('B3:K12', {'type': 'bottom',
                                          'value': '10',
                                          'format': format2})

#####
#
# Example 6.
#
caption = ('Cells with values >= 50 are in light red. '
          'Values < 50 are in light green. Non-contiguous ranges.')

# Write the data.
worksheet6.write('A1', caption)

for row, row_data in enumerate(data):
    worksheet6.write_row(row + 2, 1, row_data)

# Write a conditional format over a range.
worksheet6.conditional_format('B3:K6', {'type': 'cell',
                                          'criteria': '>=',
                                          'value': 50,
                                          'format': format1,
                                          'multi_range': 'B3:K6 B9:K12'})

```



```
# Write another conditional format over the same range.
worksheet6.conditional_format('B3:K6', {'type': 'cell',
                                         'criteria': '<',
                                         'value': 50,
                                         'format': format2,
                                         'multi_range': 'B3:K6 B9:K12'})

#####
#
# Example 7.
#
caption = 'Examples of color scales and data bars. Default colours.'

data = range(1, 13)

worksheet7.write('A1', caption)

worksheet7.write('B2', "2 Color Scale")
worksheet7.write('D2', "3 Color Scale")
worksheet7.write('F2', "Data Bars")

for row, row_data in enumerate(data):
    worksheet7.write(row + 2, 1, row_data)
    worksheet7.write(row + 2, 3, row_data)
    worksheet7.write(row + 2, 5, row_data)

worksheet7.conditional_format('B3:B14', {'type': '2_color_scale'})
worksheet7.conditional_format('D3:D14', {'type': '3_color_scale'})
worksheet7.conditional_format('F3:F14', {'type': 'data_bar'})

#####
#
# Example 8.
#
caption = 'Examples of color scales and data bars. Modified colours.'

data = range(1, 13)

worksheet8.write('A1', caption)

worksheet8.write('B2', "2 Color Scale")
worksheet8.write('D2', "3 Color Scale")
worksheet8.write('F2', "Data Bars")

for row, row_data in enumerate(data):
    worksheet8.write(row + 2, 1, row_data)
    worksheet8.write(row + 2, 3, row_data)
    worksheet8.write(row + 2, 5, row_data)

worksheet8.conditional_format('B3:B14', {'type': '2_color_scale',
```

```

        'min_color': "#FF0000",
        'max_color': "#00FF00"})

worksheet8.conditional_format('D3:D14', {'type': '3_color_scale',
        'min_color': "#C5D9F1",
        'mid_color': "#8DB4E3",
        'max_color': "#538ED5"})

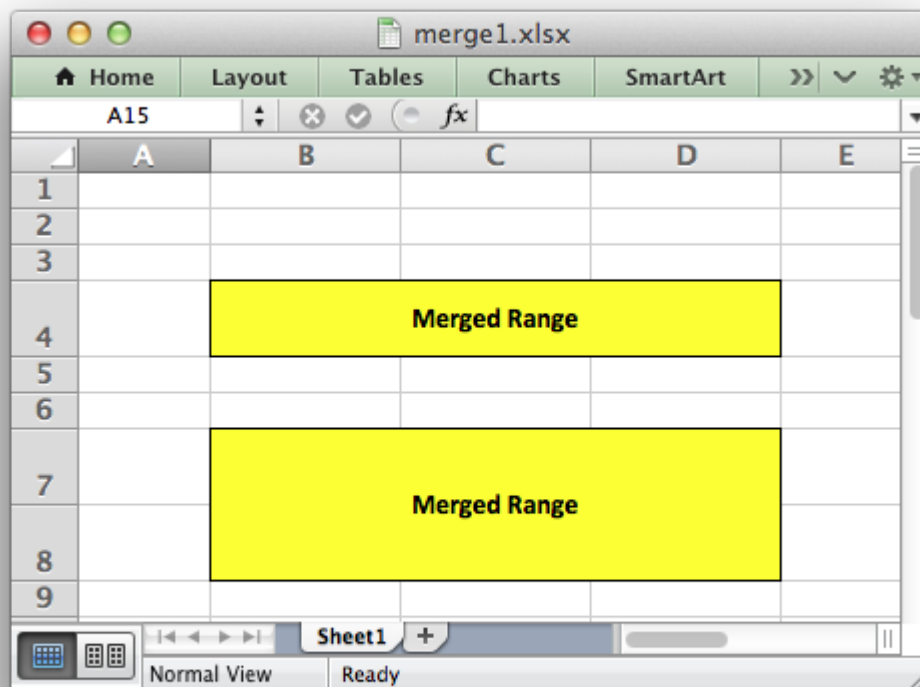
worksheet8.conditional_format('F3:F14', {'type': 'data_bar',
        'bar_color': '#63C384'})

workbook.close()

```

18.9 Example: Merging Cells

This program is an example of merging cells in a worksheet. See the `merge_range()` method for more details.



Code:

```

#####
#

```

```
# A simple example of merging cells with the XlsxWriter Python module.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

# Create an new Excel file and add a worksheet.
workbook = Workbook('merge1.xlsx')
worksheet = workbook.add_worksheet()

# Increase the cell size of the merged cells to highlight the formatting.
worksheet.set_column('B:D', 12)
worksheet.set_row(3, 30)
worksheet.set_row(6, 30)
worksheet.set_row(7, 30)

# Create a format to use in the merged range.
merge_format = workbook.add_format({
    'bold': 1,
    'border': 1,
    'align': 'center',
    'valign': 'vcenter',
    'fg_color': 'yellow'})

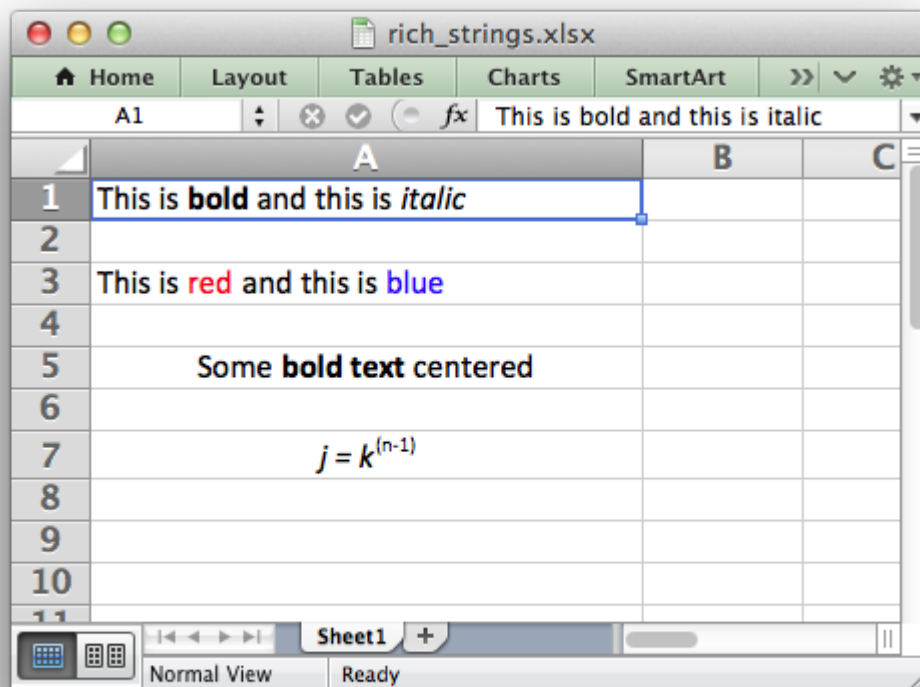
# Merge 3 cells.
worksheet.merge_range('B4:D4', 'Merged Range', merge_format)

# Merge 3 cells over two rows.
worksheet.merge_range('B7:D8', 'Merged Range', merge_format)

workbook.close()
```

18.10 Example: Writing “Rich” strings with multiple formats

This program is an example of writing rich strings with multiple format to a cell in a worksheet. See the `write_rich_string()` method for more details.



Code:

```
#####
#
# An example of using Python and XlsxWriter to write some "rich strings",
# i.e., strings with multiple formats.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

workbook = Workbook('rich_strings.xlsx')
worksheet = workbook.add_worksheet()

worksheet.set_column('A:A', 30)

# Set up some formats to use.
bold = workbook.add_format({'bold': True})
italic = workbook.add_format({'italic': True})
red = workbook.add_format({'color': 'red'})
blue = workbook.add_format({'color': 'blue'})
center = workbook.add_format({'align': 'center'})
superscript = workbook.add_format({'font_script': 1})

# Write some strings with multiple formats.
```

```
worksheet.write_rich_string('A1',
                             'This is ',
                             bold, 'bold',
                             ' and this is ',
                             italic, 'italic')

worksheet.write_rich_string('A3',
                             'This is ',
                             red, 'red',
                             ' and this is ',
                             blue, 'blue')

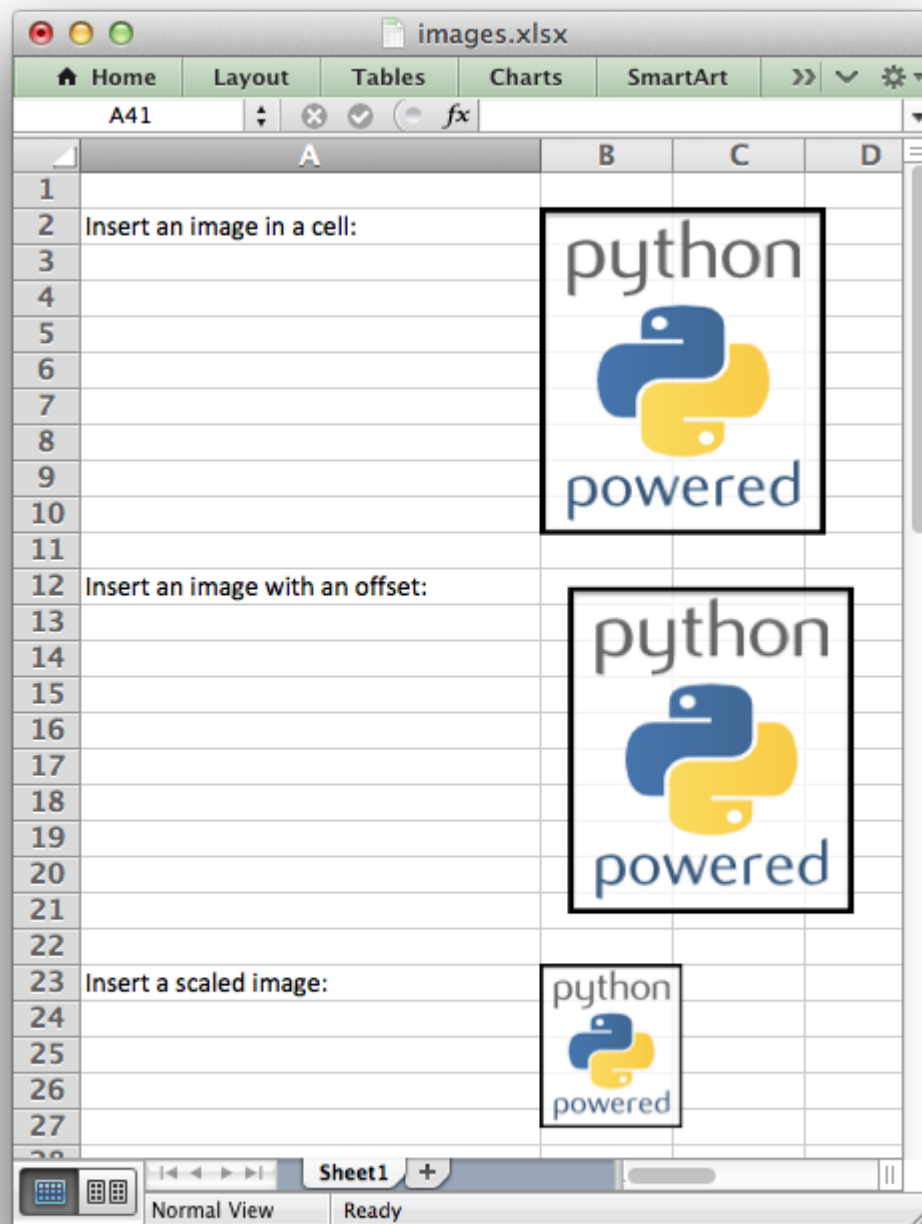
worksheet.write_rich_string('A5',
                             'Some ',
                             bold, 'bold text',
                             ' centered',
                             center)

worksheet.write_rich_string('A7',
                             italic,
                             'j = k',
                             superscript, '(n-1)',
                             center)

workbook.close()
```

18.11 Example: Inserting images into a worksheet

This program is an example of inserting images into a worksheet. See the `insert_image()` method for more details.



Code:

```
#####
#
# An example of inserting images into a worksheet using the XlsxWriter
# Python module.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
```

```
#
from xlsxwriter.workbook import Workbook

# Create an new Excel file and add a worksheet.
workbook = Workbook('images.xlsx')
worksheet = workbook.add_worksheet()

# Widen the first column to make the text clearer.
worksheet.set_column('A:A', 30)

# Insert an image.
worksheet.write('A2', 'Insert an image in a cell:')
worksheet.insert_image('B2', 'python.png')

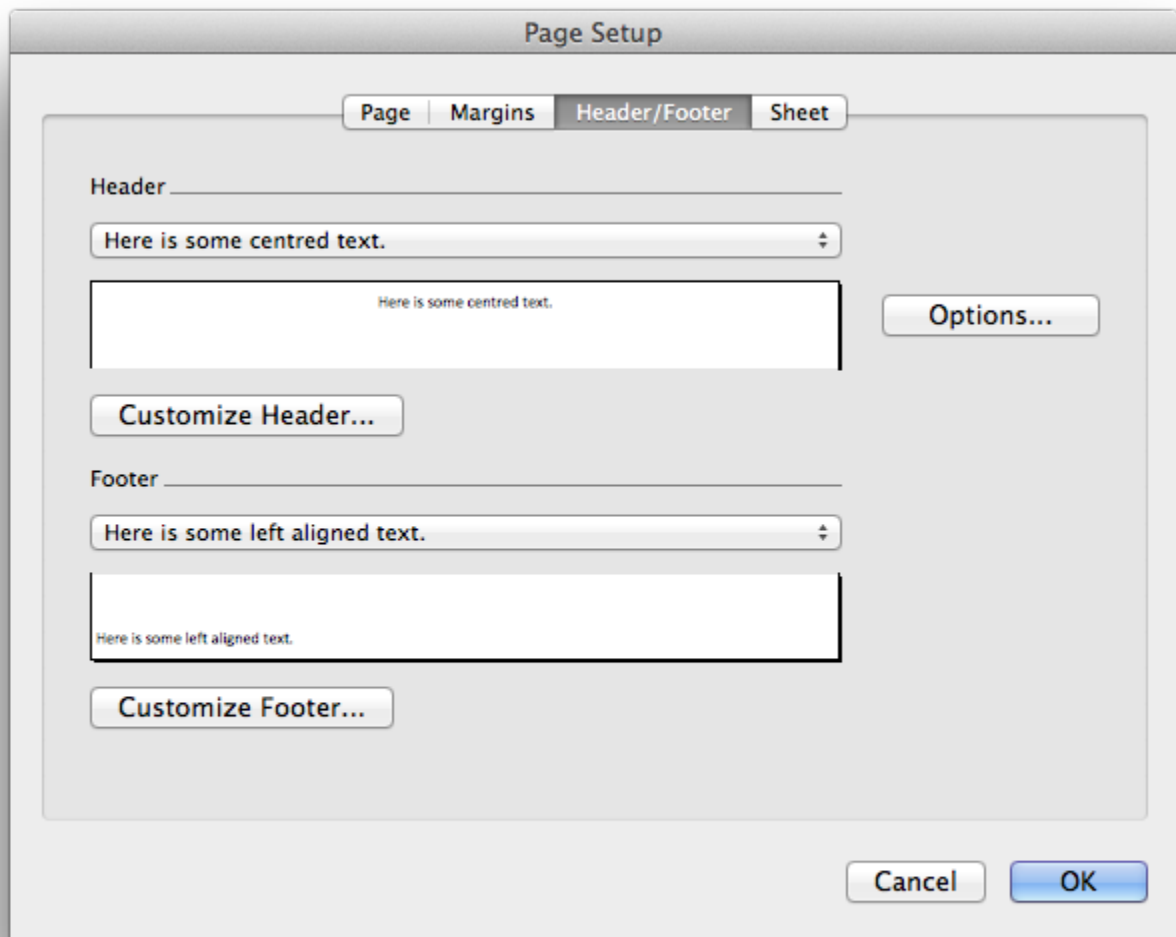
# Insert an image offset in the cell.
worksheet.write('A12', 'Insert an image with an offset:')
worksheet.insert_image('B12', 'python.png', {'x_offset': 15, 'y_offset': 10})

# Insert an image with scaling.
worksheet.write('A23', 'Insert a scaled image:')
worksheet.insert_image('B23', 'python.png', {'x_scale': 0.5, 'y_scale': 0.5})

workbook.close()
```

18.12 Example: Adding Headers and Footers to Worksheets

This program is an example of adding headers and footers to worksheets. See the `set_header()` and `set_footer()` methods for more details.



Code:

```
#####
#
# This program shows several examples of how to set up headers and
# footers with XlsxWriter.
#
# The control characters used in the header/footer strings are:
#
# Control          Category          Description
# =====          =====          =====
# &L                Justification    Left
# &C                Justification    Center
# &R                Justification    Right
#
# &P                Information      Page number
```



```
#      &N      Total number of pages
#      &D      Date
#      &T      Time
#      &F      File name
#      &A      Worksheet name
#
#      &fontsize      Font      Font size
#      &"font,style"      Font name and style
#      &U      Single underline
#      &E      Double underline
#      &S      Strikethrough
#      &X      Superscript
#      &Y      Subscript
#
#      &&      Miscellaneous      Literal ampersand &
#
# See the main XlsxWriter documentation for more information.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

workbook = Workbook('headers_footers.xlsx')
preview = 'Select Print Preview to see the header and footer'

#####
#
# A simple example to start
#
worksheet1 = workbook.add_worksheet('Simple')
header1 = '&CHere is some centred text.'
footer1 = '&LHere is some left aligned text.'

worksheet1.set_header(header1)
worksheet1.set_footer(footer1)

worksheet1.set_column('A:A', 50)
worksheet1.write('A1', preview)

#####
#
# This is an example of some of the header/footer variables.
#
worksheet2 = workbook.add_worksheet('Variables')
header2 = '&LPage &P of &N' + '&CFilename: &F' + '&RSheetname: &A'
footer2 = '&LCurrent date: &D' + '&RCurrent time: &T'

worksheet2.set_header(header2)
worksheet2.set_footer(footer2)

worksheet2.set_column('A:A', 50)
worksheet2.write('A1', preview)
worksheet2.write('A21', 'Next sheet')
```

```

worksheet2.set_h_pagebreaks([20])

#####
#
# This example shows how to use more than one font
#
worksheet3 = workbook.add_worksheet('Mixed fonts')
header3 = '&C&"Courier New,Bold"Hello &"Arial,Italic"World'
footer3 = '&C&"Symbol"e&"Arial" = mc&X2'

worksheet3.set_header(header3)
worksheet3.set_footer(footer3)

worksheet3.set_column('A:A', 50)
worksheet3.write('A1', preview)

#####
#
# Example of line wrapping
#
worksheet4 = workbook.add_worksheet('Word wrap')
header4 = "&CHeading 1\nHeading 2"

worksheet4.set_header(header4)

worksheet4.set_column('A:A', 50)
worksheet4.write('A1', preview)

#####
#
# Example of inserting a literal ampersand &
#
worksheet5 = workbook.add_worksheet('Ampersand')
header5 = '&CCuriouslyer && Curiouslyer - Attorneys at Law'

worksheet5.set_header(header5)

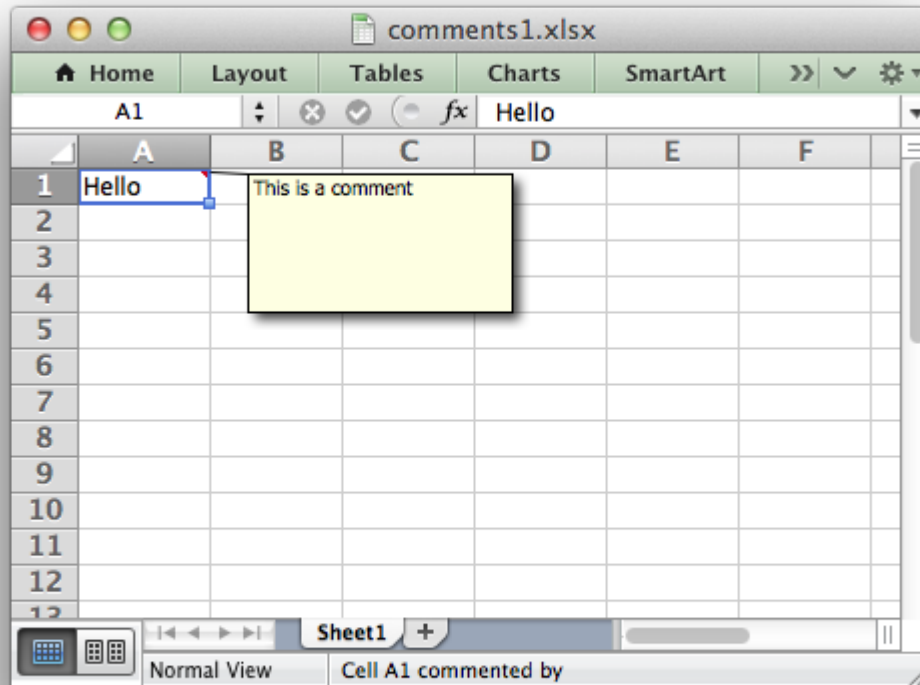
worksheet5.set_column('A:A', 50)
worksheet5.write('A1', preview)

workbook.close()

```

18.13 Example: Adding Cell Comments to Worksheets (Simple)

A simple example of adding cell comments to a worksheet. For more details see [Working with Cell Comments](#).



Code:

```
#####
#
# An example of writing cell comments to a worksheet using Python and
# XlsxWriter.
#
# For more advanced comment options see comments2.py.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

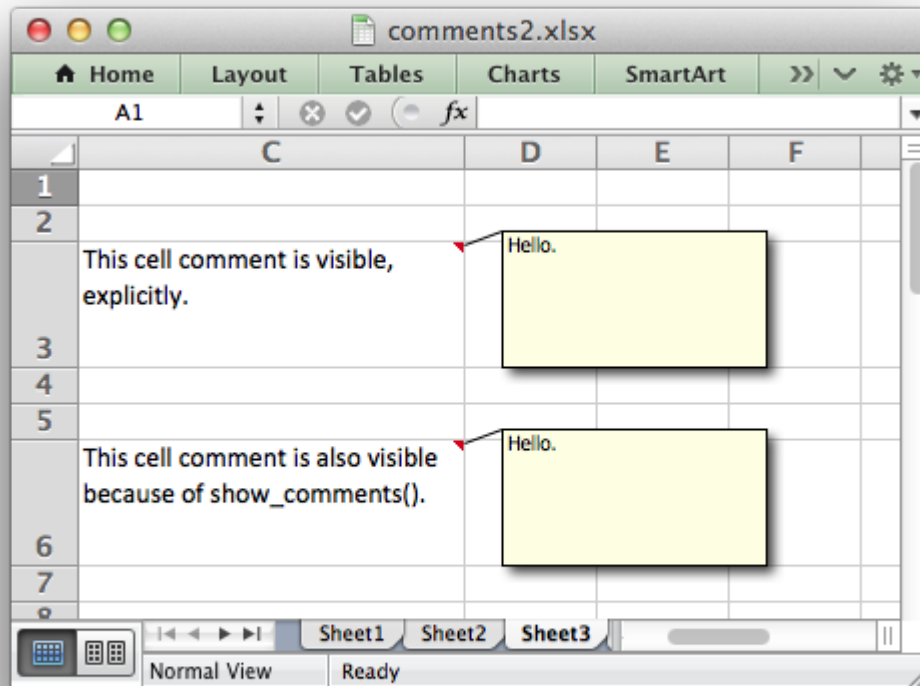
workbook = Workbook('comments1.xlsx')
worksheet = workbook.add_worksheet()

worksheet.write('A1', 'Hello')
worksheet.write_comment('A1', 'This is a comment')

workbook.close()
```

18.14 Example: Adding Cell Comments to Worksheets (Advanced)

Another example of adding cell comments to a worksheet. This example demonstrates most of the available comment formatting options. For more details see [Working with Cell Comments](#).



Code:

```
#####
#
# An example of writing cell comments to a worksheet using Python and
# XlsxWriter.
#
# Each of the worksheets demonstrates different features of cell comments.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

workbook = Workbook('comments2.xlsx')

worksheet1 = workbook.add_worksheet()
worksheet2 = workbook.add_worksheet()
worksheet3 = workbook.add_worksheet()
worksheet4 = workbook.add_worksheet()
```

```

worksheet5 = workbook.add_worksheet()
worksheet6 = workbook.add_worksheet()
worksheet7 = workbook.add_worksheet()
worksheet8 = workbook.add_worksheet()

text_wrap = workbook.add_format({'text_wrap': 1, 'valign': 'top'})

#####
#
# Example 1. Demonstrates a simple cell comments without formatting.
#           comments.
#
# Set up some formatting.
worksheet1.set_column('C:C', 25)
worksheet1.set_row(2, 50)
worksheet1.set_row(5, 50)

# Simple ASCII string.
cell_text = 'Hold the mouse over this cell to see the comment.'

comment = 'This is a comment.'

worksheet1.write('C3', cell_text, text_wrap)
worksheet1.write_comment('C3', comment)

#####
#
# Example 2. Demonstrates visible and hidden comments.
#
# Set up some formatting.
worksheet2.set_column('C:C', 25)
worksheet2.set_row(2, 50)
worksheet2.set_row(5, 50)

cell_text = 'This cell comment is visible.'
comment = 'Hello.'

worksheet2.write('C3', cell_text, text_wrap)
worksheet2.write_comment('C3', comment, {'visible': True})

cell_text = "This cell comment isn't visible (the default)."

worksheet2.write('C6', cell_text, text_wrap)
worksheet2.write_comment('C6', comment)

#####
#
# Example 3. Demonstrates visible and hidden comments set at the worksheet

```

```

#         level.
#

# Set up some formatting.
worksheet3.set_column('C:C', 25)
worksheet3.set_row(2, 50)
worksheet3.set_row(5, 50)
worksheet3.set_row(8, 50)

# Make all comments on the worksheet visible.
worksheet3.show_comments()

cell_text = 'This cell comment is visible, explicitly.'
comment = 'Hello.'

worksheet3.write('C3', cell_text, text_wrap)
worksheet3.write_comment('C3', comment, {'visible': 1})

cell_text = 'This cell comment is also visible because of show_comments().'

worksheet3.write('C6', cell_text, text_wrap)
worksheet3.write_comment('C6', comment)

cell_text = 'However, we can still override it locally.'

worksheet3.write('C9', cell_text, text_wrap)
worksheet3.write_comment('C9', comment, {'visible': False})

#####
#
# Example 4. Demonstrates changes to the comment box dimensions.
#

# Set up some formatting.
worksheet4.set_column('C:C', 25)
worksheet4.set_row(2, 50)
worksheet4.set_row(5, 50)
worksheet4.set_row(8, 50)
worksheet4.set_row(15, 50)

worksheet4.show_comments()

cell_text = 'This cell comment is default size.'
comment = 'Hello.'

worksheet4.write('C3', cell_text, text_wrap)
worksheet4.write_comment('C3', comment)

cell_text = 'This cell comment is twice as wide.'

worksheet4.write('C6', cell_text, text_wrap)
worksheet4.write_comment('C6', comment, {'x_scale': 2})

```

```

cell_text = 'This cell comment is twice as high.'

worksheet4.write('C9', cell_text, text_wrap)
worksheet4.write_comment('C9', comment, {'y_scale': 2})

cell_text = 'This cell comment is scaled in both directions.'

worksheet4.write('C16', cell_text, text_wrap)
worksheet4.write_comment('C16', comment, {'x_scale': 1.2, 'y_scale': 0.8})

cell_text = 'This cell comment has width and height specified in pixels.'

worksheet4.write('C19', cell_text, text_wrap)
worksheet4.write_comment('C19', comment, {'width': 200, 'height': 20})

#####
#
# Example 5. Demonstrates changes to the cell comment position.
#
worksheet5.set_column('C:C', 25)
worksheet5.set_row(2, 50)
worksheet5.set_row(5, 50)
worksheet5.set_row(8, 50)
worksheet5.set_row(11, 50)

worksheet5.show_comments()

cell_text = 'This cell comment is in the default position.'
comment = 'Hello.'

worksheet5.write('C3', cell_text, text_wrap)
worksheet5.write_comment('C3', comment)

cell_text = 'This cell comment has been moved to another cell.'

worksheet5.write('C6', cell_text, text_wrap)
worksheet5.write_comment('C6', comment, {'start_cell': 'E4'})

cell_text = 'This cell comment has been moved to another cell.'

worksheet5.write('C9', cell_text, text_wrap)
worksheet5.write_comment('C9', comment, {'start_row': 8, 'start_col': 4})

cell_text = 'This cell comment has been shifted within its default cell.'

worksheet5.write('C12', cell_text, text_wrap)
worksheet5.write_comment('C12', comment, {'x_offset': 30, 'y_offset': 12})

#####
#
# Example 6. Demonstrates changes to the comment background colour.

```

```
#
worksheet6.set_column('C:C', 25)
worksheet6.set_row(2, 50)
worksheet6.set_row(5, 50)
worksheet6.set_row(8, 50)

worksheet6.show_comments()

cell_text = 'This cell comment has a different colour.'
comment = 'Hello.'

worksheet6.write('C3', cell_text, text_wrap)
worksheet6.write_comment('C3', comment, {'color': 'green'})

cell_text = 'This cell comment has the default colour.'

worksheet6.write('C6', cell_text, text_wrap)
worksheet6.write_comment('C6', comment)

cell_text = 'This cell comment has a different colour.'

worksheet6.write('C9', cell_text, text_wrap)
worksheet6.write_comment('C9', comment, {'color': '#CCFFCC'})

#####
#
# Example 7. Demonstrates how to set the cell comment author.
#
worksheet7.set_column('C:C', 30)
worksheet7.set_row(2, 50)
worksheet7.set_row(5, 50)
worksheet7.set_row(8, 50)

author = ''
cell = 'C3'

cell_text = ("Move the mouse over this cell and you will see 'Cell commented "
            "by (blank)' in the status bar at the bottom")

comment = 'Hello.'

worksheet7.write(cell, cell_text, text_wrap)
worksheet7.write_comment(cell, comment)

author = 'Python'
cell = 'C6'
cell_text = ("Move the mouse over this cell and you will see 'Cell commented "
            "by Python' in the status bar at the bottom")

worksheet7.write(cell, cell_text, text_wrap)
worksheet7.write_comment(cell, comment, {'author': author})
```



```
#####
#
# Example 8. Demonstrates the need to explicitly set the row height.
#
# Set up some formatting.
worksheet8.set_column('C:C', 25)
worksheet8.set_row(2, 80)

worksheet8.show_comments()

cell_text = ('The height of this row has been adjusted explicitly using '
             'set_row(). The size of the comment box is adjusted '
             'accordingly by XlsxWriter.')

comment = 'Hello.'

worksheet8.write('C3', cell_text, text_wrap)
worksheet8.write_comment('C3', comment)

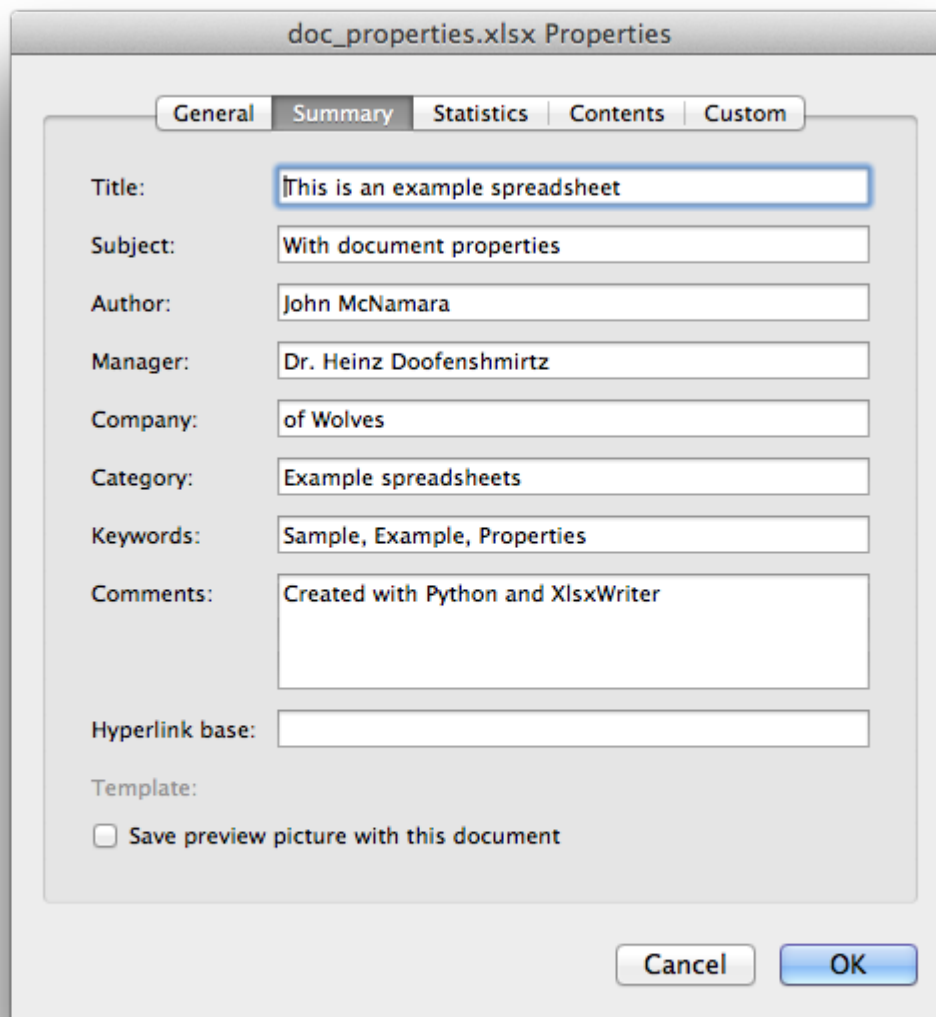
cell_text = ('The height of this row has been adjusted by Excel due to the '
             'text wrap property being set. Unfortunately this means that '
             'the height of the row is unknown to XlsxWriter at run time '
             'and thus the comment box is stretched as well.\n\n'
             'Use set_row() to specify the row height explicitly to avoid '
             'this problem.')

worksheet8.write('C6', cell_text, text_wrap)
worksheet8.write_comment('C6', comment)

workbook.close()
```

18.15 Example: Setting Document Properties

This program is an example setting document properties. See the `set_properties()` workbook method for more details.



Code:

```
#####
#
# An example of adding document properties to a XlsxWriter file.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

workbook = Workbook('doc_properties.xlsx')
worksheet = workbook.add_worksheet()

workbook.set_properties({
```

```
'title':    'This is an example spreadsheet',
'subject':  'With document properties',
'author':   'John McNamara',
'manager':  'Dr. Heinz Doofenshmirtz',
'company':  'of Wolves',
'category': 'Example spreadsheets',
'keywords': 'Sample, Example, Properties',
'comments': 'Created with Python and XlsxWriter',
'status':   'Quo',
})

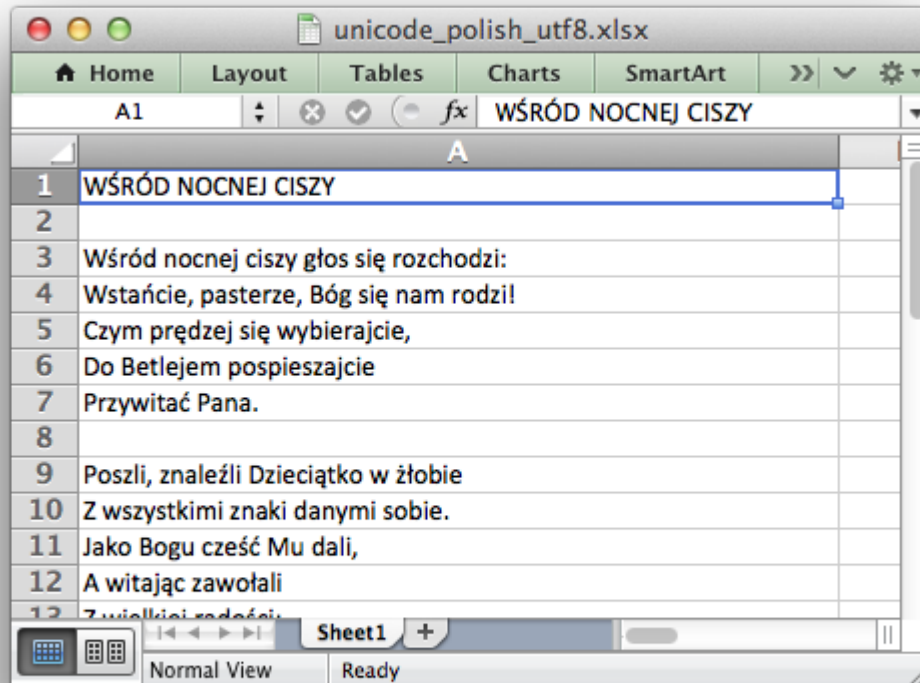
worksheet.set_column('A:A', 70)
worksheet.write('A1', "Select 'Workbook Properties' to see properties.")
```

18.16 Example: Unicode - Polish in UTF-8

This program is an example of reading in data from a UTF-8 encoded text file and converting it to a worksheet.

The main trick is to ensure that the data read in is converted to UTF-8 within the Python program. The XlsxWriter module will then take care of writing the encoding to the Excel file.

The encoding of the input data shouldn't matter once it can be converted to UTF-8 via the `codecs` module.



Code:

```
#####
#
# A simple example of converting some Unicode text to an Excel file using
# the XlsxWriter Python module.
#
# This example generates a spreadsheet with some Polish text from a file
# with UTF8 encoded text.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
import codecs
from xlsxwriter.workbook import Workbook

# Open the input file with the correct encoding.
textfile = codecs.open('unicode_polish_utf8.txt', 'r', 'utf-8')

# Create an new Excel file and convert the text data.
workbook = Workbook('unicode_polish_utf8.xlsx')
worksheet = workbook.add_worksheet()

# Widen the first column to make the text clearer.
worksheet.set_column('A:A', 50)
```

```
# Start from the first cell.
row = 0
col = 0

# Read the text file and write it to the worksheet.
for line in textfile:
    # Ignore the comments in the text file.
    if line.startswith('#'):
        continue

    # Write any other lines to the worksheet.
    worksheet.write(row, col, line.rstrip("\n"))
    row += 1

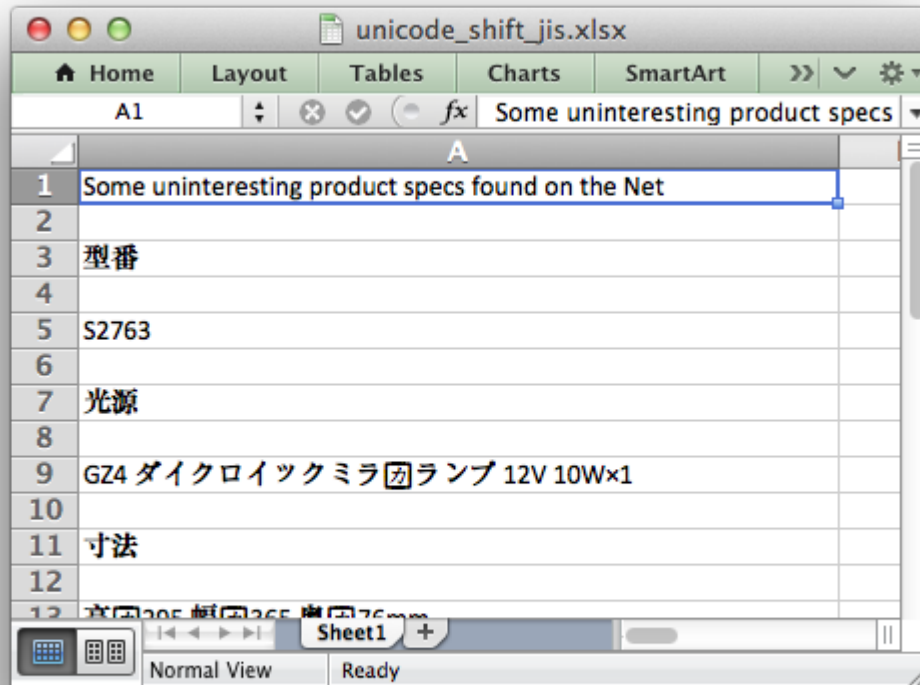
workbook.close()
```

18.17 Example: Unicode - Shift JIS

This program is an example of reading in data from a Shift JIS encoded text file and converting it to a worksheet.

The main trick is to ensure that the data read in is converted to UTF-8 within the Python program. The XlsxWriter module will then take care of writing the encoding to the Excel file.

The encoding of the input data shouldn't matter once it can be converted to UTF-8 via the `codecs` module.



Code:

```
#####
#
# A simple example of converting some Unicode text to an Excel file using
# the XlsxWriter Python module.
#
# This example generates a spreadsheet with some Japanese text from a file
# with Shift-JIS encoded text.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
import codecs
from xlsxwriter.workbook import Workbook

# Open the input file with the correct encoding.
textfile = codecs.open('unicode_shift_jis.txt', 'r', 'shift_jis')

# Create an new Excel file and convert the text data.
workbook = Workbook('unicode_shift_jis.xlsx')
worksheet = workbook.add_worksheet()

# Widen the first column to make the text clearer.
worksheet.set_column('A:A', 50)
```

```

# Start from the first cell.
row = 0
col = 0

# Read the text file and write it to the worksheet.
for line in textfile:
    # Ignore the comments in the text file.
    if line.startswith('#'):
        continue

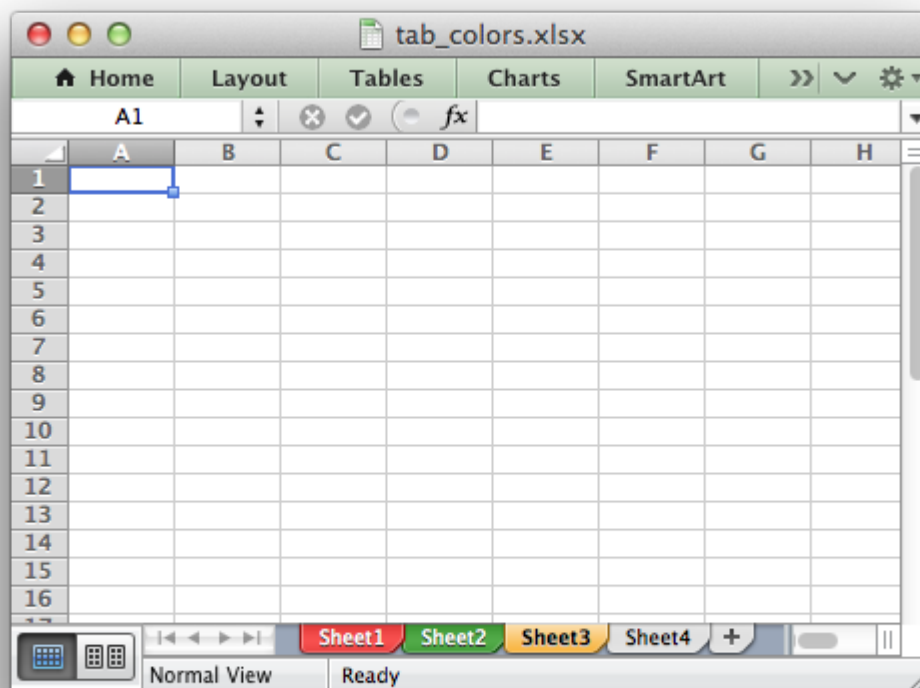
    # Write any other lines to the worksheet.
    worksheet.write(row, col, line.rstrip("\n"))
    row += 1

workbook.close()

```

18.18 Example: Setting Worksheet Tab Colours

This program is an example of setting worksheet tab colours. See the `set_tab_color()` method for more details.

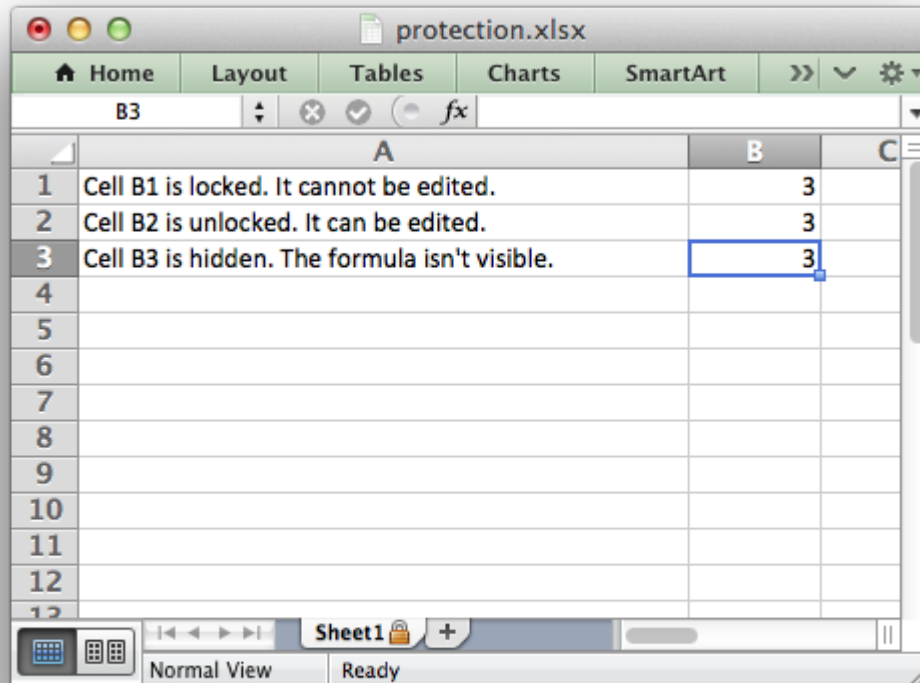


Code:

```
#####  
#  
# Example of how to set Excel worksheet tab colours using Python  
# and the XlsxWriter module..  
#  
# Copyright 2013, John McNamara, jmcnamara@cpan.org  
#  
from xlsxwriter.workbook import Workbook  
  
workbook = Workbook('tab_colors.xlsx')  
  
# Set up some worksheets.  
worksheet1 = workbook.add_worksheet()  
worksheet2 = workbook.add_worksheet()  
worksheet3 = workbook.add_worksheet()  
worksheet4 = workbook.add_worksheet()  
  
# Set tab colours  
worksheet1.set_tab_color('red')  
worksheet2.set_tab_color('green')  
worksheet3.set_tab_color('#FF9900') # Orange  
  
# worksheet4 will have the default colour.  
  
workbook.close()
```

18.19 Example: Enabling Cell protection in Worksheets

This program is an example cell locking and formula hiding in an Excel worksheet using the `protect()` worksheet method.



Code:

```
#####
#
# Example of cell locking and formula hiding in an Excel worksheet
# using Python and the XlsxWriter module.
#
# Copyright 2013, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

workbook = Workbook('protection.xlsx')
worksheet = workbook.add_worksheet()

# Create some cell formats with protection properties.
unlocked = workbook.add_format({'locked': 0})
hidden = workbook.add_format({'hidden': 1})

# Format the columns to make the text more visible.
worksheet.set_column('A:A', 40)

# Turn worksheet protection on.
worksheet.protect()

# Write a locked, unlocked and hidden cell.
```

```
worksheet.write('A1', 'Cell B1 is locked. It cannot be edited.')
worksheet.write('A2', 'Cell B2 is unlocked. It can be edited.')
worksheet.write('A3', "Cell B3 is hidden. The formula isn't visible.")

worksheet.write_formula('B1', '=1+2') # Locked by default.
worksheet.write_formula('B2', '=1+2', unlocked)
worksheet.write_formula('B3', '=1+2', hidden)

workbook.close()
```

COMPARISON WITH EXCEL::WRITER::XLSX

`Excel::Writer::XLSX` is a module written in Perl for creating Excel 2007+ XLSX files.

`Excel::Writer::XLSX` is an API compatible rewrite of an older Perl module called `Spreadsheet::WriteExcel` that creates Excel XLS file.

In terms of features `Excel::Writer::XLSX` is one most complete open source libraries for writing Excel files. It supports:

- Multiple worksheets
- Strings and numbers
- Unicode text
- Cell formatting
- Formulas
- Images
- Charts
- Autofilters
- Data validation
- Conditional formatting
- Macros
- Tables
- Shapes
- Sparklines
- Hyperlinks
- Rich string formats
- Defined names
- Grouping/Outlines

- Cell comments
- Panes
- Page set-up and printing options

Excel::Writer::XLSX has comprehensive documentation, a large number of [example files](#) and an extensive test suite.

Excel::Writer::XLSX and XlsxWriter are written by [John McNamara](#).

19.1 Compatibility with Excel::Writer::XLSX

Porting of Excel::Writer::XLSX to XlsxWriter is a work in progress. The following table shows the level of compatibility between the two module.

19.1.1 Workbook

Status: ongoing.

Workbook Methods	XlsxWriter	Excel::Writer::XLSX
<code>add_worksheet()</code>	Yes	Yes
<code>add_format()</code>	Yes	Yes
<code>add_chart()</code>	No	Yes
<code>add_shape()</code>	No	Yes
<code>add_vba_project()</code>	No	Yes
<code>close()</code>	Yes	Yes
<code>set_properties()</code>	Yes	Yes
<code>define_name()</code>	Yes	Yes
<code>set_tmpdir()</code>	No	Yes
<code>set_custom_color()</code>	No (1)	Yes
<code>worksheets()</code>	Yes (2)	Yes
<code>set_1904()</code>	No	Yes
<code>set_optimization()</code>	Yes (3)	Yes

1. Not required in XlsxWriter. Full RGB colors are supported.
2. Called `sheets()` in Excel::Writer::XLSX.
3. This is a constructor parameter in XlsxWriter.

19.1.2 Worksheet

Status: ongoing.

Worksheet Methods	XlsxWriter	Excel::Writer::XLSX
<code>write()</code>	Yes	Yes
Continued on next page		

Table 19.1 – continued from previous page

Worksheet Methods	XlsxWriter	Excel::Writer::XLSX
<code>write_number()</code>	Yes	Yes
<code>write_string()</code>	Yes	Yes
<code>write_rich_string()</code>	Yes	Yes
<code>write_blank()</code>	Yes	Yes
<code>write_row()</code>	Yes	Yes
<code>write_column()</code>	Yes	Yes
<code>write_datetime()</code>	Yes	Yes
<code>write_url()</code>	Yes	Yes
<code>write_formula()</code>	Yes	Yes
<code>write_array_formula()</code>	Yes	Yes
<code>keep_leading_zeros()</code>	No	Yes
<code>write_comment()</code>	Yes	Yes
<code>show_comments()</code>	Yes	Yes
<code>set_comments_author()</code>	Yes	Yes
<code>add_write_handler()</code>	No	Yes
<code>insert_image()</code>	Yes	Yes
<code>insert_chart()</code>	No	Yes
<code>insert_shape()</code>	No	Yes
<code>insert_button()</code>	No	Yes
<code>data_validation()</code>	Yes	Yes
<code>conditional_format()</code>	Yes (1)	Yes
<code>add_sparkline()</code>	No	Yes
<code>add_table()</code>	No	Yes
<code>get_name()</code>	Yes	Yes
<code>activate()</code>	Yes	Yes
<code>select()</code>	Yes	Yes
<code>hide()</code>	Yes	Yes
<code>set_first_sheet()</code>	Yes	Yes
<code>protect()</code>	Yes	Yes
<code>set_selection()</code>	No	Yes
<code>set_row()</code>	Yes	Yes
<code>set_column()</code>	Yes	Yes
<code>set_default_row()</code>	No	Yes
<code>outline_settings()</code>	No	Yes
<code>freeze_panes()</code>	No	Yes
<code>split_panes()</code>	No	Yes
<code>merge_range()</code>	Yes	Yes
<code>merge_range_type()</code>	No (2)	Yes
<code>set_zoom()</code>	Yes	Yes
<code>right_to_left()</code>	Yes	Yes
<code>hide_zero()</code>	Yes	Yes
<code>set_tab_color()</code>	Yes	Yes
<code>autofilter()</code>	Yes	Yes

Continued on next page

Table 19.1 – continued from previous page

Worksheet Methods	XlsxWriter	Excel::Writer::XLSX
<code>filter_column()</code>	Yes	Yes
<code>filter_column_list()</code>	Yes	Yes

1. Called `conditional_formatting()` in `Excel::Writer::XLSX`.
2. Not required in `XlsxWriter`. The same functionality is available via `merge_range()`.

19.1.3 Page Setup

Status: complete.

Page Set-up Methods	XlsxWriter	Excel::Writer::XLSX
<code>set_landscape()</code>	Yes	Yes
<code>set_portrait()</code>	Yes	Yes
<code>set_page_view()</code>	Yes	Yes
<code>set_paper()</code>	Yes	Yes
<code>center_horizontally()</code>	Yes	Yes
<code>center_vertically()</code>	Yes	Yes
<code>set_margins()</code>	Yes	Yes
<code>set_header()</code>	Yes	Yes
<code>set_footer()</code>	Yes	Yes
<code>repeat_rows()</code>	Yes	Yes
<code>repeat_columns()</code>	Yes	Yes
<code>hide_gridlines()</code>	Yes	Yes
<code>print_row_col_headers()</code>	Yes	Yes
<code>print_area()</code>	Yes	Yes
<code>print_across()</code>	Yes	Yes
<code>fit_to_pages()</code>	Yes	Yes
<code>set_start_page()</code>	Yes	Yes
<code>set_print_scale()</code>	Yes	Yes
<code>set_h_pagebreaks()</code>	Yes	Yes
<code>set_v_pagebreaks()</code>	Yes	Yes

19.1.4 Format

Status: complete.

Format Methods	XlsxWriter	Excel::Writer::XLSX
<code>set_font_name()</code>	Yes	Yes
<code>set_font_size()</code>	Yes	Yes
<code>set_font_color()</code>	Yes	Yes
<code>set_bold()</code>	Yes	Yes
<code>set_italic()</code>	Yes	Yes
Continued on next page		

Table 19.2 – continued from previous page

Format Methods	XlsxWriter	Excel::Writer::XLSX
<code>set_underline()</code>	Yes	Yes
<code>set_font_strikeout()</code>	Yes	Yes
<code>set_font_script()</code>	Yes	Yes
<code>set_num_format()</code>	Yes	Yes
<code>set_locked()</code>	Yes	Yes
<code>set_hidden()</code>	Yes	Yes
<code>set_align()</code>	Yes	Yes
<code>set_rotation()</code>	Yes	Yes
<code>set_text_wrap()</code>	Yes	Yes
<code>set_text_justlast()</code>	Yes	Yes
<code>set_center_across()</code>	Yes	Yes
<code>set_indent()</code>	Yes	Yes
<code>set_shrink()</code>	Yes	Yes
<code>set_pattern()</code>	Yes	Yes
<code>set_bg_color()</code>	Yes	Yes
<code>set_fg_color()</code>	Yes	Yes
<code>set_border()</code>	Yes	Yes
<code>set_bottom()</code>	Yes	Yes
<code>set_top()</code>	Yes	Yes
<code>set_left()</code>	Yes	Yes
<code>set_right()</code>	Yes	Yes
<code>set_border_color()</code>	Yes	Yes
<code>set_bottom_color()</code>	Yes	Yes
<code>set_top_color()</code>	Yes	Yes
<code>set_left_color()</code>	Yes	Yes
<code>set_right_color()</code>	Yes	Yes

ALTERNATIVE MODULES FOR HANDLING EXCEL FILES

The following are some Python alternatives to XlsxWriter.

20.1 XLWT

From the [xlwt](#) page on PyPI:

Library to create spreadsheet files compatible with MS Excel 97/2000/XP/2003 XLS files, on any platform, with Python 2.3 to 2.7.

xlwt is a library for generating spreadsheet files that are compatible with Excel 97/2000/XP/2003, OpenOffice.org Calc, and Gnumeric. xlwt has full support for Unicode. Excel spreadsheets can be generated on any platform without needing Excel or a COM server. The only requirement is Python 2.3 to 2.7.

20.2 XLRD

From the [xlrd](#) page on PyPI:

Library for developers to extract data from Microsoft Excel (tm) spreadsheet files. Extract data from Excel spreadsheets (.xls and .xlsx, versions 2.0 onwards) on any platform. Pure Python (2.6, 2.7, 3.2+). Strong support for Excel dates. Unicode-aware.

20.3 OpenPyXL

From the [openpyxl](#) page on PyPI:

A Python library to read/write Excel 2007 xlsx/xlsm files. Openpyxl is a pure python reader and writer of Excel OpenXML files. It is ported from the PHPExcel project.

KNOWN ISSUES AND BUGS

This section lists known issues and bugs and gives some information on how to submit bug reports.

21.1 ‘unknown encoding: utf-8’ Error

The following error can occur on Windows if the `close()` method isn’t used at the end of the program:

```
Exception LookupError: 'unknown encoding: utf-8' in <bound method
Workbook.__del__ of <xlsxwriter.workbook.Workbook object at 0x022C1450>>
```

This appears to be an issue with the implicit destructor on Windows. It is under investigation. Use `close()` as a workaround.

21.2 Formula results not displaying in Excel

Some early versions of Excel 2007 do not display the calculated values of formulas written by `XlsxWriter`. Applying all available Service Packs to Excel should fix this.

21.3 Formula results displaying as zero in non-Excel applications

Due to wide range of possible formulas and interdependencies between them `XlsxWriter` doesn’t, and realistically cannot, calculate the result of a formula when it is written to an XLSX file. Instead, it stores the value 0 as the formula result. It then sets a global flag in the XLSX file to say that all formulas and functions should be recalculated when the file is opened.

This is the method recommended in the Excel documentation and in general it works fine with spreadsheet applications. However, applications that don’t have a facility to calculate formulas, such as Excel Viewer, or several mobile applications, will only display the 0 results.

If required, it is also possible to specify the calculated result of the formula using the optional `value` parameter in `write_formula()`:

```
worksheet.write_formula('A1', '=2+2', num_format, 4)
```

21.4 Strings aren't displayed in Apple Numbers in 'constant_memory' mode

In `Workbook()` 'constant_memory' mode XlsxWriter uses an optimisation where cell strings aren't stored in an Excel structure call "shared strings" and instead are written "in-line".

This is a documented Excel feature that is supported by most spreadsheet applications. One known exception is Apple Numbers for Mac where the string data isn't displayed.

REPORTING BUGS

Here are some tips on reporting bugs in XlsxWriter.

22.1 Upgrade to the latest version of the module

The bug you are reporting may already be fixed in the latest version of the module. Check the *Changes in XlsxWriter* section as well.

22.2 Read the documentation

The XlsxWriter documentation has been refined in response to user questions. Therefore, if you have a question it is possible that someone else has asked it before you and that it is already addressed in the documentation.

22.3 Look at the example programs

There are several example programs in the distribution. Many of these were created in response to user questions. Try to identify an example program that corresponds to your query and adapt it to your needs.

22.4 Use the official XlsxWriter Issue tracker on GitHub

The official XlsxWriter [Issue tracker](#) is on GitHub.

22.5 Pointers for submitting a bug report

1. Describe the problem as clearly and as concisely as possible.
2. Include a sample program. This is probably the most important step. Also, it is often easier to describe a problem in code than in written prose.

3. The sample program should be as small as possible to demonstrate the problem. Don't copy and past large sections of your program. The program should also be self contained and working.

A sample bug report is shown below. If you use this format then it will help to analyse your question and respond to it more quickly.

XlsxWriter Issue with SOMETHING

I am using XlsxWriter and I have encountered a problem. I want it to do SOMETHING but the module appears to do SOMETHING ELSE.

I am using Python version X.Y.Z and XlsxWriter x.y.z.

Here is some code that demonstrates the problem:

```
from xlsxwriter.workbook import Workbook

workbook = Workbook('hello.xlsx')
worksheet = workbook.add_worksheet()

worksheet.write('A1', 'Hello world')

workbook.close()
```

FREQUENTLY ASKED QUESTIONS

The section outlines some answers to frequently asked questions.

23.1 Q. Can XlsxWriter use an existing Excel file as a template?

No.

XlsxWriter is designed only as a file *writer*. It cannot read or modify an existing Excel file.

23.2 Q. Why do my formulas show a zero result in some, non-Excel applications?

Due to wide range of possible formulas and interdependencies between them XlsxWriter doesn't, and realistically cannot, calculate the result of a formula when it is written to an XLSX file. Instead, it stores the value 0 as the formula result. It then sets a global flag in the XLSX file to say that all formulas and functions should be recalculated when the file is opened.

This is the method recommended in the Excel documentation and in general it works fine with spreadsheet applications. However, applications that don't have a facility to calculate formulas, such as Excel Viewer, or several mobile applications, will only display the 0 results.

If required, it is also possible to specify the calculated result of the formula using the optional `value` parameter in `write_formula()`:

```
worksheet.write_formula('A1', '=2+2', num_format, 4)
```

23.3 Q. Can I apply a format to a range of cells in one go?

Currently no. However, it is a planned features to allow cell formats and data to be written separately.

23.4 Q. Is feature X supported or will it be supported?

All supported features are documented.

Future features will match features that are available in `Excel::Writer::XLSX`. Check the feature matrix in the [Comparison with Excel::Writer::XLSX](#) section.

23.5 Q. Is there an “AutoFit” option for columns?

Unfortunately, there is no way to specify “AutoFit” for a column in the Excel file format. This feature is only available at runtime from within Excel. It is possible to simulate “AutoFit” by tracking the width of the data in the column as you write it.

23.6 Q. Do people actually ask these questions frequently, or at all?

Apart from this question, yes.

CHANGES IN XLSXWRITER

This section shows changes and bug fixes in the XlsxWriter module.

24.1 Release 0.2.4 - March 31 2013

- Added `Workbook()` 'constant_memory' constructor property to minimise memory usage when writing large files. See *Working with Memory and Performance* for more details.
- Fixed bug with handling of UTF-8 strings in worksheet names (and probably some other places as well). Reported by Josh English.
- Fixed bug where temporary directory used to create xlsx files wasn't cleaned up after program close.

24.2 Release 0.2.3 - March 27 2013

- Fixed bug that was killing performance for medium sized files. The module is now 10x faster than previous versions. Reported by John Yeung.

24.3 Release 0.2.2 - March 27 2013

- Added worksheet data validation options. See the `data_validation()` method, *Working with Data Validation* and *Example: Data Validation and Drop Down Lists*.
- There are now over 600 unit tests including more than 130 tests that compare against the output of Excel.

24.4 Release 0.2.1 - March 25 2013

- Added support for `datetime.datetime`, `datetime.date` and `datetime.time` to the `write_datetime()` method. GitHub issue #3. Thanks to Eduardo (eazb) and Josh English for the prompt.

24.5 Release 0.2.0 - March 24 2013

- Added conditional formatting. See the `conditional_format()` method, *Working with Conditional Formatting* and *Example: Conditional Formatting*.

24.6 Release 0.1.9 - March 19 2013

- Added Python 2.6 support. All tests now pass in the following versions:
 - Python 2.6
 - Python 2.7.2
 - Python 2.7.3
 - Python 3.1
 - Python 3.2
 - Python 3.3.0

24.7 Release 0.1.8 - March 18 2013

- Fixed Python 3 support.

24.8 Release 0.1.7 - March 18 2013

- Added the option to write cell comments to a worksheet. See `write_comment()` and *Working with Cell Comments*.

24.9 Release 0.1.6 - March 17 2013

- Added `insert_image()` worksheet method to support inserting PNG and JPEG images into a worksheet. See also the example program *Example: Inserting images into a worksheet*.
- There are now over 500 unit tests including more than 100 tests that compare against the output of Excel.

24.10 Release 0.1.5 - March 10 2013

- Added the `write_rich_string()` worksheet method to allow writing of text with multiple formats to a cell. Also added example program: *Example: Writing “Rich” strings with multiple formats*.

- Added the `hide()` worksheet method to hide worksheets.
- Added the `set_first_sheet()` worksheet method.

24.11 Release 0.1.4 - March 8 2013

- Added the `protect()` worksheet method to allow protection of cells from editing. Also added example program: *Example: Enabling Cell protection in Worksheets*.

24.12 Release 0.1.3 - March 7 2013

- Added worksheet methods:
 - `set_zoom()` for setting worksheet zoom levels.
 - `right_to_left()` for middle eastern versions of Excel.
 - `hide_zero()` for hiding zero values in cells.
 - `set_tab_color()` for setting the worksheet tab colour.

24.13 Release 0.1.2 - March 6 2013

- Added autofilters. See *Working with Autofilters* for more details.
- Added the `write_row()` and `write_column()` worksheet methods.

24.14 Release 0.1.1 - March 3 2013

- Added the `write_url()` worksheet method for writing hyperlinks to a worksheet.

24.15 Release 0.1.0 - February 28 2013

- Added the `set_properties()` workbook method for setting document properties.
- Added several new examples programs with documentation. The examples now include:
 - `array_formula.py`
 - `cell_indentation.py`
 - `datetimes.py`
 - `defined_name.py`
 - `demo.py`

- doc_properties.py
- headers_footers.py
- hello_world.py
- merge1.py
- tutorial1.py
- tutorial2.py
- tutorial3.py
- unicode_polish_utf8.py
- unicode_shift_jis.py

24.16 Release 0.0.9 - February 27 2013

- Added the `define_name()` method to create defined names and ranges in a workbook or worksheet.
- Added the `worksheets()` method as an accessor for the worksheets in a workbook.

24.17 Release 0.0.8 - February 26 2013

- Added the `merge_range()` method to merge worksheet cells.

24.18 Release 0.0.7 - February 25 2013

- Added final page setup methods to complete the page setup section.
 - `print_area()`
 - `fit_to_pages()`
 - `set_start_page()`
 - `set_print_scale()`
 - `set_h_pagebreaks()`
 - `set_v_pagebreaks()`

24.19 Release 0.0.6 - February 22 2013

- Added page setup method.
 - `print_row_col_headers`

24.20 Release 0.0.5 - February 21 2013

- Added page setup methods.
 - repeat_rows()
 - repeat_columns()

24.21 Release 0.0.4 - February 20 2013

- Added Python 3 support with help from John Evans. Tested with:
 - Python-2.7.2
 - Python-2.7.3
 - Python-3.2
 - Python-3.3.0
- Added page setup methods.
 - center_horizontally()
 - center_vertically()
 - set_header()
 - set_footer()
 - hide_gridlines()

24.22 Release 0.0.3 - February 19 2013

- Added page setup method.
 - set_margins()

24.23 Release 0.0.2 - February 18 2013

- Added page setup methods.
 - set_landscape()
 - set_portrait()
 - set_page_view()
 - set_paper()
 - print_across()

24.24 Release 0.0.1 - February 17 2013

- First public release.

AUTHOR

XlsxWriter was written by John McNamara.

- [GitHub repos](#)
- [Perl CPAN modules](#)
- [Twitter @jmcnamara13](#)
- [Coderwall](#)
- [Ohloh](#)

You can contact me at jmcnamara@cpan.org.

LICENSE

XlsxWriter is released under a BSD license.

Copyright (c) 2013, John McNamara <jmcnamara@cpan.org> All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the FreeBSD Project.

INDEX

A

activate() (built-in function), 56
add_format() (built-in function), 25
add_worksheet() (built-in function), 25
autofilter() (built-in function), 59

C

center_horizontally() (built-in function), 67
center_vertically() (built-in function), 67
close() (built-in function), 26
conditional_format() (built-in function), 51

D

data_validation() (built-in function), 50
define_name() (built-in function), 28

F

filter_column() (built-in function), 60
filter_column_list() (built-in function), 61
fit_to_pages() (built-in function), 74

G

get_name() (built-in function), 55

H

hide() (built-in function), 57
hide_gridlines() (built-in function), 72
hide_zero() (built-in function), 62

I

insert_image() (built-in function), 48

M

merge_range() (built-in function), 58

P

print_across() (built-in function), 73

print_area() (built-in function), 73
print_row_col_headers() (built-in function), 73
protect() (built-in function), 63

R

repeat_columns() (built-in function), 72
repeat_rows() (built-in function), 71
right_to_left() (built-in function), 62

S

select() (built-in function), 56
set_align() (built-in function), 84
set_bg_color() (built-in function), 87
set_bold() (built-in function), 79
set_border() (built-in function), 88
set_border_color() (built-in function), 89
set_bottom() (built-in function), 89
set_bottom_color() (built-in function), 90
set_center_across() (built-in function), 84
set_column() (built-in function), 46
set_comments_author() (built-in function), 55
set_fg_color() (built-in function), 87
set_first_sheet() (built-in function), 57
set_font_color() (built-in function), 78
set_font_name() (built-in function), 78
set_font_script() (built-in function), 80
set_font_size() (built-in function), 78
set_font_strikeout() (built-in function), 79
set_footer() (built-in function), 71
set_h_pagebreaks() (built-in function), 75
set_header() (built-in function), 68
set_hidden() (built-in function), 83
set_indent() (built-in function), 86
set_italic() (built-in function), 79
set_landscape() (built-in function), 65
set_left() (built-in function), 89
set_left_color() (built-in function), 90

set_locked() (built-in function), 83
set_margins() (built-in function), 68
set_num_format() (built-in function), 80
set_page_view() (built-in function), 65
set_paper() (built-in function), 66
set_pattern() (built-in function), 86
set_portrait() (built-in function), 65
set_print_scale() (built-in function), 75
set_properties() (built-in function), 26
set_right() (built-in function), 89
set_right_color() (built-in function), 90
set_rotation() (built-in function), 85
set_row() (built-in function), 44
set_shrink() (built-in function), 86
set_start_page() (built-in function), 75
set_tab_color() (built-in function), 62
set_text_justlast() (built-in function), 86
set_text_wrap() (built-in function), 85
set_top() (built-in function), 89
set_top_color() (built-in function), 90
set_underline() (built-in function), 79
set_v_pagebreaks() (built-in function), 76
set_zoom() (built-in function), 61
show_comments() (built-in function), 55

W

Workbook() (built-in function), 23
worksheets() (built-in function), 29
write() (built-in function), 31
write_array_formula() (built-in function), 37
write_blank() (built-in function), 38
write_column() (built-in function), 44
write_comment() (built-in function), 53
write_datetime() (built-in function), 39
write_formula() (built-in function), 36
write_number() (built-in function), 35
write_rich_string() (built-in function), 41
write_row() (built-in function), 43
write_string() (built-in function), 34
write_url() (built-in function), 39