Creating Excel files with Python and XlsxWriter

Release 0.6.7

John McNamara

March 02, 2015

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 **21**
6.1 Constructor 21
6.2 workbook.add_worksheet() 24
6.3 workbook.add_format() 24
6.4 workbook.add_chart() 25
6.5 workbook.add_chartsheet() 26
6.6 workbook.close() 27
6.7 workbook.set_properties() 27
6.8 workbook.define_name() 29
6.9 workbook.add_vba_project() 30
6.10 workbook.set_vba_name() 30
6.11 workbook.worksheets() 31
6.12 workbook.set_calc_mode() 31
6.13 workbook.use_zip64() 31

7 The Worksheet Class **33**
7.1 worksheet.write() 33
7.2 worksheet.write_string() 36
7.3 worksheet.write_number() 37
7.4 worksheet.write_formula() 38
7.5 worksheet.write_array_formula() 39
7.6 worksheet.write_blank() 40
7.7 worksheet.write_boolean() 41

7.8 worksheet.write_datetime() . . . 41
7.9 worksheet.write_url() . . . 42
7.10 worksheet.write_rich_string() . . . 44
7.11 worksheet.write_row() . . . 46
7.12 worksheet.write_column() . . . 47
7.13 worksheet.set_row() . . . 47
7.14 worksheet.set_column() . . . 49
7.15 worksheet.insert_image() . . . 51
7.16 worksheet.insert_chart() . . . 54
7.17 worksheet.insert_button() . . . 55
7.18 worksheet.data_validation() . . . 57
7.19 worksheet.conditional_format() . . . 58
7.20 worksheet.add_table() . . . 60
7.21 worksheet.add_sparkline() . . . 60
7.22 worksheet.write_comment() . . . 61
7.23 worksheet.show_comments() . . . 63
7.24 worksheet.set_comments_author() . . . 63
7.25 worksheet.get_name() . . . 64
7.26 worksheet.activate() . . . 64
7.27 worksheet.select() . . . 65
7.28 worksheet.hide() . . . 65
7.29 worksheet.set_first_sheet() . . . 66
7.30 worksheet.merge_range() . . . 67
7.31 worksheet.autofilter() . . . 69
7.32 worksheet.filter_column() . . . 70
7.33 worksheet.filter_column_list() . . . 71
7.34 worksheet.set_selection() . . . 71
7.35 worksheet.freeze_panes() . . . 72
7.36 worksheet.split_panes() . . . 73
7.37 worksheet.set_zoom() . . . 74
7.38 worksheet.right_to_left() . . . 74
7.39 worksheet.hide_zero() . . . 74
7.40 worksheet.set_tab_color() . . . 74
7.41 worksheet.protect() . . . 75
7.42 worksheet.set_default_row() . . . 76
7.43 worksheet.outline_settings() . . . 76
7.44 worksheet.set_vba_name() . . . 77

8 The Worksheet Class (Page Setup) 79
8.1 worksheet.set_landscape() . . . 79
8.2 worksheet.set_portrait() . . . 79
8.3 worksheet.set_page_view() . . . 79
8.4 worksheet.set_paper() . . . 80
8.5 worksheet.center_horizontally() . . . 81
8.6 worksheet.center_vertically() . . . 81
8.7 worksheet.set_margins() . . . 81
8.8 worksheet.set_header() . . . 82
8.9 worksheet.set_footer() . . . 86

8.10 worksheet.repeat_rows() . . . 87
8.11 worksheet.repeat_columns() . . . 87
8.12 worksheet.hide_gridlines() . . . 87
8.13 worksheet.print_row_col_headers() . . . 88
8.14 worksheet.print_area() . . . 88
8.15 worksheet.print_across() . . . 89
8.16 worksheet.fit_to_pages() . . . 89
8.17 worksheet.set_start_page() . . . 90
8.18 worksheet.set_print_scale() . . . 90
8.19 worksheet.set_h_pagebreaks() . . . 91
8.20 worksheet.set_v_pagebreaks() . . . 91

9 The Format Class **93**
9.1 format.set_font_name() . . . 93
9.2 format.set_font_size() . . . 94
9.3 format.set_font_color() . . . 94
9.4 format.set_bold() . . . 95
9.5 format.set_italic() . . . 95
9.6 format.set_underline() . . . 95
9.7 format.set_font_strikeout() . . . 95
9.8 format.set_font_script() . . . 96
9.9 format.set_num_format() . . . 96
9.10 format.set_locked() . . . 99
9.11 format.set_hidden() . . . 99
9.12 format.set_align() . . . 99
9.13 format.set_center_across() . . . 100
9.14 format.set_text_wrap() . . . 101
9.15 format.set_rotation() . . . 101
9.16 format.set_indent() . . . 101
9.17 format.set_shrink() . . . 102
9.18 format.set_text_justlast() . . . 103
9.19 format.set_pattern() . . . 103
9.20 format.set_bg_color() . . . 103
9.21 format.set_fg_color() . . . 104
9.22 format.set_border() . . . 104
9.23 format.set_bottom() . . . 105
9.24 format.set_top() . . . 105
9.25 format.set_left() . . . 106
9.26 format.set_right() . . . 106
9.27 format.set_border_color() . . . 106
9.28 format.set_bottom_color() . . . 106
9.29 format.set_top_color() . . . 107
9.30 format.set_left_color() . . . 107
9.31 format.set_right_color() . . . 107
9.32 format.set_diag_border() . . . 107
9.33 format.set_diag_type() . . . 108
9.34 format.set_diag_color() . . . 108

10 The Chart Class **111**
10.1 chart.add_series() . . . 113
10.2 chart.set_x_axis() . . . 115
10.3 chart.set_y_axis() . . . 121
10.4 chart.set_x2_axis() . . . 121
10.5 chart.set_y2_axis() . . . 121
10.6 chart.set_size() . . . 122
10.7 chart.set_title() . . . 122
10.8 chart.set_legend() . . . 123
10.9 chart.set_chartarea() . . . 126
10.10 chart.set_plotarea() . . . 127
10.11 chart.set_style() . . . 128
10.12 chart.set_table() . . . 128
10.13 chart.set_up_down_bars() . . . 129
10.14 chart.set_drop_lines() . . . 130
10.15 chart.set_high_low_lines() . . . 131
10.16 chart.show_blanks_as() . . . 132
10.17 chart.show_hidden_data() . . . 132
10.18 chart.set_rotation() . . . 133
10.19 chart.set_hole_size() . . . 133

11 The Chartsheet Class **135**
11.1 chartsheet.set_chart() . . . 136
11.2 Worksheet methods . . . 136
11.3 Chartsheet Example . . . 137

12 Working with Cell Notation **139**

13 Relative and Absolute cell references **141**

14 Defined Names and Named Ranges **143**

15 Cell Utility Functions **145**
15.1 xl_rowcol_to_cell() . . . 145
15.2 xl_cell_to_rowcol() . . . 146
15.3 xl_col_to_name() . . . 146
15.4 xl_range() . . . 146
15.5 xl_range_abs() . . . 147

16 Working with Formats **149**
16.1 Creating and using a Format object . . . 149
16.2 Format methods and Format properties . . . 149
16.3 Format Colors . . . 151
16.4 Format Defaults . . . 151
16.5 Modifying Formats . . . 151

17 Working with Dates and Time **153**
17.1 Default Date Formatting . . . 156

18 Working with Colors **157**

19 Working with Charts **159**
19.1 Chart Value and Category Axes . . . 160
19.2 Chart Series Options . . . 161
19.3 Chart series option: Marker . . . 161
19.4 Chart series option: Trendline . . . 163
19.5 Chart series option: Error Bars . . . 166
19.6 Chart series option: Data Labels . . . 168
19.7 Chart series option: Points . . . 171
19.8 Chart series option: Smooth . . . 173
19.9 Chart Formatting . . . 173
19.10 Chart formatting: Line . . . 174
19.11 Chart formatting: Border . . . 177
19.12 Chart formatting: Fill . . . 177
19.13 Chart Fonts . . . 178
19.14 Chart Layout . . . 180
19.15 Date Category Axes . . . 182
19.16 Chart Secondary Axes . . . 183
19.17 Chartsheets . . . 184
19.18 Chart Limitations . . . 185
19.19 Chart Examples . . . 185

20 Working with Autofilters **187**
20.1 Applying an autofilter . . . 187
20.2 Filter data in an autofilter . . . 188
20.3 Setting a filter criteria for a column . . . 189
20.4 Setting a column list filter . . . 190
20.5 Example . . . 191

21 Working with Data Validation **193**
21.1 data_validation() . . . 195
21.2 Data Validation Examples . . . 200

22 Working with Conditional Formatting **203**
22.1 The conditional_format() method . . . 206
22.2 Conditional Format Options . . . 207
22.3 Conditional Formatting Examples . . . 217

23 Working with Worksheet Tables **219**
23.1 add_table() . . . 220
23.2 data . . . 221
23.3 header_row . . . 222
23.4 autofilter . . . 223
23.5 banded_rows . . . 224
23.6 banded_columns . . . 225
23.7 first_column . . . 225
23.8 last_column . . . 226

23.9 style . . . 226
23.10 name . . . 227
23.11 total_row . . . 228
23.12 columns . . . 228
23.13 Example . . . 233

24 Working with Sparklines **235**
24.1 The add_sparkline() method . . . 235
24.2 range . . . 238
24.3 type . . . 238
24.4 style . . . 238
24.5 markers . . . 239
24.6 negative_points . . . 239
24.7 axis . . . 239
24.8 reverse . . . 239
24.9 weight . . . 239
24.10 high_point, low_point, first_point, last_point . . . 240
24.11 max, min . . . 240
24.12 empty_cells . . . 240
24.13 show_hidden . . . 240
24.14 date_axis . . . 241
24.15 series_color . . . 241
24.16 location . . . 241
24.17 Grouped Sparklines . . . 241
24.18 Sparkline examples . . . 242

25 Working with Cell Comments **243**
25.1 Setting Comment Properties . . . 243

26 Working with Outlines and Grouping **247**
26.1 Outlines and Grouping in XlsxWriter . . . 249

27 Working with Memory and Performance **253**
27.1 Performance Figures . . . 254
27.2 Benchmark of Python Excel Writers . . . 254

28 Working with VBA Macros **257**
28.1 The Excel XLSM file format . . . 257
28.2 How VBA macros are included in XlsxWriter . . . 258
28.3 The vba_extract utility . . . 258
28.4 Adding the VBA macros to a XlsxWriter file . . . 258
28.5 Setting the VBA codenames . . . 259
28.6 What to do if it doesn't work . . . 260

29 Examples **261**
29.1 Example: Hello World . . . 261
29.2 Example: Simple Feature Demonstration . . . 262
29.3 Example: Dates and Times in Excel . . . 263
29.4 Example: Adding hyperlinks . . . 265

29.5 Example: Array formulas . . . 267
29.6 Example: Applying Autofilters . . . 269
29.7 Example: Data Validation and Drop Down Lists . . . 274
29.8 Example: Conditional Formatting . . . 279
29.9 Example: Defined names/Named ranges . . . 285
29.10 Example: Merging Cells . . . 286
29.11 Example: Writing “Rich” strings with multiple formats . . . 288
29.12 Example: Merging Cells with a Rich String . . . 289
29.13 Example: Inserting images into a worksheet . . . 291
29.14 Example: Inserting images from a URL or byte stream into a worksheet . . . 293
29.15 Example: Simple HTTP Server . . . 295
29.16 Example: Adding Headers and Footers to Worksheets . . . 296
29.17 Example: Freeze Panes and Split Panes . . . 300
29.18 Example: Worksheet Tables . . . 302
29.19 Example: Sparklines (Simple) . . . 310
29.20 Example: Sparklines (Advanced) . . . 311
29.21 Example: Adding Cell Comments to Worksheets (Simple) . . . 318
29.22 Example: Adding Cell Comments to Worksheets (Advanced) . . . 320
29.23 Example: Outline and Grouping . . . 325
29.24 Example: Collapsed Outline and Grouping . . . 330
29.25 Example: Setting Document Properties . . . 334
29.26 Example: Simple Unicode with Python 2 . . . 336
29.27 Example: Simple Unicode with Python 3 . . . 337
29.28 Example: Unicode - Polish in UTF-8 . . . 337
29.29 Example: Unicode - Shift JIS . . . 339
29.30 Example: Setting Worksheet Tab Colours . . . 341
29.31 Example: Diagonal borders in cells . . . 342
29.32 Example: Enabling Cell protection in Worksheets . . . 344
29.33 Example: Hiding Worksheets . . . 345
29.34 Example: Hiding Rows and Columns . . . 347
29.35 Example: Adding a VBA macro to a Workbook . . . 348

30 Chart Examples **351**
30.1 Example: Chart (Simple) . . . 351
30.2 Example: Area Chart . . . 352
30.3 Example: Bar Chart . . . 356
30.4 Example: Column Chart . . . 359
30.5 Example: Line Chart . . . 363
30.6 Example: Pie Chart . . . 365
30.7 Example: Doughnut Chart . . . 368
30.8 Example: Scatter Chart . . . 371
30.9 Example: Radar Chart . . . 377
30.10 Example: Stock Chart . . . 381
30.11 Example: Secondary Axis Chart . . . 383
30.12 Example: Date Axis Chart . . . 385
30.13 Example: Charts with Data Tables . . . 386
30.14 Example: Charts with Data Tools . . . 389
30.15 Example: Chartsheet . . . 396

31 Alternative modules for handling Excel files **399**
31.1 XLWT . . . 399
31.2 XLRD . . . 399
31.3 OpenPyXL . . . 399

32 Known Issues and Bugs **401**
32.1 "Content is Unreadable. Open and Repair" . . . 401
32.2 "Exception caught in workbook destructor. Explicit close() may be required" . . . 401
32.3 Formulas displayed as #NAME? until edited . . . 401
32.4 Formula results displaying as zero in non-Excel applications . . . 402
32.5 Strings aren't displayed in Apple Numbers in 'constant_memory' mode . . . 402
32.6 Images not displayed correctly in Excel 2001 for Mac and non-Excel applications . . . 402

33 Reporting Bugs **403**
33.1 Upgrade to the latest version of the module . . . 403
33.2 Read the documentation . . . 403
33.3 Look at the example programs . . . 403
33.4 Use the official XlsxWriter Issue tracker on GitHub . . . 403
33.5 Pointers for submitting a bug report . . . 403

34 Frequently Asked Questions **405**
34.1 Q. Can XlsxWriter use an existing Excel file as a template? . . . 405
34.2 Q. Why do my formulas show a zero result in some, non-Excel applications? . . . 405
34.3 Q. Can I apply a format to a range of cells in one go? . . . 405
34.4 Q. Is feature X supported or will it be supported? . . . 406
34.5 Q. Is there an "AutoFit" option for columns? . . . 406
34.6 Q. Do people actually ask these questions frequently, or at all? . . . 406

35 Changes in XlsxWriter **407**
35.1 Release 0.6.7 - March 1 2015 . . . 407
35.2 Release 0.6.6 - January 16 2015 . . . 407
35.3 Release 0.6.5 - December 31 2014 . . . 407
35.4 Release 0.6.4 - November 15 2014 . . . 407
35.5 Release 0.6.3 - November 6 2014 . . . 408
35.6 Release 0.6.2 - November 1 2014 . . . 408
35.7 Release 0.6.1 - October 29 2014 . . . 408
35.8 Release 0.6.0 - October 15 2014 . . . 408
35.9 Release 0.5.9 - October 11 2014 . . . 408
35.10 Release 0.5.8 - September 28 2014 . . . 409
35.11 Release 0.5.7 - August 13 2014 . . . 409
35.12 Release 0.5.6 - July 22 2014 . . . 409
35.13 Release 0.5.5 - May 6 2014 . . . 409
35.14 Release 0.5.4 - May 4 2014 . . . 409
35.15 Release 0.5.3 - February 20 2014 . . . 409
35.16 Release 0.5.2 - December 31 2013 . . . 410
35.17 Release 0.5.1 - December 2 2013 . . . 410
35.18 Release 0.5.0 - November 17 2013 . . . 410
35.19 Release 0.4.9 - November 17 2013 . . . 410

35.20 Release 0.4.8 - November 13 2013 . . . 411
35.21 Release 0.4.7 - November 9 2013 . . . 411
35.22 Release 0.4.6 - October 23 2013 . . . 411
35.23 Release 0.4.5 - October 21 2013 . . . 411
35.24 Release 0.4.4 - October 16 2013 . . . 411
35.25 Release 0.4.3 - September 12 2013 . . . 411
35.26 Release 0.4.2 - August 30 2013 . . . 411
35.27 Release 0.4.1 - August 28 2013 . . . 412
35.28 Release 0.4.0 - August 26 2013 . . . 412
35.29 Release 0.3.9 - August 24 2013 . . . 412
35.30 Release 0.3.8 - August 23 2013 . . . 412
35.31 Release 0.3.7 - August 16 2013 . . . 412
35.32 Release 0.3.6 - July 26 2013 . . . 412
35.33 Release 0.3.5 - June 28 2013 . . . 413
35.34 Release 0.3.4 - June 27 2013 . . . 413
35.35 Release 0.3.3 - June 10 2013 . . . 413
35.36 Release 0.3.2 - May 1 2013 . . . 413
35.37 Release 0.3.1 - April 27 2013 . . . 413
35.38 Release 0.3.0 - April 7 2013 . . . 413
35.39 Release 0.2.9 - April 7 2013 . . . 413
35.40 Release 0.2.8 - April 4 2013 . . . 414
35.41 Release 0.2.7 - April 3 2013 . . . 414
35.42 Release 0.2.6 - April 1 2013 . . . 414
35.43 Release 0.2.5 - April 1 2013 . . . 414
35.44 Release 0.2.4 - March 31 2013 . . . 415
35.45 Release 0.2.3 - March 27 2013 . . . 415
35.46 Release 0.2.2 - March 27 2013 . . . 415
35.47 Release 0.2.1 - March 25 2013 . . . 415
35.48 Release 0.2.0 - March 24 2013 . . . 415
35.49 Release 0.1.9 - March 19 2013 . . . 415
35.50 Release 0.1.8 - March 18 2013 . . . 416
35.51 Release 0.1.7 - March 18 2013 . . . 416
35.52 Release 0.1.6 - March 17 2013 . . . 416
35.53 Release 0.1.5 - March 10 2013 . . . 416
35.54 Release 0.1.4 - March 8 2013 . . . 416
35.55 Release 0.1.3 - March 7 2013 . . . 417
35.56 Release 0.1.2 - March 6 2013 . . . 417
35.57 Release 0.1.1 - March 3 2013 . . . 417
35.58 Release 0.1.0 - February 28 2013 . . . 417
35.59 Release 0.0.9 - February 27 2013 . . . 418
35.60 Release 0.0.8 - February 26 2013 . . . 418
35.61 Release 0.0.7 - February 25 2013 . . . 418
35.62 Release 0.0.6 - February 22 2013 . . . 418
35.63 Release 0.0.5 - February 21 2013 . . . 418
35.64 Release 0.0.4 - February 20 2013 . . . 419
35.65 Release 0.0.3 - February 19 2013 . . . 419
35.66 Release 0.0.2 - February 18 2013 . . . 419
35.67 Release 0.0.1 - February 17 2013 . . . 419

36 Author **421**
36.1 Sponsorship 421
36.2 Donations 421

37 License **423**

Index **425**

XlsxWriter is a Python module for creating Excel XLSX files.

XlsxWriter is a Python module that can be used to write text, numbers, formulas and hyperlinks to multiple worksheets in an Excel 2007+ XLSX file. It supports features such as formatting and many more, including:

- 100% compatible Excel XLSX files.
- Full formatting.
- Merged cells.
- Defined names.
- Charts.
- Autofilters.
- Data validation and drop down lists.
- Conditional formatting.
- Worksheet PNG/JPEG images.
- Rich multi-format strings.
- Cell comments.

- Memory optimisation mode for writing large files.

It supports Python 2.5, 2.6, 2.7, 3.1, 3.2, 3.3, 3.4, Jython and PyPy and uses standard libraries only.

CHAPTER

ONE

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, charts, page setup, autofilters, conditional formatting and many others.

XlsxWriter has some advantages and disadvantages over the *alternative Python modules* for writing Excel files.

- Advantages:
  - It supports more Excel features than any of the alternative modules.
  - It has a high degree of fidelity with files produced by Excel. In most cases the files produced are 100% equivalent to files produced by Excel.
  - It has extensive documentation, example files and tests.
  - It is fast and can be configured to use very little memory even for very large output files.
- Disadvantages:
  - It cannot read or modify existing Excel XLSX files.

XlsxWriter is licensed under a BSD *License* and the source code is available on GitHub.

To try out the module see the next section on *Getting Started with XlsxWriter*.

CHAPTER

TWO

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 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 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:

```
import xlsxwriter

workbook = xlsxwriter.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.

CHAPTER

THREE

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:

```
import xlsxwriter

# Create a workbook and add a worksheet.
workbook = xlsxwriter.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:

```
import xlsxwriter
```

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 = xlsxwriter.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.

CHAPTER

FOUR

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 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:

```
import xlsxwriter

# Create a workbook and add a worksheet.
workbook = xlsxwriter.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 headers.
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 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.

CHAPTER

FIVE

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 differences here are that we have added a Date column with formatting and made that column a little wider to accommodate the dates.

To do this we can extend our program as follows:

```
from datetime import datetime
import xlsxwriter

# Create a workbook and add a worksheet.
workbook = xlsxwriter.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': 'mmmm 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, such as strings and numbers, differently although it generally does it transparently to the user. XlsxWriter tries to emulate this in the *worksheet*.`write()` method by mapping Python data types to types that Excel supports.

The `write()` method acts as a general alias for several more specific methods:

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

In this version of our program we have used some of these 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 actually have worked just as well.

The handling of dates is also new to our program.

Dates and times in Excel are floating point numbers that have a number format applied to display them in the correct format. If the date and time are Python `datetime` objects XlsxWriter makes the required number conversion automatically. However, we also need to add the number format to ensure that Excel displays it as as date:

```
from datetime import datetime
...

date_format = workbook.add_format({'num_format': 'mmmm 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 )
    ...
```

Date handling is explained in more detail in *Working with Dates and Time*.

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)
```

That completes the tutorial section.

In the next sections we will look at the API in more detail starting with *The Workbook Class*.

CHAPTER

SIX

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. See below.

Return type A Workbook object.

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

```
import xlsxwriter

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

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

workbook.close()
```

The constructor options are:

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

  ```
  workbook = xlsxwriter.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. Therefore, once this mode is active, data should be written in sequential row order.

  See *Working with Memory and Performance* for more details.

- **tmpdir**: `XlsxWriter` stores worksheet data in a temporary files prior to assembling the final XLSX file. The temporary files are created in the system's temp directory. If the default temporary directory isn't accessible to your application, or doesn't contain enough space, you can specify an alternative location using the `tempdir` option:

  ```
  workbook = xlsxwriter.Workbook(filename, {'tmpdir': '/home/user/tmp'})
  ```

  The temporary directory must exist and will not be created.

- **in_memory**: To avoid the use of temporary files in the assembly of the final XLSX file, for example on servers that don't allow temp files such as the Google APP Engine, set the `in_memory` constructor option to `True`:

```
workbook = xlsxwriter.Workbook(filename, {'in_memory': True})
```

This option overrides the `constant_memory` option.

- **strings_to_numbers**: Enable the *worksheet*.`write()` method to convert strings to numbers, where possible, using `float()` in order to avoid an Excel warning about "Numbers Stored as Text". The default is `False`:

```
workbook = xlsxwriter.Workbook(filename, {'strings_to_numbers': True})
```

- **strings_to_formulas**: Enable the *worksheet*.`write()` method to convert strings to formulas. The default is `True`:

```
workbook = xlsxwriter.Workbook(filename, {'strings_to_formulas': False})
```

- **strings_to_urls**: Enable the *worksheet*.`write()` method to convert strings to urls. The default is `True`:

```
workbook = xlsxwriter.Workbook(filename, {'strings_to_urls': True})
```

- **default_date_format**: This option is used to specify a default date format string for use with the *worksheet*.`write_datetime()` method when an explicit format isn't given. See *Working with Dates and Time* for more details:

```
xlsxwriter.Workbook(filename, {'default_date_format': 'dd/mm/yy'})
```

- **date_1904**: Excel for Windows uses a default epoch of 1900 and Excel for Mac uses an epoch of 1904. However, Excel on either platform will convert automatically between one system and the other. XlsxWriter stores dates in the 1900 format by default. If you wish to change this you can use the `date_1904` workbook option. This option is mainly for enhanced compatibility with Excel and in general isn't required very often:

```
workbook = xlsxwriter.Workbook(filename, {'date_1904': True})
```

When specifying a filename it is recommended that you use an `.xlsx` extension or Excel will generate a warning when opening the file.

It is possible to write files to in-memory strings using StringIO as follows:

```
output = StringIO()
workbook = xlsxwriter.Workbook(output)
worksheet = workbook.add_worksheet()

worksheet.write('A1', 'Hello')
workbook.close()

xlsx_data = output.getvalue()
```

To avoid the use of any temporary files and keep the entire file in-memory use the `in_memory` constructor option shown above.

See also *Example: Simple HTTP Server*.

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 formats 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.add_chart()

add_chart(*options*)

Create a chart object that can be added to a worksheet.

Parameters options (*dictionary*) – An dictionary of chart type options.

Return type A *Chart* object.

This method is use to create a new chart object that can be inserted into a worksheet via the `insert_chart()` Worksheet method:

```
chart = workbook.add_chart({'type': 'column'})
```

The properties that can be set are:

```
type    (required)
subtype (optional)
```

- `type`

  This is a required parameter. It defines the type of chart that will be created:

  ```
  chart = workbook.add_chart({'type': 'line'})
  ```

  The available types are:

  ```
  area
  bar
  column
  doughnut
  line
  pie
  radar
  scatter
  stock
  ```

- `subtype`

  Used to define a chart subtype where available:

```
workbook.add_chart({'type': 'bar', 'subtype': 'stacked'})
```

See the *The Chart Class* for a list of available chart subtypes.

Note: A chart can only be inserted into a worksheet once. If several similar charts are required then each one must be created separately with `add_chart()`.

See also *Working with Charts* and *Chart Examples*.

6.5 workbook.add_chartsheet()

add_chartsheet([*sheetname*])

Add a new add_chartsheet to a workbook.

Parameters **sheetname** (*string*) – Optional chartsheet name, defaults to Chart1, etc.

Return type A *chartsheet* object.

The `add_chartsheet()` method adds a new chartsheet to a workbook.

See *The Chartsheet Class* for details.

The `sheetname` parameter is optional. If it is not specified the default Excel convention will be followed, i.e. Chart1, Chart2, etc.

The chartsheet 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 chartsheet.

6.6 workbook.close()

`close``()`

Close the Workbook object and write the XLSX file.

The workbook `close()` method writes all data to the xlsx file and closes it. This is a mandatory method call:

```
workbook.close()
```

Note: Earlier versions of XlsxWriter allowed an implicit `close()` that was triggered by the garbage collector. However, this proved to be too problematic and non-deterministic. An explicit `close()` is now recommended in all XlsxWriter programs.

6.7 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 are all optional and 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*.

6.8 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. These are sometimes referred to as a "Named Range".

Defined names are generally used to simplify or clarify formulas by using descriptive variable names:

```
workbook.define_name('Exchange_rate', '=0.96')
worksheet.write('B3', '=B2*Exchange_rate')
```

As in Excel a name defined like this is "global" to the workbook and can be referred to from any worksheet:

```
# Global workbook name.
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 follow the Excel convention and enclose it in single quotes:

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

The rules for names in Excel are explained in the Miscrosoft Office documentation.

See also *Example: Defined names/Named ranges*.

6.9 workbook.add_vba_project()

add_vba_project(*vba_project*[, *is_stream*])

Add a vbaProject binary to the Excel workbook.

Parameters

- **vba_project** – The vbaProject binary file name.
- **is_stream** (*bool*) – The vba_project is an in memory byte stream.

The `add_vba_project()` method can be used to add macros or functions to a workbook using a binary VBA project file that has been extracted from an existing Excel xlsm file:

```
workbook.add_vba_project('./vbaProject.bin')
```

Only one `vbaProject.bin` file can be added per workbook.

The `is_stream` parameter is used to indicate that `vba_project` refers to a BytesIO byte stream rather than a physical file. This can be used when working with the workbook `in_memory` mode.

See *Working with VBA Macros* for more details.

6.10 workbook.set_vba_name()

set_vba_name(*name*)

Set the VBA name for the workbook.

Parameters **name** (*string*) – The VBA name for the workbook.

The `set_vba_name()` method can be used to set the VBA codename for the workbook. This is sometimes required when a vbaProject macro included via `add_vba_project()` refers to the

workbook. The default Excel VBA name of `ThisWorkbook` is used if a user defined name isn't specified.

See *Working with VBA Macros* for more details.

6.11 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')
```

6.12 workbook.set_calc_mode()

set_calc_mode(*mode*)

Set the Excel calculation mode for the workbook.

Parameters **mode** (*string*) – The calculation mode string

Set the calculation mode for formulas in the workbook. This is mainly of use for workbooks with slow formulas where you want to allow the user to calculate them manually.

The `mode` parameter can be:

- `auto`: The default. Excel will re-calculate formulas when a formula or a value affecting the formula changes.
- `manual`: Only re-calculate formulas when the user requires it. Generally by pressing F9.
- `auto_except_tables`: Excel will automatically re-calculate formulas except for tables.

6.13 workbook.use_zip64()

use_zip64()

Allow ZIP64 extensions when writing xlsx file zip container.

Use ZIP64 extensions when writing the xlsx file zip container and allow files greater than 4 GB.

CHAPTER

SEVEN

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   = xlsxwriter.Workbook('filename.xlsx')

worksheet1 = workbook.add_worksheet()
worksheet2 = workbook.add_worksheet()

worksheet1.write('A1', 123)

workbook.close()
```

7.1 worksheet.write()

write(*row*, *col*, **args*)

Write generic data to a worksheet cell.

Parameters

- **row** – The cell row (zero indexed).
- **col** – The cell column (zero indexed).
- ***args** – The additional args that are passed to the sub methods such as number, string and cell_format.

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_datetime()`
- `write_boolean()`
- `write_url()`

The rules for handling data in `write()` are as follows:

- Data types `float`, `int`, `long`, `decimal.Decimal` and `fractions.Fraction` are written using `write_number()`.
- Data types `datetime.datetime`, `datetime.date` `datetime.time` or `datetime.timedelta` are written using `write_datetime()` .
- None and empty strings `""` are written using `write_blank()`.
- Data type `bool` is written using `write_boolean()`.

Strings are then handled as follows:

- Strings that start with `"="` are take to match a formula and are written using `write_formula()`. This can be overridden, see below.
- Strings that match supported URL types are written using `write_url()`. This can be overridden, see below.
- When the `Workbook()` constructor `strings_to_numbers` option is `True` strings that convert to numbers using `float()` are written using `write_number()` in order to avoid Excel warnings about "Numbers Stored as Text". See the note below.
- Strings that don't match any of the above criteria are written using `write_string()`.

If none of the above types are matched the value is evaluated with `float()` to see if it corresponds to a user defined float type. If it does then it is written using `write_number()`.

Finally, if none of these rules are matched then a `TypeError` exception is raised.

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, 3.00001)          # 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:

Note: The `Workbook()` constructor option takes three optional arguments that can be used to override string handling in the `write()` function. These options are shown below with their default values:

```
xlsxwriter.Workbook(filename, {'strings_to_numbers':  False,
                               'strings_to_formulas': True,
                               'strings_to_urls':     True})
```

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 in the sub `write` methods 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.
```

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 supported. 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 it can only **display** 1000 in a cell. However, all 32,767 characters are displayed in the formula bar.

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 numeric types to the cell specified by `row` and `column`:

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

The numeric types supported are `float`, `int`, `long`, `decimal.Decimal` and `fractions.Fraction` or anything that can be converted via `float()`.

When written to an Excel file numbers are converted to 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.

Note: NAN and INF are not supported and will raise a TypeError exception.

Both row-column and A1 style notation are supported. 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.

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 supported. 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 options `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)
```

Excel stores formulas in US style formatting regardless of the Locale or Language of the Excel version. Therefore all formula names written using XlsxWriter must be in English (use the following formula translator if necessary). Also, formulas must be written with the US style separator/range operator which is a comma (not semi-colon). Therefore a formula with multiple values should be written as follows:

```
worksheet.write_formula('A1', '=SUM(1, 2, 3)')  # OK
worksheet.write_formula('A2', '=SUM(1; 2; 3)')  # NO. Error on load.
```

Excel 2010 and 2013 added functions which weren't defined in the original file specification. These functions are referred to as *future* functions. Examples of these functions are `ACOT`, `CHISQ.DIST.RT` , `CONFIDENCE.NORM`, `STDEV.P`, `STDEV.S` and `WORKDAY.INTL`. The full list is given in the MS XLSX extensions documentation on future functions.

When written using `write_formula()` these functions need to be fully qualified with the `_xlfn.` prefix as they are shown in the MS XLSX documentation link above. For example:

```
worksheet.write_formula('A1', '=_xlfn.STDEV.S(B1:B10)')
```

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)}`.

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)}')
```

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)}')
```

As shown above, both row-column and A1 style notation are supported. 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.

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)
```

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 or formatting whilst a "Blank" cell 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_boolean()

write_boolean(*row*, *col*, *boolean*[, *cell_format*])

Write a boolean value to a worksheet cell.

Parameters

- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **boolean** (*bool*) – Boolean value to write to cell.
- **cell_format** (*Format*) – Optional Format object.

The `write_boolean()` method writes a boolean value to the cell specified by `row` and `column`:

```
worksheet.write_boolean(0, 0, True)
worksheet.write_boolean('A2', False)
```

Both row-column and A1 style notation are supported. 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.

7.8 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, .time or .delta 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` `datetime.time` or `datetime.timedelta` 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 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)
```

If required, a default date format string can be set using the `Workbook()` constructor `default_date_format` option.

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

7.9 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. Defaults to blue underline.
- **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 supported. See *Working with Cell Notation* for more details.

The `cell_format` parameter is used to apply formatting to the cell. This parameter is optional. Since a hyperlink without a format doesn't look like a link the following default *Format* is used:

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

Therefore the following are equivalent:

```
link_format = workbook.add_format({'color': 'blue', 'underline': 1})
worksheet.write_url('A1', 'ftp://www.python.org/', link_format)

# Same as:
worksheet.write_url('A1', 'ftp://www.python.org/')  # Default format.
```

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

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

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

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

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

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

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

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 by passed directly to Excel.

7.10 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* and *Example: Merging Cells with a Rich String*.

7.11 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])
```

7.12 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_column('A1', data)

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

7.13 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** (*float*) – 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 above.
```

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': True})

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

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': True})
```

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.

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})
```

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})
```

7.14 worksheet.set_column()

set_column(*first_col*, *last_col*, *width*, *cell_format*, *options*)

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** (*float*) – 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 your 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.

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

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

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})
```

7.15 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 for image position, scale and url.

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,
    'url':         None,
    'tip':         None,
    'image_data':  None,
    'positioning': None,
}
```

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})
```

The `url` parameter can used to add a hyperlink/url to the image. The `tip` parameter gives an option mouseover tooltip for images with hyperlinks:

```
worksheet.insert_image('B4', 'python.png', {'url': 'http://python.org'})
```

See also `write_url()` for details on supported URIs.

The `image_data` parameter is used to add an in-memory byte stream in `io.BytesIO` format:

```
worksheet.insert_image('B5', 'python.png', {'image_data': image_data})
```

This is generally used for inserting images from URLs:

```
url = 'http://python.org/logo.png'
image_data = io.BytesIO(urllib2.urlopen(url).read())

worksheet.insert_image('B5', url, {'image_data': image_data})
```

When using the `image_data` parameter a filename must still be passed to `insert_image()` since it is required by Excel. In the previous example the filename is extracted from the URL string. See also *Example: Inserting images from a URL or byte stream into a worksheet*.

The `positioning` parameter can be used to control the object positioning of the image:

```
worksheet.insert_image('B3', 'python.png', {'positioning': 1})
```

Where `positioning` has the following allowable values:

1. Move and size with cells.
2. Move but don't size with cells (the default).
3. Don't move or size with cells.

Note: The scaling of a image may be affected if is crosses a row that has its default height changed due to a font that is larger than the default font size or that has text wrapping turned on. To avoid this you should explicitly set the height of the row using `set_row()` if it crosses an inserted image.

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*.

7.16 worksheet.insert_chart()

insert_chart(*row*, *col*, *chart*[, *options*])

Write a string to a worksheet cell.

Parameters

- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **chart** – A chart object.
- **options** (*dict*) – Optional parameters to position and scale the chart.

This method can be used to insert a chart into a worksheet. A chart object is created via the Workbook `add_chart()` method where the chart type is specified:

```
chart = workbook.add_chart({type, 'column'})
```

It is then inserted into a worksheet as an embedded chart:

```
worksheet.insert_chart('B5', chart)
```

Note: A chart can only be inserted into a worksheet once. If several similar charts are required then each one must be created separately with `add_chart()`.

See *The Chart Class*, *Working with Charts* and *Chart Examples*.

The `insert_chart()` method takes optional parameters in a dictionary to position and scale the chart. 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:

```
worksheet.insert_chart('B5', chart, {'x_offset': 25, 'y_offset': 10})
```

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

```
worksheet.insert_chart('B5', chart, {'x_scale': 0.5, 'y_scale': 0.5})
```

These properties can also be set via the Chart `set_size()` method.

Note: The scaling of a chart may be affected if is crosses a row that has its default height changed due to a font that is larger than the default font size or that has text wrapping turned on. To avoid this you should explicitly set the height of the row using `set_row()` if it crosses an inserted chart.

7.17 worksheet.insert_button()

insert_button(*row*, *col*[, *options*])

Insert a VBA button control on a worksheet.

Parameters

- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **options** (*dict*) – Optional parameters to position and scale the button.

The `insert_button()` method can be used to insert an Excel form button into a worksheet.

This method is generally only useful when used in conjunction with the Workbook `add_vba_project()` method to tie the button to a macro from an embedded VBA project:

```
# Add the VBA project binary.
workbook.add_vba_project('./vbaProject.bin')

# Add a button tied to a macro in the VBA project.
worksheet.insert_button('B3',{'macro':   'say_hello',
                              'caption': 'Press Me'})
```

See *Working with VBA Macros* and *Example: Adding a VBA macro to a Workbook* for more details.

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

```
{
    'macro':    None,
    'caption':  'Button 1',
    'width':    64,
    'height':   20.
    'x_offset': 0,
    'y_offset': 0,
    'x_scale':  1,
    'y_scale':  1,
}
```

The `macro` option is used to set the macro that the button will invoke when the user clicks on it. The macro should be included using the Workbook `add_vba_project()` method shown above.

The `caption` is used to set the caption on the button. The default is `Button n` where n is the button number.

The default button `width` is 64 pixels which is the width of a default cell and the default button

`height` is 20 pixels which is the height of a default cell.

The offset and scale options are the same as for `insert_chart()`, see above.

7.18 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*.

7.19 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_* 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*.

7.20 worksheet.add_table()

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

Add an Excel table to a 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.
- **options** (*dict*) – Table formatting options. (Optional)

The `add_table()` method is used to group a range of cells into an Excel Table:

```
worksheet.add_table('B3:F7', { ... })
```

This method contains a lot of parameters and is described in *Working with Worksheet Tables*.

See also *Example: Worksheet Tables*.

7.21 worksheet.add_sparkline()

add_sparkline(*row*, *col*, *options*)

Add sparklines to a worksheet.

Parameters

- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **options** (*dict*) – Sparkline formatting options.

Sparklines are small charts that fit in a single cell and are used to show trends in data.

The `add_sparkline()` worksheet method is used to add sparklines to a cell or a range of cells:

```
worksheet.add_sparkline('F1', {'range': 'A1:E1'})
```

This method contains a lot of parameters and is described in detail in *Working with Sparklines*.

See also *Example: Sparklines (Simple)* and *Example: Sparklines (Advanced)*.

Note: Sparklines are a feature of Excel 2010+ only. You can write them to an XLSX file that can be read by Excel 2007 but they won't be displayed.

7.22 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)* .

7.23 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)* .

7.24 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)* .

7.25 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.

7.26 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.

7.27 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.

7.28 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()
```

See *Example: Hiding Worksheets* for more details.

7.29 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 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.

7.30 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)
```

See *Example: Merging Cells* for more details.

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 with a Rich String*.

7.31 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.

7.32 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.

7.33 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.

7.34 worksheet.set_selection()

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

Set the selected cell or cells in a 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 `set_selection()` method can be used to specify which cell or range of cells is selected in a worksheet. The most common requirement is to select a single cell, in which case the `first_` and `last_` parameters should be the same.

The active cell within a selected range is determined by the order in which `first_` and `last_` are specified.

Examples:

```
worksheet1.set_selection(3, 3, 3, 3)  # 1. Cell D4.
worksheet2.set_selection(3, 3, 6, 6)  # 2. Cells D4 to G7.
worksheet3.set_selection(6, 6, 3, 3)  # 3. Cells G7 to D4.
worksheet4.set_selection('D4')        # Same as 1.
worksheet5.set_selection('D4:G7')     # Same as 2.
worksheet6.set_selection('G7:D4')     # Same as 3.
```

As shown above, both row-column and A1 style notation are supported. See *Working with Cell Notation* for more details. The default cell selection is `(0, 0), 'A1'`.

7.35 worksheet.freeze_panes()

freeze_panes(*row*, *col*[, *top_row*, *left_col*])

Create worksheet panes and mark them as frozen.

Parameters

- **row** (*int*) – The cell row (zero indexed).
- **col** (*int*) – The cell column (zero indexed).
- **top_row** (*int*) – Topmost visible row in scrolling region of pane.
- **left_col** (*int*) – Leftmost visible row in scrolling region of pane.

This `freeze_panes` method can be used to divide a worksheet into horizontal or vertical regions known as panes and to "freeze" these panes so that the splitter bars are not visible.

The parameters `row` and `col` are used to specify the location of the split. It should be noted that the split is specified at the top or left of a cell and that the method uses zero based indexing. Therefore to freeze the first row of a worksheet it is necessary to specify the split at row 2 (which is 1 as the zero-based index).

You can set one of the `row` and `col` parameters as zero if you do not want either a vertical or horizontal split.

Examples:

```
worksheet.freeze_panes(1, 0)  # Freeze the first row.
worksheet.freeze_panes('A2')  # Same using A1 notation.
worksheet.freeze_panes(0, 1)  # Freeze the first column.
worksheet.freeze_panes('B1')  # Same using A1 notation.
worksheet.freeze_panes(1, 2)  # Freeze first row and first 2 columns.
worksheet.freeze_panes('C2')  # Same using A1 notation.
```

The parameters `top_row` and `left_col` are optional. They are used to specify the top-most or left-most visible row or column in the scrolling region of the panes. For example to freeze the first row and to have the scrolling region begin at row twenty:

```
worksheet.freeze_panes(1, 0, 20, 0)
```

You cannot use A1 notation for the `top_row` and `left_col` parameters.

See *Example: Freeze Panes and Split Panes* for more details.

7.36 worksheet.split_panes()

split_panes(*x*, *y*[, *top_row*, *left_col*])

Create worksheet panes and mark them as split.

Parameters

- **x** (*float*) – The position for the vertical split.
- **y** (*float*) – The position for the horizontal split.
- **top_row** (*int*) – Topmost visible row in scrolling region of pane.
- **left_col** (*int*) – Leftmost visible row in scrolling region of pane.

The `split_panes` method can be used to divide a worksheet into horizontal or vertical regions known as panes. This method is different from the `freeze_panes()` method in that the splits between the panes will be visible to the user and each pane will have its own scroll bars.

The parameters `y` and `x` are used to specify the vertical and horizontal position of the split. The units for `y` and `x` are the same as those used by Excel to specify row height and column width. However, the vertical and horizontal units are different from each other. Therefore you must specify the `y` and `x` parameters in terms of the row heights and column widths that you have set or the default values which are `15` for a row and `8.43` for a column.

You can set one of the `y` and `x` parameters as zero if you do not want either a vertical or horizontal split. The parameters `top_row` and `left_col` are optional. They are used to specify the top-most or left-most visible row or column in the bottom-right pane.

Example:

```
worksheet.split_panes(15, 0)     # First row.
worksheet.split_panes(0, 8.43)   # First column.
worksheet.split_panes(15, 8.43)  # First row and column.
```

You cannot use A1 notation with this method.

See *Example: Freeze Panes and Split Panes* for more details.

7.37 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()`.

7.38 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.

7.39 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()
```

7.40 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 *Working with Colors*.

See *Example: Setting Worksheet Tab Colours* for more details.

7.41 worksheet.protect()

protect()

Protect elements of a worksheet from modification.

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':               False,
    'scenarios':             False,
    'format_cells':          False,
    'format_columns':        False,
    'format_rows':           False,
    'insert_columns':        False,
    'insert_rows':           False,
    'insert_hyperlinks':     False,
    'delete_columns':        False,
    'delete_rows':           False,
    'select_locked_cells':   True,
    'sort':                  False,
```

```
        'autofilter':            False,
        'pivot_tables':          False,
        'select_unlocked_cells': True,
    }
```

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

```
worksheet.protect('acb123', { '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 do 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.

7.42 worksheet.set_default_row()

`set_default_row`(*height*, *hide_unused_rows*)

Set the default row properties.

Parameters

- **height** (*float*) – Default height. Optional, defaults to 15.
- **hide_unused_rows** (*bool*) – Hide unused rows. Optional, defaults to False.

The `set_default_row()` method is used to set the limited number of default row properties allowed by Excel which are the default height and the option to hide unused rows. These parameters are an optimisation used by Excel to set row properties without generating a very large file with an entry for each row.

To set the default row height:

```
worksheet.set_default_row(24)
```

To hide unused rows:

```
worksheet.set_default_row(hide_unused_rows=True)
```

See *Example: Hiding Rows and Columns* for more details.

7.43 worksheet.outline_settings()

`outline_settings`(*visible*, *symbols_below*, *symbols_right*, *auto_style*)

Control outline settings.

Parameters

- **visible** (*bool*) – Outlines are visible. Optional, defaults to True.
- **symbols_below** (*bool*) – Show row outline symbols below the outline bar. Optional, defaults to True.
- **symbols_right** (*bool*) – Show column outline symbols to the right of the outline bar. Optional, defaults to True.
- **auto_style** (*bool*) – Use Automatic style. Optional, defaults to False.

The `outline_settings()` method is used to control the appearance of outlines in Excel. Outlines are described in *Working with Outlines and Grouping*:

```
worksheet1.outline_settings(False, False, False, True)
```

The `'visible'` parameter is used to control whether or not outlines are visible. Setting this parameter to `False` will cause all outlines on the worksheet to be hidden. They can be un-hidden in Excel by means of the "Show Outline Symbols" command button. The default setting is `True` for visible outlines.

The `'symbols_below'` parameter is used to control whether the row outline symbol will appear above or below the outline level bar. The default setting is `True` for symbols to appear below the outline level bar.

The `'symbols_right'` parameter is used to control whether the column outline symbol will appear to the left or the right of the outline level bar. The default setting is `True` for symbols to appear to the right of the outline level bar.

The `'auto_style'` parameter is used to control whether the automatic outline generator in Excel uses automatic styles when creating an outline. This has no effect on a file generated by `XlsxWriter` but it does have an effect on how the worksheet behaves after it is created. The default setting is `False` for "Automatic Styles" to be turned off.

The default settings for all of these parameters correspond to Excel's default parameters.

The worksheet parameters controlled by `outline_settings()` are rarely used.

7.44 worksheet.set_vba_name()

`set_vba_name`(*name*)

Set the VBA name for the worksheet.

Parameters name (*string*) – The VBA name for the worksheet.

The `set_vba_name()` method can be used to set the VBA codename for the worksheet (there is a similar method for the workbook VBA name). This is sometimes required when a vbaProject macro included via `add_vba_project()` refers to the worksheet. The default Excel VBA name of `Sheet1`, etc., is used if a user defined name isn't specified.

See *Working with VBA Macros* for more details.

CHAPTER

EIGHT

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	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
30	Envelope C5	162 x 229 mm
31	Envelope C6	114 x 162 mm
32	Envelope C65	114 x 229 mm

Continued on next page

Table 8.1 – continued from previous page

Index	Paper format	Paper size
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.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='',] options]]*)

Set the printed page header caption and options.

Parameters

- **header** (*string*) – Header string with Excel control characters.
- **options** (*dict*) – Header options.

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
&[Picture]	Images	Image placeholder
&G		Same as &[Picture]

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:

```
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                       |
|                                                               |
```

Images can be inserted using the `options` shown below. Each image must have a placeholder in header string using the `&[Picture]` or &G control characters:

```
worksheet.set_header('&L&G', {'image_left': 'logo.jpg'})
```

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 = ""
```

Alternatively you can inspect the header and footer strings in an Excel file by unzipping it and grepping the XML sub-files. The following shows how to do that using libxml's xmllint to format the XML for clarity:

```
$ unzip myfile.xlsm -d myfile
$ xmllint --format find myfile -name "*.xml" | xargs | egrep "Header|Footer"

  <headerFooter scaleWithDoc="0">
    <oddHeader>&amp;L&amp;P</oddHeader>
  </headerFooter>
```

Note that in this case you need to unescape the Html. In the above example the header string would be:

```
'&L&P'
```

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

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

The available optons are:

- `margin`: (float) Header margin in inches. Defaults to 0.3 inch.
- `image_left`: (string) The path to the image. Needs &G placeholder.
- `image_center`: (string) Same as above.
- `image_right`: (string) Same as above.
- `image_data_left`: (BytesIO) A byte stream of the image data.
- `image_data_center`: (BytesIO) Same as above.
- `image_data_right`: (BytesIO) Same as above.
- `scale_with_doc`: (boolean) Scale header with document. Defaults to True.
- `align_with_margins`: (boolean) Align header to margins. Defaults to True.

As with the other margins the `margin` value should be in inches. The default header and footer margin is 0.3 inch. It can be changed as follows:

```
worksheet.set_header('&CHello', {'margin': 0.75})
```

The header and footer margins are independent of, and should not be confused with, the top and bottom worksheet margins.

The image options must have an accompanying `&[Picture]` or `&G` control character in the header string:

```
worksheet.set_header('&L&[Picture]&C&[Picture]&R&[Picture]',
                     {'image_left':   'red.jpg',
                      'image_center': 'blue.jpg',
                      'image_right':  'yellow.jpg'})
```

The `image_data_` parameters are used to add an in-memory byte stream in `io.BytesIO` format:

```
image_file = open('logo.jpg', 'rb')
image_data = BytesIO(image_file.read())

worksheet.set_header('&L&G',
                     {'image_left': 'logo.jpg',
                      'image_data_left': image_data})
```

When using the `image_data_` parameters a filename must still be passed to to the equivalent `image_` parameter since it is required by Excel. See also `insert_image()` for details on handling images from byte streams.

Note, Excel does not allow header or footer strings longer than 255 characters, including control 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='',] options]]*)

Set the printed page footer caption and options.

Parameters

- **footer** (*string*) – Footer string with Excel control characters.
- **options** (*dict*) – Footer options.

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.

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.
```

In order to set a row or column range you must specify the entire range:

```
worksheet3.print_area('A1:H1048576')  # Same as A:H.
```

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 in Excel only one of these options to 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:

```
# Start print from page 2.
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 Excel only allows one of these options to 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 page break 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.

The `set_h_pagebreaks()` method takes a list of one or more page breaks:

```
worksheet1.set_v_pagebreaks([20])
worksheet2.set_v_pagebreaks([20, 40, 60, 80, 100])
```

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:

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

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 page break columns.

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

CHAPTER

NINE

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 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 *Working with 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('00000')
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. The listed date and currency formats may also vary depending on system settings.

Note: The dollar sign in the above format 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:

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

format1.set_indent(1)
format2.set_indent(2)

worksheet.write('A1', "This text is indented 1 level", format1)
worksheet.write('A2', "This text is indented 2 levels", format2)
```

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 *Working with 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 *Working with 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 *Working with 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.

9.32 format.set_diag_border()

set_diag_border(*style*)

Set the diagonal cell border style.

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

Set the style for a diagonal border. The *style* is the same as those used in `set_border()`.

See *Example: Diagonal borders in cells*.

9.33 format.set_diag_type()

set_diag_type(*style*)

Set the diagonal cell border type.

Parameters **style** (*int*) – Border type, 1-3. No default.

Set the type of the diagonal border. The *style* should be one of the following values:

1. From bottom left to top right.
2. From top left to bottom right.
3. Same as type 1 and 2 combined.

9.34 format.set_diag_color()

set_diag_color(*color*)

Set the color of the diagonal cell border.

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

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

CHAPTER

TEN

THE CHART CLASS

The `Chart` module is a base class for modules that implement charts in XlsxWriter. The information in this section is applicable to all of the available chart subclasses, such as Area, Bar, Column, Doughnut, Line, Pie, Scatter, Stock and Radar.

A chart object is created via the Workbook `add_chart()` method where the chart type is specified:

```
chart = workbook.add_chart({type, 'column'})
```

It is then inserted into a worksheet as an embedded chart using the `insert_chart()` Worksheet method:

```
worksheet.insert_chart('A7', chart)
```

Or it can be set in a chartsheet using the `set_chart()` Chartsheet method:

```
chartsheet = workbook.add_chartsheet()
# ...
chartsheet.set_chart(chart)
```

The following is a small working example or adding an embedded chart:

```
import xlsxwriter

workbook = xlsxwriter.Workbook('chart.xlsx')
worksheet = workbook.add_worksheet()

# Create a new Chart object.
chart = workbook.add_chart({'type': 'column'})

# Write some data to add to plot on the chart.
data = [
    [1, 2, 3, 4, 5],
    [2, 4, 6, 8, 10],
    [3, 6, 9, 12, 15],
]

worksheet.write_column('A1', data[0])
worksheet.write_column('B1', data[1])
worksheet.write_column('C1', data[2])
```

```
# Configure the chart. In simplest case we add one or more data series.
chart.add_series({'values': '=Sheet1!$A$1:$A$5'})
chart.add_series({'values': '=Sheet1!$B$1:$B$5'})
chart.add_series({'values': '=Sheet1!$C$1:$C$5'})

# Insert the chart into the worksheet.
worksheet.insert_chart('A7', chart)

workbook.close()
```

The supported chart types are:

- `area`: Creates an Area (filled line) style chart.
- `bar`: Creates a Bar style (transposed histogram) chart.
- `column`: Creates a column style (histogram) chart.
- `line`: Creates a Line style chart.
- `pie`: Creates a Pie style chart.
- `doughnut`: Creates a Doughnut style chart.
- `scatter`: Creates a Scatter style chart.
- `stock`: Creates a Stock style chart.
- `radar`: Creates a Radar style chart.

Chart subtypes are also supported for some chart types:

```
workbook.add_chart({'type': 'bar', 'subtype': 'stacked'})
```

The available subtypes are:

```
area
    stacked
    percent_stacked
```

```
bar
    stacked
    percent_stacked

column
    stacked
    percent_stacked

scatter
    straight_with_markers
    straight
    smooth_with_markers
    smooth

radar
    with_markers
    filled
```

Methods that are common to all chart types are documented below. See *Working with Charts* for chart specific information.

10.1 chart.add_series()

add_series(*options*)

Add a data series to a chart.

Parameters options (*dict*) – A dictionary of chart series options.

In Excel a chart **series** is a collection of information that define which data is plotted such as values, axis labels and formatting.

For an XlsxWriter chart object the `add_series()` method is used to set the properties for a series:

```
chart.add_series({
    'categories': '=Sheet1!$A$1:$A$5',
    'values':     '=Sheet1!$B$1:$B$5',
    'line':       {'color': 'red'},
})

# Or using a list of values instead of category/value formulas:
#     [sheetname, first_row, first_col, last_row, last_col]
chart.add_series({
    'categories': ['Sheet1', 0, 0, 4, 0],
    'values':     ['Sheet1', 0, 1, 4, 1],
    'line':       {'color': 'red'},
})
```

As shown above the `categories` and `values` can take either a range formula such as `=Sheet1!$A$2:$A$7` or, more usefully when generating the range programmatically, a list with zero indexed row/column values.

The series options that can be set are:

- `values`: This is the most important property of a series and is the only mandatory option for every chart object. This option links the chart with the worksheet data that it displays. The data range can be set using a formula as shown in the first example above or using a list of values as shown in the second example.
- `categories`: This sets the chart category labels. The category is more or less the same as the X axis. In most chart types the `categories` property is optional and the chart will just assume a sequential series from `1..n`.
- `name`: Set the name for the series. The name is displayed in the chart legend and in the formula bar. The name property is optional and if it isn't supplied it will default to `Series 1..n`. The name can also be a formula such as `=Sheet1!$A$1` or a list with a sheetname, row and column such as `['Sheet1', 0, 0]`.
- `line`: Set the properties of the series line type such as colour and width. See *Chart formatting: Line*.
- `border`: Set the border properties of the series such as colour and style. See *Chart formatting: Border*.
- `fill`: Set the fill properties of the series such as colour. See *Chart formatting: Fill*.
- `marker`: Set the properties of the series marker such as style and colour. See *Chart series option: Marker*.
- `trendline`: Set the properties of the series trendline such as linear, polynomial and moving average types. See *Chart series option: Trendline*.
- `smooth`: Set the smooth property of a line series.
- `y_error_bars`: Set vertical error bounds for a chart series. See *Chart series option: Error Bars*.
- `x_error_bars`: Set horizontal error bounds for a chart series. See *Chart series option: Error Bars*.
- `data_labels`: Set data labels for the series. See *Chart series option: Data Labels*.
- `points`: Set properties for individual points in a series. See *Chart series option: Points*.
- `invert_if_negative`: Invert the fill colour for negative values. Usually only applicable to column and bar charts.
- `overlap`: Set the overlap between series in a Bar/Column chart. The range is +/- 100. The default is 0:

  ```
  chart.add_series({
      'categories': '=Sheet1!$A$1:$A$5',
      'values':     '=Sheet1!$B$1:$B$5',
      'overlap':    10,
  })
  ```

  Note, it is only necessary to apply the `overlap` property to one series in the chart.

- gap: Set the gap between series in a Bar/Column chart. The range is 0 to 500. The default is 150:

  ```
  chart.add_series({
      'categories': '=Sheet1!$A$1:$A$5',
      'values':     '=Sheet1!$B$1:$B$5',
      'gap':        200,
  })
  ```

  Note, it is only necessary to apply the gap property to one series in the chart.

More than one series can be added to a chart. In fact, some chart types such as stock require it. The series numbering and order in the Excel chart will be the same as the order in which they are added in XlsxWriter.

It is also possible to specify non-contiguous ranges:

```
chart.add_series({
    'categories': '=(Sheet1!$A$1:$A$9,Sheet1!$A$14:$A$25)',
    'values':     '=(Sheet1!$B$1:$B$9,Sheet1!$B$14:$B$25)',
})
```

10.2 chart.set_x_axis()

set_x_axis(*options*)

Set the chart X axis options.

Parameters options (*dict*) – A dictionary of axis options.

The set_x_axis() method is used to set properties of the X axis:

```
chart.set_x_axis({
    'name': 'Earnings per Quarter',
    'name_font': {'size': 14, 'bold': True},
    'num_font':  {'italic': True },
})
```

The options that can be set are:

```
name
name_font
name_layout
num_font
num_format
line
fill
min
max
minor_unit
major_unit
interval_unit
crossing
position_axis
reverse
log_base
label_position
major_gridlines
minor_gridlines
visible
date_axis
text_axis
minor_unit_type
major_unit_type
```

These options are explained below. Some properties are only applicable to **value**, **category** or **date** axes (this is noted in each case). See *Chart Value and Category Axes* for an explanation of Excel's distinction between the axis types.

- name: Set the name (also known as title or caption) for the axis. The name is displayed below the X axis. (Applicable to category, date and value axes.):

```
chart.set_x_axis({'name': 'Earnings per Quarter'})
```

This property is optional. The default is to have no axis name.

The name can also be a formula such as `=Sheet1!$A$1` or a list with a sheetname, row and column such as `['Sheet1', 0, 0]`.

- `name_font`: Set the font properties for the axis name. (Applicable to category, date and value axes.):

```
chart.set_x_axis({'name_font': {'bold': True, 'italic': True}})
```

  See the *Chart Fonts* section for more details on font properties.

- `name_layout`: Set the `(x, y)` position of the axis caption in chart relative units. (Applicable to category, date and value axes.):

```
chart.set_x_axis({
    'name': 'X axis',
    'name_layout': {
        'x': 0.34,
        'y': 0.85,
    }
})
```

  See the *Chart Layout* section for more details.

- `num_font`: Set the font properties for the axis numbers. (Applicable to category, date and value axes.):

```
chart.set_x_axis({'name_font': {'bold': True, 'italic': True}})
```

  See the *Chart Fonts* section for more details on font properties.

- `num_format`: Set the number format for the axis. (Applicable to category, date and value axes.):

```
chart.set_x_axis({'num_format': '#,##0.00'})
chart.set_y_axis({'num_format': '0.00%'})
```

  The number format is similar to the Worksheet Cell Format `num_format` apart from the fact that a format index cannot be used. An explicit format string must be used as shown above. See `set_num_format()` for more information.

- `line`: Set the properties of the axis line type such as colour and width. See *Chart formatting: Line*:

```
chart.set_x_axis({'line': {'none': True}})
```

- `fill`: Set the fill properties of the axis such as colour. See *Chart formatting: Fill*. Note, in Excel the axis fill is applied to the area of the numbers of the axis and not to the area of the axis bounding box. That background is set from the chartarea fill.

- `min`: Set the minimum value for the axis range. (Applicable to value and date axes only.):

```
chart.set_x_axis({'min': 3, 'max': 6})
```

- `max`: Set the maximum value for the axis range. (Applicable to value and date axes only.)
- `minor_unit`: Set the increment of the minor units in the axis range. (Applicable to value and date axes only.):

  ```
  chart.set_x_axis({'minor_unit': 0.4, 'major_unit': 2})
  ```

- `major_unit`: Set the increment of the major units in the axis range. (Applicable to value and date axes only.)
- `interval_unit`: Set the interval unit for a category axis. Should be an integer value. (Applicable to category axes only.):

  ```
  chart.set_x_axis({'interval_unit': 2})
  ```

- `crossing`: Set the position where the y axis will cross the x axis. (Applicable to all axes.)

  The `crossing` value can either be the string `'max'` to set the crossing at the maximum axis value or a numeric value:

  ```
  chart.set_x_axis({'crossing': 3})
  chart.set_y_axis({'crossing': 'max'})
  ```

  For category axes the numeric value must be an integer to represent the category number that the axis crosses at. For value and date axes it can have any value associated with the axis. See also *Chart Value and Category Axes*.

  If crossing is omitted (the default) the crossing will be set automatically by Excel based on the chart data.

- `position_axis`: Position the axis on or between the axis tick marks. (Applicable to category axes only.)

  There are two allowable values `on_tick` and `between`:

  ```
  chart.set_x_axis({'position_axis': 'on_tick'})
  chart.set_x_axis({'position_axis': 'between'})
  ```

- `reverse`: Reverse the order of the axis categories or values. (Applicable to category, date and value axes.):

```
chart.set_y_axis({'reverse': True})
```

- `log_base`: Set the log base of the axis range. (Applicable to value axes only.):

```
chart.set_y_axis({'log_base': 10})
```

- `label_position`: Set the "Axis labels" position for the axis. The following positions are available:

```
next_to (the default)
high
low
none
```

For example:

```
chart.set_x_axis({'label_position': 'high'})
chart.set_y_axis({'label_position': 'low'})
```

- `major_gridlines`: Configure the major gridlines for the axis. The available properties are:

```
visible
line
```

For example:

```
chart.set_x_axis({
    'major_gridlines': {
        'visible': True,
        'line': {'width': 1.25, 'dash_type': 'dash'}
    },
})
```

The `visible` property is usually on for the X axis but it depends on the type of chart.

The `line` property sets the gridline properties such as colour and width. See *Chart Formatting*.

- `minor_gridlines`: This takes the same options as `major_gridlines` above.

  The minor gridline `visible` property is off by default for all chart types.

- `visible`: Configure the visibility of the axis:

  ```
  chart.set_y_axis({'visible': False})
  ```

  Axes are visible by default.

- `date_axis`: This option is used to treat a category axis with date or time data as a Date Axis. (Applicable to date category axes only.):

  ```
  chart.set_x_axis({'date_axis': True})
  ```

  This option also allows you to set `max` and `min` values for a category axis which isn't allowed by Excel for non-date category axes.

  See *Date Category Axes* for more details.

- `text_axis`: This option is used to treat a category axis explicitly as a Text Axis. (Applicable to category axes only.):

  ```
  chart.set_x_axis({'text_axis': True})
  ```

- `minor_unit_type`: For `date_axis` axes, see above, this option is used to set the type of the minor units. (Applicable to date category axes only.):

  ```
  chart.set_x_axis({
      'date_axis': True,
      'minor_unit': 4,
      'minor_unit_type': 'months',
  })
  ```

- `major_unit_type`: Same as `minor_unit_type`, see above, but for major axes unit types.

10.3 chart.set_y_axis()

set_y_axis(*options*)

Set the chart Y axis options.

Parameters options (*dict*) – A dictionary of axis options.

The `set_y_axis()` method is used to set properties of the Y axis.

The properties that can be set are the same as for `set_x_axis`, see above.

10.4 chart.set_x2_axis()

set_x2_axis(*options*)

Set the chart secondary X axis options.

Parameters options (*dict*) – A dictionary of axis options.

The `set_x2_axis()` method is used to set properties of the secondary X axis, see `chart_secondary_axes()`.

The properties that can be set are the same as for `set_x_axis`, see above.

The default properties for this axis are:

```
'label_position': 'none',
'crossing':       'max',
'visible':        False,
```

10.5 chart.set_y2_axis()

set_y2_axis(*options*)

Set the chart secondary Y axis options.

Parameters options (*dict*) – A dictionary of axis options.

The `set_y2_axis()` method is used to set properties of the secondary Y axis, see `chart_secondary_axes()`.

The properties that can be set are the same as for `set_x_axis`, see above.

The default properties for this axis are:

```
'major_gridlines': {'visible': True}
```

10.6 chart.set_size()

set_size(*options*)

Set the size or scale of the chart.

Parameters options (*dict*) – A dictionary of chart size options.

The `set_size()` method is used to set the dimensions of the chart. The size properties that can be set are:

```
width
height
x_scale
y_scale
x_offset
y_offset
```

The `width` and `height` are in pixels. The default chart width x height is 480 x 288 pixels. The size of the chart can be modified by setting the `width` and `height` or by setting the `x_scale` and `y_scale`:

```
chart.set_size({'width': 720, 'height': 576})
# Same as:
chart.set_size({'x_scale': 1.5, 'y_scale': 2})
```

The `x_offset` and `y_offset` position the top left corner of the chart in the cell that it is inserted into.

Note: the `x_offset` and `y_offset` parameters can also be set via the `insert_chart()` method:

```
worksheet.insert_chart('E2', chart, {'x_offset': 25, 'y_offset': 10})
```

10.7 chart.set_title()

set_title(*options*)

Set the chart title options.

Parameters options (*dict*) – A dictionary of chart size options.

The `set_title()` method is used to set properties of the chart title:

```
chart.set_title({'name': 'Year End Results'})
```

The properties that can be set are:

- `name`: Set the name (title) for the chart. The name is displayed above the chart. The name can also be a formula such as `=Sheet1!$A$1` or a list with a sheetname, row and column such as `['Sheet1', 0, 0]`. The name property is optional. The default is to have no chart title.
- `name_font`: Set the font properties for the chart title. See *Chart Fonts*.
- `overlay`: Allow the title to be overlaid on the chart. Generally used with the layout property below.
- `layout`: Set the `(x, y)` position of the title in chart relative units:

  ```
  chart.set_title({
      'name': 'Title',
      'overlay': True,
      'layout': {
          'x': 0.42,
          'y': 0.14,
      }
  })
  ```

  See the *Chart Layout* section for more details.
- `none`: By default Excel adds an automatic chart title to charts with a single series and a user defined series name. The `none` option turns this default title off. It also turns off all other `set_title()` options:

  ```
  chart.set_title({'none': True})
  ```

10.8 chart.set_legend()

`set_legend`(*options*)

Set the chart legend options.

Parameters options (*dict*) – A dictionary of chart legend options.

The `set_legend()` method is used to set properties of the chart legend. For example it can be used to turn off the default chart legend:

```
chart.set_legend({'none': True})
```

The options that can be set are:

```
none
position
layout
font
delete_series
```

- `none`: In Excel chart legends are on by default. The `none` option turns off the chart legend:

  ```
  chart.set_legend({'none': True})
  ```

  For backward compatibility, it is also possible to turn off the legend via the `position` property:

  ```
  chart.set_legend({'position': 'none'})
  ```

- `position`: Set the position of the chart legend:

  ```
  chart.set_legend({'position': 'bottom'})
  ```

The default legend position is `right`. The available positions are:

```
top
bottom
left
right
overlay_left
overlay_right
none
```

- `layout`: Set the `(x, y)` position of the legend in chart relative units:

```
chart.set_legend({
    'layout': {
        'x':      0.80,
        'y':      0.37,
        'width':  0.12,
        'height': 0.25,
    }
})
```

  See the *Chart Layout* section for more details.

- `font`: Set the font properties of the chart legend:

```
chart.set_legend({'font': {'size': 9, 'bold': 1}})
```

  See the *Chart Fonts* section for more details on font properties.

- `delete_series`: This allows you to remove one or more series from the the legend (the series will still display on the chart). This property takes a list as an argument and the series are zero indexed:

```
# Delete/hide series index 0 and 2 from the legend.
chart.set_legend({'delete_series': [0, 2]})
```

10.9 chart.set_chartarea()

set_chartarea(*options*)

Set the chart area options.

Parameters options (*dict*) – A dictionary of chart area options.

The `set_chartarea()` method is used to set the properties of the chart area. In Excel the chart area is the background area behind the chart:

```
chart.set_chartarea({
    'border': {'none': True},
    'fill':   {'color': 'red'}
})
```

The properties that can be set are:

- `border`: Set the border properties of the chartarea such as colour and style. See *Chart Formatting*.
- `fill`: Set the fill properties of the chartarea such as colour. See *Chart Formatting*.

10.10 chart.set_plotarea()

set_plotarea(*options*)

Set the plot area options.

Parameters options (*dict*) – A dictionary of plot area options.

The `set_plotarea()` method is used to set properties of the plot area of a chart. In Excel the plot area is the area between the axes on which the chart series are plotted:

```
chart.set_plotarea({
    'border': {'color': 'red', 'width': 2, 'dash_type': 'dash'},
    'fill':   {'color': '#FFFFC2'}
})
```

The properties that can be set are:

- `border`: Set the border properties of the plotarea such as colour and style. See *Chart Formatting*.
- `fill`: Set the fill properties of the plotarea such as colour. See *Chart Formatting*.
- `layout`: Set the `(x, y)` position of the plotarea in chart relative units:

```
chart.set_plotarea({
    'layout': {
        'x':      0.13,
        'y':      0.26,
        'width':  0.73,
        'height': 0.57,
```

```
        }
    })
```

See the *Chart Layout* section for more details.

10.11 chart.set_style()

set_style(*style_id*)

Set the chart style type.

Parameters **style_id** (*int*) – An index representing the chart style.

The `set_style()` method is used to set the style of the chart to one of the 48 built-in styles available on the 'Design' tab in Excel:

```
chart.set_style(37)
```

The style index number is counted from 1 on the top left. The default style is 2.

Note: In Excel 2013 the Styles section of the 'Design' tab in Excel shows what were referred to as 'Layouts' in previous versions of Excel. These layouts are not defined in the file format. They are a collection of modifications to the base chart type. They can be replicated using the XlsxWriter Chart API but they cannot be defined by the `set_style()` method.

10.12 chart.set_table()

set_table(*options*)

Set properties for an axis data table.

Parameters **options** (*dict*) – A dictionary of axis table options.

The `set_table()` method adds a data table below the horizontal axis with the data used to plot the chart:

```
chart.set_table()
```

The available options, with default values are:

```
'horizontal': True   # Display vertical lines in the table.
'vertical':   True   # Display horizontal lines in the table.
'outline':    True   # Display an outline in the table.
'show_keys':  False  # Show the legend keys with the table data.

chart.set_table({'show_keys': True})
```

The data table can only be shown with Bar, Column, Line, Area and stock charts.

10.13 chart.set_up_down_bars()

set_up_down_bars(*options*)

Set properties for the chart up-down bars.

Parameters **options** (*dict*) – A dictionary of options.

The `set_up_down_bars()` method adds Up-Down bars to Line charts to indicate the difference between the first and last data series:

```
chart.set_up_down_bars()
```

It is possible to format the up and down bars to add `fill` and `border` properties if required. See *Chart Formatting*:

```
chart.set_up_down_bars({
    'up': {
        'fill':   {'color': '#00B050'},
```

```
            'border': {'color': 'black'}
        },
        'down': {
            'fill':   {'color': 'red'},
            'border': {'color': 'black'},
        },
    })
```

Up-down bars can only be applied to Line charts and to Stock charts (by default).

10.14 chart.set_drop_lines()

set_drop_lines(*options*)

Set properties for the chart drop lines.

Parameters options (*dict*) – A dictionary of options.

The `set_drop_lines()` method adds Drop Lines to charts to show the Category value of points in the data:

```
chart.set_drop_lines()
```

It is possible to format the Drop Line `line` properties if required. See *Chart Formatting*:

```
chart.set_drop_lines({'line': {'color': 'red',
                               'dash_type': 'square_dot'}})
```

Drop Lines are only available in Line, Area and Stock charts.

10.15 chart.set_high_low_lines()

set_high_low_lines(*options*)

Set properties for the chart high-low lines.

Parameters options (*dict*) – A dictionary of options.

The `set_high_low_lines()` method adds High-Low lines to charts to show the maximum and minimum values of points in a Category:

```
chart.set_high_low_lines()
```

It is possible to format the High-Low Line `line` properties if required. See *Chart Formatting*:

```
chart.set_high_low_lines({
    'line': {
        'color': 'red',
        'dash_type': 'square_dot'
    }
})
```

High-Low Lines are only available in Line and Stock charts.

10.16 chart.show_blanks_as()

show_blanks_as(*option*)

Set the option for displaying blank data in a chart.

Parameters option (*string*) – A string representing the display option.

The `show_blanks_as()` method controls how blank data is displayed in a chart:

```
chart.show_blanks_as('span')
```

The available options are:

```
'gap'   # Blank data is shown as a gap. The default.
'zero'  # Blank data is displayed as zero.
'span'  # Blank data is connected with a line.
```

10.17 chart.show_hidden_data()

show_hidden_data()

Display data on charts from hidden rows or columns.

Display data in hidden rows or columns on the chart:

```
chart.show_hidden_data()
```

10.18 chart.set_rotation()

set_rotation(*rotation*)

Set the Pie/Doughnut chart rotation.

Parameters **rotation** (*int*) – The angle of rotation.

The `set_rotation()` method is used to set the rotation of the first segment of a Pie/Doughnut chart. This has the effect of rotating the entire chart:

```
chart->set_rotation(90)
```

The angle of rotation must be `0 <= rotation <= 360`.

This option is only available for Pie/Doughnut charts.

10.19 chart.set_hole_size()

set_hole_size(*size*)

Set the Doughnut chart hole size.

Parameters **size** (*int*) – The hole size as a percentage.

The `set_hole_size()` method is used to set the hole size of a Doughnut chart:

```
chart->set_hole_size(33)
```

The angle of hole size must be `10 <= size <= 90`.

This option is only available for Doughnut charts.

See also *Working with Charts* and *Chart Examples*.

CHAPTER

ELEVEN

THE CHARTSHEET CLASS

In Excel a chartsheet is a worksheet that only contains a chart.

The **Chartsheet** class has some of the functionality of data *Worksheets* such as tab selection, headers, footers, margins and print properties but its primary purpose is to display a single chart. This makes it different from ordinary data worksheets which can have one or more *embedded* charts.

Like a data worksheet a chartsheet object isn't instantiated directly. Instead a new chartsheet is created by calling the `add_chartsheet()` method from a *Workbook* object:

```
workbook   = xlsxwriter.Workbook('filename.xlsx')
worksheet  = workbook.add_worksheet()  # Required for the chart data.
chartsheet = workbook.add_chartsheet()
#...
workbook.close()
```

A chartsheet object functions as a worksheet and not as a chart. In order to have it display data a *Chart* object must be created and added to the chartsheet:

```
chartsheet = workbook.add_chartsheet()
chart      = workbook.add_chart({'type': 'bar'})

# Configure the chart.

chartsheet.set_chart(chart)
```

The data for the chartsheet chart must be contained on a separate worksheet. That is why it is always created in conjunction with at least one data worksheet, as shown above.

11.1 chartsheet.set_chart()

set_chart(*chart*)

Add a chart to a chartsheet.

Parameters **chart** – A chart object.

The `set_chart()` method is used to insert a chart into a chartsheet. A chart object is created via the Workbook `add_chart()` method where the chart type is specified:

```
chart = workbook.add_chart({type, 'column'})

chartsheet.set_chart(chart)
```

Only one chart can be added to an individual chartsheet.

See *The Chart Class*, *Working with Charts* and *Chart Examples*.

11.2 Worksheet methods

The following *The Worksheet Class* methods are also available through a chartsheet:

- `activate()`
- `select()`
- `hide()`
- `set_first_sheet()`
- `protect()`

- `set_zoom()`
- `set_tab_color()`
- `set_landscape()`
- `set_portrait()`
- `set_paper()`
- `set_margins()`
- `set_header()`
- `set_footer()`
- `get_name()`

For example:

```
chartsheet.set_tab_color('#FF9900')
```

The `set_zoom()` method can be used to modify the displayed size of the chart.

11.3 Chartsheet Example

See *Example: Chartsheet*.

CHAPTER

TWELVE

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.

Note: Ranges in A1 notation must be in uppercase, like in Excel.

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

CHAPTER

THIRTEEN

RELATIVE AND ABSOLUTE CELL REFERENCES

When dealing with Excel cell references it is important to distinguish between relative and absolute cell references in Excel.

Relative cell references change when they are copied while **Absolute** references maintain fixed row and/or column references. In Excel absolute references are prefixed by the dollar symbol as shown below:

```
A1    # Column and row are relative.
$A1   # Column is absolute and row is relative.
A$1   # Column is relative and row is absolute.
$A$1  # Column and row are absolute.
```

See the Microsoft Office documentation for more information on relative and absolute references.

Some functions such as `conditional_format()` require absolute references.

CHAPTER

FOURTEEN

DEFINED NAMES AND NAMED RANGES

It is also possible to define and use "Defined names/Named ranges" in workbooks and worksheets, see `define_name()`:

```
workbook.define_name('Exchange_rate', '=0.96')
worksheet.write('B3', '=B2*Exchange_rate')
```

See also *Example: Defined names/Named ranges*.

CHAPTER

FIFTEEN

CELL UTILITY FUNCTIONS

The `XlsxWriter utility` module contains several helper functions for dealing with A1 notation as shown below. These functions can be imported as follows:

```
from xlsxwriter.utility import xl_rowcol_to_cell

cell = xl_rowcol_to_cell(1, 2)  # C2
```

15.1 xl_rowcol_to_cell()

xl_rowcol_to_cell(*row*, *col*[, *row_abs*, *col_abs*])

Convert a zero indexed row and column cell reference to a A1 style string.

Parameters

- **row** (*int*) – The cell row.
- **col** (*int*) – The cell column.
- **row_abs** (*bool*) – Optional flag to make the row absolute.
- **col_abs** (*bool*) – Optional flag to make the column absolute.

Return type A1 style string.

The `xl_rowcol_to_cell()` function converts a zero indexed row and column cell values to an A1 style string:

```
cell = xl_rowcol_to_cell(0, 0)   # A1
cell = xl_rowcol_to_cell(0, 1)   # B1
cell = xl_rowcol_to_cell(1, 0)   # A2
```

The optional parameters `row_abs` and `col_abs` can be used to indicate that the row or column is absolute:

```
str = xl_rowcol_to_cell(0, 0, col_abs=True)                # $A1
str = xl_rowcol_to_cell(0, 0, row_abs=True)                # A$1
str = xl_rowcol_to_cell(0, 0, row_abs=True, col_abs=True)  # $A$1
```

15.2 xl_cell_to_rowcol()

xl_cell_to_rowcol(*cell_str*)

Convert a cell reference in A1 notation to a zero indexed row and column.

Parameters **cell_str** (*string*) – A1 style string, absolute or relative.

Return type Tuple of ints for (row, col).

The `xl_cell_to_rowcol()` function converts an Excel cell reference in `A1` notation to a zero based row and column. The function will also handle Excel's absolute, `$`, cell notation:

```
(row, col) = xl_cell_to_rowcol('A1')    # (0, 0)
(row, col) = xl_cell_to_rowcol('B1')    # (0, 1)
(row, col) = xl_cell_to_rowcol('C2')    # (1, 2)
(row, col) = xl_cell_to_rowcol('$C2')   # (1, 2)
(row, col) = xl_cell_to_rowcol('C$2')   # (1, 2)
(row, col) = xl_cell_to_rowcol('$C$2')  # (1, 2)
```

15.3 xl_col_to_name()

xl_col_to_name(*col*[, *col_abs*])

Convert a zero indexed column cell reference to a string.

Parameters

- **col** (*int*) – The cell column.
- **col_abs** (*bool*) – Optional flag to make the column absolute.

Return type Column style string.

The `xl_col_to_name()` converts a zero based column reference to a string:

```
column = xl_col_to_name(0)    # A
column = xl_col_to_name(1)    # B
column = xl_col_to_name(702)  # AAA
```

The optional parameter `col_abs` can be used to indicate if the column is absolute:

```
column = xl_col_to_name(0, False)  # A
column = xl_col_to_name(0, True)   # $A
column = xl_col_to_name(1, True)   # $B
```

15.4 xl_range()

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

Converts zero indexed row and column cell references to a A1:B1 range string.

Parameters

- **first_row** (*int*) – The first cell row.
- **first_col** (*int*) – The first cell column.
- **last_row** (*int*) – The last cell row.
- **last_col** (*int*) – The last cell column.

Return type A1:B1 style range string.

The `xl_range()` function converts zero based row and column cell references to an `A1:B1` style range string:

```
cell_range = xl_range(0, 0, 9, 0)  # A1:A10
cell_range = xl_range(1, 2, 8, 2)  # C2:C9
cell_range = xl_range(0, 0, 3, 4)  # A1:E4
```

15.5 xl_range_abs()

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

Converts zero indexed row and column cell references to a $A$1:$B$1 absolute range string.

Parameters

- **first_row** (*int*) – The first cell row.
- **first_col** (*int*) – The first cell column.
- **last_row** (*int*) – The last cell row.
- **last_col** (*int*) – The last cell column.

Return type $A$1:$B$1 style range string.

The `xl_range_abs()` function converts zero based row and column cell references to an absolute `$A$1:$B$1` style range string:

```
cell_range = xl_range_abs(0, 0, 9, 0)  # $A$1:$A$10
cell_range = xl_range_abs(1, 2, 8, 2)  # $C$2:$C$9
cell_range = xl_range_abs(0, 0, 3, 4)  # $A$1:$E$4
```

CHAPTER

SIXTEEN

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.

16.1 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 formatting properties for a row or column:

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

16.2 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 size
	Font color
		`'font_name'`
		`'font_size'`
		`'font_color'`
			`set_font_name()`
			`set_font_size()`
			`set_font_color()`

			Continued on next page

Table 16.1 – continued from previous page

Category	Description	Property	Method Name
	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. The key/value interface has proved to be more flexible in real world programs and is the recommended method for setting format properties.

16.3 Format Colors

Format property colors are specified using a Html sytle #RRGGBB value or a limited number of named colors:

```
cell_format1.set_font_color('#FF0000')
cell_format2.set_font_color('red')
```

See *Working with Colors* for more details.

16.4 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.
```

16.5 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.

CHAPTER

SEVENTEEN

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:

```
import xlsxwriter

workbook = xlsxwriter.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 2013 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`, `datetime.time` and `datetime.timedelta`.

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:

```python
from datetime import datetime
import xlsxwriter

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

# Expand the first columns so that the dates are 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

    workbook.close()
```

17.1 Default Date Formatting

In certain circumstances you may wish to apply a default date format when writing datetime objects, for example, when handling a row of data with `write_row()`.

In these cases it is possible to specify a default date format string using the `Workbook()` constructor `default_date_format` option:

```
workbook = xlsxwriter.Workbook('datetimes.xlsx', {'default_date_format':
                                                  'dd/mm/yy'})
worksheet = workbook.add_worksheet()
date_time = datetime.now()
worksheet.write_datetime(0, 0, date_time)  # Formatted as 'dd/mm/yy'

workbook.close()
```

CHAPTER

EIGHTEEN

WORKING WITH COLORS

Throughout XlsxWriter colors are specified using a Html sytle #RRGGBB value. For example with a *Format* object:

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

For backward compatibility a limited number of color names are supported:

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

The color names and corresponding #RRGGBB value 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

CHAPTER

NINETEEN

WORKING WITH CHARTS

This section explains how to work with some of the options and features of *The Chart Class*.

The majority of the examples in this section are based on a variation of the following program:

```
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_line.xlsx')
worksheet = workbook.add_worksheet()

# Add the worksheet data to be plotted.
data = [10, 40, 50, 20, 10, 50]
worksheet.write_column('A1', data)

# Create a new chart object.
chart = workbook.add_chart({'type': 'line'})

# Add a series to the chart.
chart.add_series({'values': '=Sheet1!$A$1:$A$6'})

# Insert the chart into the worksheet.
worksheet.insert_chart('C1', chart)

workbook.close()
```

19.1 Chart Value and Category Axes

When working with charts it is important to understand how Excel differentiates between a chart axis that is used for series categories and a chart axis that is used for series values.

In the example above the X axis is the **category** axis and each of the values is evenly spaced and sequential. The Y axis is the **value** axis and points are displayed according to their value.

Excel treats these two types of axis differently and exposes different properties for each.

As such some of the `XlsxWriter` axis properties can be set for a value axis, some can be set for a category axis and some properties can be set for both.

For example `reverse` can be set for either category or value axes while the `min` and `max` properties can only be set for value axes (and Date Axes).

Date Category Axes are a special type of category axis that give them some of the properties of Values axes such as `min` and `max` when used with date or time values.

19.2 Chart Series Options

This following sections detail the more complex options of the `add_series()` Chart method:

```
marker
trendline
y_error_bars
x_error_bars
data_labels
points
smooth
```

19.3 Chart series option: Marker

The marker format specifies the properties of the markers used to distinguish series on a chart. In general only Line and Scatter chart types and trendlines use markers.

The following properties can be set for `marker` formats in a chart:

```
type
size
border
fill
```

The `type` property sets the type of marker that is used with a series:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'marker': {'type': 'diamond'},
})
```

The following `type` properties can be set for `marker` formats in a chart. These are shown in the same order as in the Excel format dialog:

```
automatic
none
square
diamond
triangle
x
star
short_dash
long_dash
circle
plus
```

The `automatic` type is a special case which turns on a marker using the default marker style for the particular series number:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'marker': {'type': 'automatic'},
})
```

If `automatic` is on then other marker properties such as size, border or fill cannot be set.

The `size` property sets the size of the marker and is generally used in conjunction with `type`:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'marker': {'type': 'diamond', 'size': 7},
})
```

Nested `border` and `fill` properties can also be set for a marker:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'marker': {
        'type': 'square',
        'size': 8,
        'border': {'color': 'black'},
        'fill':   {'color': 'red'},
    },
})
```

19.4 Chart series option: Trendline

A trendline can be added to a chart series to indicate trends in the data such as a moving average or a polynomial fit.

The following properties can be set for trendlines in a chart series:

```
type
order       (for polynomial trends)
period      (for moving average)
forward     (for all except moving average)
backward    (for all except moving average)
name
line
```

The `type` property sets the type of trendline in the series:

```
chart.add_series({
    'values':    '=Sheet1!$A$1:$A$6',
    'trendline': {'type': 'linear'},
})
```

The available `trendline` types are:

```
exponential
linear
log
moving_average
polynomial
power
```

A `polynomial` trendline can also specify the `order` of the polynomial. The default value is 2:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'trendline': {
        'type': 'polynomial',
        'order': 3,
    },
})
```

A `moving_average` trendline can also specify the `period` of the moving average. The default value is 2:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'trendline': {
        'type': 'moving_average',
        'period': 2,
    },
})
```

The `forward` and `backward` properties set the forecast period of the trendline:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'trendline': {
        'type': 'polynomial',
        'name': 'My trend name',
        'order': 2,
    },
})
```

The `name` property sets an optional name for the trendline that will appear in the chart legend. If it isn't specified the Excel default name will be displayed. This is usually a combination of the trendline type and the series name:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'trendline': {
        'type': 'polynomial',
        'order': 2,
        'forward': 0.5,
        'backward': 0.5,
    },
})
```

Several of these properties can be set in one go:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'trendline': {
        'type': 'polynomial',
        'name': 'My trend name',
        'order': 2,
        'forward': 0.5,
        'backward': 0.5,
        'line': {
            'color': 'red',
            'width': 1,
            'dash_type': 'long_dash',
        },
    },
})
```

Trendlines cannot be added to series in a stacked chart or pie chart, doughnut chart, radar chart or (when implemented) to 3D or surface charts.

19.5 Chart series option: Error Bars

Error bars can be added to a chart series to indicate error bounds in the data. The error bars can be vertical `y_error_bars` (the most common type) or horizontal `x_error_bars` (for Bar and Scatter charts only).

The following properties can be set for error bars in a chart series:

```
type
value        (for all types except standard error and custom)
plus_values  (for custom only)
minus_values (for custom only)
direction
end_style
line
```

The `type` property sets the type of error bars in the series:

```
chart.add_series({
    'values':       '=Sheet1!$A$1:$A$6',
    'y_error_bars': {'type': 'standard_error'},
})
```

The available error bars types are available:

```
fixed
percentage
standard_deviation
standard_error
custom
```

All error bar types, except for `standard_error` and `custom` must also have a value associated with it for the error bounds:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'y_error_bars': {
        'type': 'percentage',
        'value': 5,
    },
})
```

The `custom` error bar type must specify `plus_values` and `minus_values` which should either by a `Sheet1!$A$1:$A$5` type range formula or a list of values:

```
chart.add_series({
    'categories': '=Sheet1!$A$1:$A$5',
    'values':     '=Sheet1!$B$1:$B$5',
    'y_error_bars': {
        'type':         'custom',
        'plus_values':  '=Sheet1!$C$1:$C$5',
        'minus_values': '=Sheet1!$D$1:$D$5',
    },
})

# or

chart.add_series({
    'categories': '=Sheet1!$A$1:$A$5',
```

```
        'values':     '=Sheet1!$B$1:$B$5',
        'y_error_bars': {
            'type':         'custom',
            'plus_values':  [1, 1, 1, 1, 1],
            'minus_values': [2, 2, 2, 2, 2],
        },
    })
```

Note, as in Excel the items in the `minus_values` do not need to be negative.

The `direction` property sets the direction of the error bars. It should be one of the following:

```
plus   # Positive direction only.
minus  # Negative direction only.
both   # Plus and minus directions, The default.
```

The `end_style` property sets the style of the error bar end cap. The options are 1 (the default) or 0 (for no end cap):

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'y_error_bars': {
        'type': 'fixed',
        'value': 2,
        'end_style': 0,
        'direction': 'minus'
    },
})
```

19.6 Chart series option: Data Labels

Data labels can be added to a chart series to indicate the values of the plotted data points.

The following properties can be set for `data_labels` formats in a chart:

```
value
category
series_name
position
leader_lines
percentage
separator
legend_key
num_format
font
```

The `value` property turns on the *Value* data label for a series:

```
chart.add_series({
    'values':      '=Sheet1!$A$1:$A$6',
    'data_labels': {'value': True},
})
```

The `category` property turns on the *Category Name* data label for a series:

```
chart.add_series({
    'values':      '=Sheet1!$A$1:$A$6',
    'data_labels': {'category': True},
})
```

The `series_name` property turns on the *Series Name* data label for a series:

```
chart.add_series({
    'values':      '=Sheet1!$A$1:$A$6',
    'data_labels': {'series_name': True},
})
```

The `position` property is used to position the data label for a series:

```
chart.add_series({
    'values':      '=Sheet1!$A$1:$A$6',
```

```
        'data_labels': {'series_name': True, 'position': 'center'},
    })
```

In Excel the allowable data label positions vary for different chart types. The allowable positions are:

Position	Line, Scatter, Stock	Bar, Column	Pie, Doughnut	Area, Radar
center	Yes	Yes	Yes	Yes*
right	Yes*			
left	Yes			
above	Yes			
below	Yes			
inside_base		Yes		
inside_end		Yes	Yes	
outside_end		Yes*	Yes	
best_fit			Yes*	

Note: The * indicates the default position for each chart type in Excel, if a position isn't specified.

The `percentage` property is used to turn on the display of data labels as a *Percentage* for a series. It is mainly used for pie charts:

```
chart.add_series({
    'values':      '=Sheet1!$A$1:$A$6',
    'data_labels': {'percentage': True},
})
```

The `leader_lines` property is used to turn on *Leader Lines* for the data label of a series. It is mainly used for pie charts:

```
chart.add_series({
    'values':      '=Sheet1!$A$1:$A$6',
    'data_labels': {'value': True, 'leader_lines': True},
})
```

Note: Even when leader lines are turned on they aren't automatically visible in Excel or XlsxWriter. Due to an Excel limitation (or design) leader lines only appear if the data label is moved manually or if the data labels are very close and need to be adjusted automatically.

The `separator` property is used to change the separator between multiple data label items:

```
chart.add_series({
    'values':      '=Sheet1!$A$1:$A$6',
    'data_labels': {'value': True, 'category': True, 'separator': "\n"},
})
```

The separator value must be one of the following strings:

```
', '
';'
'.'
```

```
'\n'
' '
```

The `legend_key` property is used to turn on the *Legend Key* for the data label of a series:

```
chart.add_series({
    'values':      '=Sheet1!$A$1:$A$6',
    'data_labels': {'value': True, 'legend_key': True},
})
```

The `num_format` property is used to set the number format for the data labels of a series:

```
chart.add_series({
    'values':      '=Sheet1!$A$1:$A$5',
    'data_labels': {'value': True, 'num_format': '#,##0.00'},
})
```

The number format is similar to the Worksheet Cell Format `num_format` apart from the fact that a format index cannot be used. An explicit format string must be used as shown above. See `set_num_format()` for more information.

The `font` property is used to set the font of the data labels of a series:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$5',
    'data_labels': {
        'value': True,
        'font': {'name': 'Consolas'}
    },
})
```

See *Chart Fonts*.

19.7 Chart series option: Points

In general formatting is applied to an entire series in a chart. However, it is occasionally required to format individual points in a series. In particular this is required for Pie/Doughnute charts where each segment is represented by a point.

In these cases it is possible to use the `points` property of `add_series()`:

```
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_pie.xlsx')

worksheet = workbook.add_worksheet()
chart = workbook.add_chart({'type': 'pie'})

data = [
    ['Pass', 'Fail'],
    [90, 10],
]
```

```
worksheet.write_column('A1', data[0])
worksheet.write_column('B1', data[1])

chart.add_series({
    'categories': '=Sheet1!$A$1:$A$2',
    'values':     '=Sheet1!$B$1:$B$2',
    'points': [
        {'fill': {'color': 'green'}},
        {'fill': {'color': 'red'}},
    ],
})

worksheet.insert_chart('C3', chart)

workbook.close()
```

The `points` property takes a list of format options (see the "Chart Formatting" section below). To assign default properties to points in a series pass `None` values in the array ref:

```
# Format point 3 of 3 only.
chart.add_series({
    'values': '=Sheet1!A1:A3',
    'points': [
        None,
        None,
        {'fill': {'color': '#990000'}},
    ],
})

# Format point 1 of 3 only.
chart.add_series({
    'values': '=Sheet1!A1:A3',
    'points': [
        {'fill': {'color': '#990000'}},
    ],
```

```
})
```

19.8 Chart series option: Smooth

The `smooth` option is used to set the smooth property of a line series. It is only applicable to the `line` and `scatter` chart types:

```
chart.add_series({
    'categories': '=Sheet1!$A$1:$A$5',
    'values':     '=Sheet1!$B$1:$B$5',
    'smooth':     True,
})
```

19.9 Chart Formatting

The following chart formatting properties can be set for any chart object that they apply to (and that are supported by XlsxWriter) such as chart lines, column fill areas, plot area borders, markers, gridlines and other chart elements:

```
line
border
fill
```

Chart formatting properties are generally set using dicts:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'line':   {'color': 'red'},
})
```

In some cases the format properties can be nested. For example a `marker` may contain `border` and `fill` sub-properties:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'line':   {'color': 'blue'},
    'marker': {'type': 'square',
               'size,': 5,
               'border': {'color': 'red'},
               'fill':   {'color': 'yellow'}
    },
})
```

19.10 Chart formatting: Line

The line format is used to specify properties of line objects that appear in a chart such as a plotted line on a chart or a border.

The following properties can be set for `line` formats in a chart:

```
none
color
width
dash_type
```

The `none` property is uses to turn the `line` off (it is always on by default except in Scatter charts). This is useful if you wish to plot a series with markers but without a line:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'line':   {'none': True},
    'marker': {'type': 'automatic'},
})
```

The `color` property sets the color of the `line`:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'line':   {'color': 'red'},
})
```

The available colours are shown in the main XlsxWriter documentation. It is also possible to set the colour of a line with a Html style #RRGGBB string or a limited number named colours, see *Working with Colors*:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'line':   {'color': '#FF9900'},
})
```

The `width` property sets the width of the `line`. It should be specified in increments of 0.25 of a point as in Excel:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'line':   {'width': 3.25},
})
```

The `dash_type` property sets the dash style of the line:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'line':   {'dash_type': 'dash_dot'},
})
```

The following `dash_type` values are available. They are shown in the order that they appear in the Excel dialog:

```
solid
round_dot
square_dot
dash
dash_dot
long_dash
long_dash_dot
long_dash_dot_dot
```

The default line style is `solid`.

More than one `line` property can be specified at a time:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
     'line': {
         'color': 'red',
         'width': 1.25,
         'dash_type': 'square_dot',
     },
})
```

19.11 Chart formatting: Border

The `border` property is a synonym for `line`.

It can be used as a descriptive substitute for `line` in chart types such as Bar and Column that have a border and fill style rather than a line style. In general chart objects with a `border` property will also have a fill property.

19.12 Chart formatting: Fill

The fill format is used to specify filled areas of chart objects such as the interior of a column or the background of the chart itself.

The following properties can be set for `fill` formats in a chart:

```
none
color
```

The `none` property is used to turn the `fill` property off (it is generally on by default):

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'fill':   {'none': True},
    'border': {'color': 'black'}
})
```

The `color` property sets the colour of the `fill` area:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'fill':   {'color': 'red'}
})
```

The available colours are shown in the main XlsxWriter documentation. It is also possible to set the colour of a fill with a Html style #RRGGBB string or a limited number named colours, see *Working with Colors*:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'fill':   {'color': '#FF9900'}
})
```

The `fill` format is generally used in conjunction with a `border` format which has the same properties as a `line` format:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'fill':   {'color': 'red'},
    'border': {'color': 'black'}
})
```

19.13 Chart Fonts

The following font properties can be set for any chart object that they apply to (and that are supported by XlsxWriter) such as chart titles, axis labels, axis numbering and data labels:

```
name
size
bold
italic
underline
rotation
color
```

These properties correspond to the equivalent Worksheet cell Format object properties. See the *The Format Class* and *Working with Formats* sections for more details about Format properties and how to set them.

The following explains the available font properties:

- `name`: Set the font name:

  ```
  chart.set_x_axis({'num_font':  {'name': 'Arial'}})
  ```

- `size`: Set the font size:

  ```
  chart.set_x_axis({'num_font':  {'name': 'Arial', 'size': 9}})
  ```

- `bold`: Set the font bold property:

  ```
  chart.set_x_axis({'num_font':  {'bold': True}})
  ```

- `italic`: Set the font italic property:

  ```
  chart.set_x_axis({'num_font':  {'italic': True}})
  ```

- `underline`: Set the font underline property:

  ```
  chart.set_x_axis({'num_font':  {'underline': True}})
  ```

- `rotation`: Set the font rotation, angle, property in the range -90 to 90 deg:

  ```
  chart.set_x_axis({'num_font':  {'rotation': 45}})
  ```

  This is useful for displaying axis data such as dates in a more compact format.

- `color`: Set the font color property. Can be a color index, a color name or HTML style RGB colour:

  ```
  chart.set_x_axis({'num_font': {'color': 'red' }})
  chart.set_y_axis({'num_font': {'color': '#92D050'}})
  ```

Here is an example of Font formatting in a Chart program:

```
chart.set_title({
    'name': 'Test Results',
    'name_font': {
        'name': 'Calibri',
        'color': 'blue',
    },
})

chart.set_x_axis({
    'name': 'Month',
    'name_font': {
        'name': 'Courier New',
        'color': '#92D050'
    },
    'num_font': {
        'name': 'Arial',
        'color': '#00B0F0',
    },
```

```
})

chart.set_y_axis({
    'name': 'Units',
    'name_font': {
        'name': 'Century',
        'color': 'red'
    },
    'num_font': {
        'bold': True,
        'italic': True,
        'underline': True,
        'color': '#7030A0',
    },
})

chart.set_legend({'font': {'bold': 1, 'italic': 1}})
```

19.14 Chart Layout

The position of the chart in the worksheet is controlled by the `set_size()` method.

It is also possible to change the layout of the following chart sub-objects:

```
plotarea
legend
title
x_axis caption
y_axis caption
```

Here are some examples:

```
chart.set_plotarea({
    'layout': {
        'x':      0.13,
        'y':      0.26,
        'width':  0.73,
        'height': 0.57,
    }
})

chart.set_legend({
    'layout': {
        'x':      0.80,
        'y':      0.37,
        'width':  0.12,
        'height': 0.25,
    }
})

chart.set_title({
    'name':    'Title',
    'overlay': True,
    'layout': {
        'x': 0.42,
        'y': 0.14,
    }
})

chart.set_x_axis({
    'name': 'X axis',
    'name_layout': {
        'x': 0.34,
        'y': 0.85,
    }
})
```

See `set_plotarea()`, `set_legend()`, `set_title()` and `set_x_axis()`,

Note: It is only possible to change the width and height for the `plotarea` and `legend` objects. For the other text based objects the width and height are changed by the font dimensions.

The layout units must be a float in the range `0 < x <= 1` and are expressed as a percentage of the chart dimensions as shown below:

From this the layout units are calculated as follows:

```
layout:
    x      = a / W
    y      = b / H
    width  = w / W
    height = h / H
```

These units are cumbersome and can vary depending on other elements in the chart such as text lengths. However, these are the units that are required by Excel to allow relative positioning. Some trial and error is generally required.

Note: The `plotarea` origin is the top left corner in the plotarea itself and does not take into account the axes.

19.15 Date Category Axes

Date Category Axes are category axes that display time or date information. In XlsxWriter Date Category Axes are set using the `date_axis` option in `set_x_axis()` or `set_y_axis()`:

```
chart.set_x_axis({'date_axis': True})
```

In general you should also specify a number format for a date axis although Excel will usually default to the same format as the data being plotted:

```
chart.set_x_axis({
    'date_axis':  True,
    'num_format': 'dd/mm/yyyy',
})
```

Excel doesn't normally allow minimum and maximum values to be set for category axes. However, date axes are an exception. The `min` and `max` values should be set as Excel times or dates:

```
chart.set_x_axis({
    'date_axis': True,
    'min': date(2013, 1, 2),
    'max': date(2013, 1, 9),
    'num_format': 'dd/mm/yyyy',
})
```

For date axes it is also possible to set the type of the major and minor units:

```
chart.set_x_axis({
    'date_axis':       True,
    'minor_unit':      4,
    'minor_unit_type': 'months',
    'major_unit':      1,
    'major_unit_type': 'years',
    'num_format':      'dd/mm/yyyy',
})
```

See *Example: Date Axis Chart*.

19.16 Chart Secondary Axes

It is possible to add a secondary axis of the same type to a chart by setting the `y2_axis` or `x2_axis` property of the series:

```
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_secondary_axis.xlsx')
worksheet = workbook.add_worksheet()

data = [
    [2, 3, 4, 5, 6, 7],
    [10, 40, 50, 20, 10, 50],
]

worksheet.write_column('A2', data[0])
worksheet.write_column('B2', data[1])

chart = workbook.add_chart({'type': 'line'})

# Configure a series with a secondary axis.
chart.add_series({
    'values': '=Sheet1!$A$2:$A$7',
    'y2_axis': True,
})

# Configure a primary (default) Axis.
chart.add_series({
    'values': '=Sheet1!$B$2:$B$7',
})
```

```
chart.set_legend({'position': 'none'})

chart.set_y_axis({'name': 'Primary Y axis'})
chart.set_y2_axis({'name': 'Secondary Y axis'})

worksheet.insert_chart('D2', chart)

workbook.close()
```

Note it isn't currently possible to add a secondary axis of a different chart type (for example line and column).

19.17 Chartsheets

The examples shown above and in general the most common type of charts in Excel are embedded charts.

However, it is also possible to create "Chartsheets" which are worksheets that are comprised of a single chart:

See *The Chartsheet Class* for details.

19.18 Chart Limitations

The following chart features aren't supported in XlsxWriter:

- Secondary axes of a different chart type to the main chart type.
- 3D charts and controls.
- Bubble, Surface or other chart types not listed in *The Chart Class*.

19.19 Chart Examples

See *Chart Examples*.

CHAPTER

TWENTY

WORKING WITH AUTOFILTERS

An autofilter in Excel is a way of filtering a 2D range of data based on some simple criteria.

	A	B	C	D
1	Region	Item	Volume	Month
2	East	Apple	9000	July
3	East	Apple	5000	July
4	South	Orange	9000	September
5	North	Apple	2000	November
6	West	Apple	9000	November
7	South	Pear	7000	October
8	North	Pear	9000	August
9	West	Orange	1000	December
10	West	Grape	1000	November
11	South	Pear	10000	April
12	West	Grape	6000	January

20.1 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 also use *Row-Column* notation:

```
worksheet.autofilter(0, 0, 10, 3)  # Same as above.
```

20.2 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 this 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
```

20.3 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
   ==
   !=
   >
   <
   >=
   <=

   and
   or
```

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.

20.4 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)
```

20.5 Example

See *Example: Applying Autofilters* for a full example of all these features.

CHAPTER

TWENTYONE

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.

21.1 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`
`maximum`
`ignore_blank`
`dropdown`
`input_title`
`input_message`
`show_input`
`error_title`
`error_message`
`error_type`
`show_error`
	`minimum`	`source`

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})
```

21.1.1 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.
- **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 required very often.

21.1.2 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	!=
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)'})
```

21.1.3 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'})
```

21.1.4 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})
```

21.1.5 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': False,
                                 })
```

21.1.6 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.

21.1.7 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.

21.1.8 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.

21.1.9 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.

21.1.10 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.

21.1.11 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.

21.1.12 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'`.

21.1.13 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.

21.2 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*.

CHAPTER

TWENTYTWO

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:

22.1 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`

22.2 Conditional Format Options

The conditional format options that can be used with `conditional_format()` are explained in the following sections.

22.2.1 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 22.1 – 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.

22.2.2 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.

22.2.3 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.

Additional criteria which are specific to other conditional format types are shown in the relevant sections below.

22.2.4 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})
```

Note: In general any `value` property that refers to a cell reference should use an *absolute reference*, especially if the conditional formatting is applied to a range of values. Without an absolute cell reference the conditional format will not be applied correctly by Excel from the first cell in the formatted range.

22.2.5 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 in a conditional format are font name, font size, superscript and subscript, diagonal borders, all alignment properties and all protection properties.

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*.

22.2.6 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,
                                    })
```

22.2.7 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.

22.2.8 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})
```

22.2.9 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'
```

22.2.10 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.

22.2.11 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',
```

22.2.12 type: duplicate

The `duplicate` type is used to highlight duplicate cells in a range:

```
worksheet.conditional_format('A1:A4', {'type':   'duplicate',
                                        'format': format1})
```

22.2.13 type: unique

The `unique` type is used to highlight unique cells in a range:

```
worksheet.conditional_format('A1:A4', {'type':   'unique',
                                        'format': format1})
```

22.2.14 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})
```

22.2.15 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.

22.2.16 type: blanks

The `blanks` type is used to highlight blank cells in a range:

```
worksheet.conditional_format('A1:A4', {'type':   'blanks',
                                        'format': format1})
```

22.2.17 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})
```

22.2.18 type: errors

The `errors` type is used to highlight error cells in a range:

```
worksheet.conditional_format('A1:A4', {'type':   'errors',
                                        'format': format1})
```

22.2.19 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})
```

22.2.20 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`, `max_value`, `min_color` and `max_color`, see below.

22.2.21 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`, `max_value`, `min_color`, `mid_color` and `max_color`, see below.

22.2.22 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`, `max_value` and `bar_color`, see below.

22.2.23 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': '=$A$1>5',
                                       'format':   format1})
```

The formula is specified in the `criteria`.

Formulas must be written with the US style separator/range operator which is a comma (not semi-colon) and should follow the same rules as `write_formula()`. Also any cell or range references in the formula should be *absolute references* if they are applied to the full range of the conditional format. See the note in the `value` section above.

22.2.24 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
```

22.2.25 mid_type:

Used for `3_color_scale`. Same as `min_type`, see above.

22.2.26 max_type:

Same as `min_type`, see above.

22.2.27 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})
```

22.2.28 mid_value:

Used for `3_color_scale`. Same as `min_value`, see above.

22.2.29 max_value:

Same as `min_value`, see above.

22.2.30 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 *Working with Colors*.

22.2.31 mid_color:

Used for `3_color_scale`. Same as `min_color`, see above.

22.2.32 max_color:

Same as `min_color`, see above.

22.2.33 bar_color:

Used for `data_bar`. Same as `min_color`, see above.

22.2.34 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'})
```

22.3 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*.

CHAPTER

TWENTYTHREE

WORKING WITH WORKSHEET TABLES

Tables in Excel are a way of grouping a range of cells into a single entity that has common formatting or that can be referenced from formulas. Tables can have column headers, autofilters, total rows, column formulas and default formatting.

For more information see An Overview of Excel Tables in the Microsoft Office documentation.

Note: Tables aren't available in XlsxWriter when `Workbook()` `'constant_memory'` mode is enabled.

23.1 add_table()

Tables are added to a worksheet using the `add_table()` method:

```
worksheet.add_table('B3:F7', {options})
```

The data range can be specified in 'A1' or 'Row/Column' notation (see *Working with Cell Notation*):

```
worksheet.add_table('B3:F7')
# Same as:
worksheet.add_table(2, 1, 6, 5)
```

The options parameter should be a dict containing the parameters that describe the table options and data. The available options are:

data
autofilter
header_row
banded_columns
banded_rows
first_column
last_column
style
total_row
columns
name

These options are explained below. There are no required parameters and the options parameter is itself optional if no options are specified (as shown above).

23.2 data

The `data` parameter can be used to specify the data in the cells of the table:

```
data = [
    ['Apples', 10000, 5000, 8000, 6000],
    ['Pears',   2000, 3000, 4000, 5000],
    ['Bananas', 6000, 6000, 6500, 6000],
    ['Oranges',  500,  300,  200,  700],

]

worksheet.add_table('B3:F7', {'data' data})
```

Table data can also be written separately, as an array or individual cells:

```
# These statements are the same as the single statement above.
worksheet.add_table('B3:F7')
worksheet.write_row('B4', data[0])
worksheet.write_row('B5', data[1])
worksheet.write_row('B6', data[2])
worksheet.write_row('B7', data[3])
```

Writing the cell data separately is occasionally required when you need to control the `write_()` methods used to populate the cells or if you wish to modify individual cell formatting.

The `data` structure should be an list of lists holding row data as shown above.

23.3 header_row

The `header_row` parameter can be used to turn on or off the header row in the table. It is on by default:

```
# Turn off the header row.
worksheet.add_table('B4:F7', {'header_row': False})
```

The header row will contain default captions such as `Column 1`, `Column 2`, etc. These captions can be overridden using the `columns` parameter below.

23.4 autofilter

The `autofilter` parameter can be used to turn on or off the autofilter in the header row. It is on by default:

```
# Turn off the default autofilter.
worksheet.add_table('B3:F7', {'autofilter': False})
```

The `autofilter` is only shown if the `header_row` is on. Filter conditions within the table are not supported.

23.5 banded_rows

The `banded_rows` parameter can be used to create rows of alternating colour in the table. It is on by default:

```
# Turn off banded rows.
worksheet.add_table('B3:F7', {'banded_rows': False})
```

23.6 banded_columns

The `banded_columns` parameter can be used to used to create columns of alternating colour in the table. It is off by default:

```
# Turn on banded columns.
worksheet.add_table('B3:F7', {'banded_columns': True})
```

See the above image.

23.7 first_column

The `first_column` parameter can be used to highlight the first column of the table. The type of highlighting will depend on the `style` of the table. It may be bold text or a different colour. It is off by default:

```
# Turn on highlighting for the first column in the table.
worksheet.add_table('B3:F7', {'first_column': True})
```

23.8 last_column

The `last_column` parameter can be used to highlight the last column of the table. The type of highlighting will depend on the `style` of the table. It may be bold text or a different colour. It is off by default:

```
# Turn on highlighting for the last column in the table.
worksheet.add_table('B3:F7', {'last_column': True})
```

See the above image.

23.9 style

The `style` parameter can be used to set the style of the table. Standard Excel table format names should be used (with matching capitalisation):

```
worksheet.add_table('B3:F7', {'data': data,
                              'style': 'Table Style Light 11'})
```

The default table style is 'Table Style Medium 9'.

23.10 name

The `name` parameter can be used to set the name of the table.

By default tables are named `Table1`, `Table2`, etc. If you override the table name you must ensure that it doesn't clash with an existing table name and that it follows Excel's requirements for table names:

```
worksheet.add_table('B3:F7', {'name': 'SalesData'})
```

If you need to know the name of the table, for example to use it in a formula, you can get it as follows:

```
table = worksheet.add_table('B3:F7')
table_name = table.name
```

23.11 total_row

The `total_row` parameter can be used to turn on the total row in the last row of a table. It is distinguished from the other rows by a different formatting and also with dropdown `SUBTOTAL` functions:

```
worksheet.add_table('B3:F7', {'total_row': True})
```

The default total row doesn't have any captions or functions. These must by specified via the `columns` parameter below.

23.12 columns

The `columns` parameter can be used to set properties for columns within the table.

The sub-properties that can be set are:

header
formula
total_string
total_function
total_value
format

The column data must be specified as a list of dicts. For example to override the default 'Column n' style table headers:

```
worksheet.add_table('B3:F7', {'data': data,
                              'columns': [{'header': 'Product'},
                                          {'header': 'Quarter 1'},
                                          {'header': 'Quarter 2'},
                                          {'header': 'Quarter 3'},
                                          {'header': 'Quarter 4'},
                                          ]})
```

See the resulting image above.

If you don't wish to specify properties for a specific column you pass an empty hash ref and the defaults will be applied:

```
...
columns, [
    {header, 'Product'},
    {header, 'Quarter 1'},
    {},                         # Defaults to 'Column 3'.
    {header, 'Quarter 3'},
    {header, 'Quarter 4'},
]
...
```

Column formulas can by applied using the column `formula` property:

```
formula = '=SUM(Table8[@[Quarter 1]:[Quarter 4]])'

worksheet.add_table('B3:G7', {'data': data,
                              'columns': [{'header': 'Product'},
                                          {'header': 'Quarter 1'},
                                          {'header': 'Quarter 2'},
                                          {'header': 'Quarter 3'},
                                          {'header': 'Quarter 4'},
                                          {'header': 'Year',
                                           'formula': formula},
                                          ]})
```

The Excel 2007 style `[#This Row]` and Excel 2010 style @ structural references are supported within the formula. However, other Excel 2010 additions to structural references aren't supported and formulas should conform to Excel 2007 style formulas. See the Microsoft documentation on Using structured references with Excel tables for details.

As stated above the `total_row` table parameter turns on the "Total" row in the table but it doesn't populate it with any defaults. Total captions and functions must be specified via the `columns` property and the `total_string` and `total_function` sub properties:

```
options = {'data': data,
           'total_row': 1,
           'columns': [{'header': 'Product', 'total_string': 'Totals'},
                       {'header': 'Quarter 1', 'total_function': 'sum'},
                       {'header': 'Quarter 2', 'total_function': 'sum'},
                       {'header': 'Quarter 3', 'total_function': 'sum'},
                       {'header': 'Quarter 4', 'total_function': 'sum'},
                       {'header': 'Year',
                        'formula': '=SUM(Table10[@[Quarter 1]:[Quarter 4]])',
                        'total_function': 'sum'
                        },
                       ]}

# Add a table to the worksheet.
worksheet.add_table('B3:G8', options)
```

The supported totals row SUBTOTAL functions are:

average
count_nums
count
max
min
std_dev
sum
var

User defined functions or formulas aren't supported.

It is also possible to set a calculated value for the `total_function` using the `total_value` sub property. This is only necessary when creating workbooks for applications that cannot calculate the value of formulas automatically. This is similar to setting the `value` optional property in `write_formula()`:

```
options = {'data': data,
           'total_row': 1,
           'columns': [{'total_string': 'Totals'},
                       {'total_function': 'sum', 'total_value': 150},
                       {'total_function': 'sum', 'total_value': 200},
                       {'total_function': 'sum', 'total_value': 333},
                       {'total_function': 'sum', 'total_value': 124},
                       {'formula': '=SUM(Table10[@[Quarter 1]:[Quarter 4]])',
                        'total_function': 'sum',
                        'total_value': 807}]}
```

Formats can also be applied to columns:

```
currency_format = workbook.add_format(num_format, '$#,##0')

worksheet.add_table('B3:D8', {'data': data,
                              'total_row': 1,
                              'columns': [{'header': 'Product'},
                                          {'header': 'Quarter 1',
                                           'total_function': 'sum',
                                           'format': currency_format,
                                           },
                                          {'header': 'Quarter 2',
                                           'total_function': 'sum',
                                           'format': currency_format,
                                           },
                                          ]})
```

Standard XlsxWriter *Format object* objects are used for this formatting. However, they should be limited to numerical formats. Overriding other table formatting may produce inconsistent results.

23.13 Example

All of the images shown above are taken from *Example: Worksheet Tables*.

CHAPTER

TWENTYFOUR

WORKING WITH SPARKLINES

Sparklines are a feature of Excel 2010+ which allows you to add small charts to worksheet cells. These are useful for showing visual trends in data in a compact format.

Sparklines were invented by Edward Tufte: http://en.wikipedia.org/wiki/Sparklines

24.1 The add_sparkline() method

The `add_sparkline()` worksheet method is used to add sparklines to a cell or a range of cells:

```
worksheet.add_sparkline(0, 5, {'range': 'Sheet1!A1:E1'})
```

Both row-column and A1 style notation are supported. See *Working with Cell Notation* for more details.

The parameters to `add_sparkline()` must be passed in a dictionary. The main sparkline parameters are:

range (required)
type
style
markers
negative_points
axis
reverse

Other, less commonly used parameters are:

location
high_point
low_point
first_point
last_point
max
min
empty_cells
show_hidden
date_axis
weight
series_color
negative_color
markers_color
first_color
last_color
high_color
low_color

sparklines2.xlsx
Home
Layout
Tables
Charts
SmartArt
A48
A
B
Sparkline
Description
A default "line" sparkline.
A default "column" sparkline.
A default "win/loss" sparkline.
Line with markers.
Line with high and low points.
Line with first and last point markers.
Line with negative point markers.
Line with axis.
Column with default style (1).
Column with style 2.
Column with style 3.
Column with style 4.
Column with style 5.
Column with style 6.
Column with a user defined colour.
A win/loss sparkline.
A win/loss sparkline with negative point
A left to right column (the default).
A right to left column.
Growth
Sparkline and text in one cell.
A grouped sparkline. Changes are applie
Sheet1
Sheet2
Normal View
Ready

These parameters are explained in the sections below.

Note: Sparklines are a feature of Excel 2010+ only. You can write them to an XLSX file that can be read by Excel 2007 but they won't be displayed.

24.2 range

The `range` specifier is the only non-optional parameter.

It specifies the cell data range that the sparkline will plot:

```
worksheet.add_sparkline('F1', {'range': 'A1:E1'})
```

The `range` should be a 2D array. (For 3D arrays of cells see "Grouped Sparklines" below).

If `range` is not on the same worksheet you can specify its location using the usual Excel notation:

```
worksheet.add_sparkline('F1', {'range': 'Sheet2!A1:E1'})
```

If the worksheet contains spaces or special characters you should quote the worksheet name in the same way that Excel does:

```
worksheet.add_sparkline('F1', {'range': "'Monthly Data'!A1:E1"})
```

24.3 type

Specifies the type of sparkline. There are 3 available sparkline types:

```
line (default)
column
win_loss
```

For example:

```
worksheet.add_sparkline('F2', {'range': 'A2:E2',
                               'type': 'column'})
```

24.4 style

Excel provides 36 built-in Sparkline styles in 6 groups of 6. The `style` parameter can be used to replicate these and should be a corresponding number from 1 .. 36:

```
worksheet.add_sparkline('F2', {'range': 'A2:E2',
                               'type': 'column',
                               'style': 12})
```

The style number starts in the top left of the style grid and runs left to right. The default style is 1. It is possible to override colour elements of the sparklines using the `_color` parameters below.

24.5 markers

Turn on the markers for `line` style sparklines:

```
worksheet.add_sparkline('A6', {'range': 'Sheet2!A1:J1',
                               'markers': True})
```

Markers aren't shown in Excel for `column` and `win_loss` sparklines.

24.6 negative_points

Highlight negative values in a sparkline range. This is usually required with `win_loss` sparklines:

```
worksheet.add_sparkline('A9', {'range': 'Sheet2!A1:J1',
                               'negative_points': True})
```

24.7 axis

Display a horizontal axis in the sparkline:

```
worksheet.add_sparkline('A10', {'range': 'Sheet2!A1:J1',
                                'axis': True})
```

24.8 reverse

Plot the data from right-to-left instead of the default left-to-right:

```
worksheet.add_sparkline('A24', {'range': 'Sheet2!A4:J4',
                                'type': 'column',
                                'style': 20,
                                'reverse': True})
```

24.9 weight

Adjust the default line weight (thickness) for `line` style sparklines:

```
worksheet.add_sparkline('F2', {'range': 'A2:E2',
                               'weight': 0.25})
```

The weight value should be one of the following values allowed by Excel:

```
0.25, 0.5, 0.75, 1, 1.25, 2.25, 3, 4.25, 6
```

24.10 high_point, low_point, first_point, last_point

Highlight points in a sparkline range:

```
worksheet.add_sparkline('A7', {'range': 'Sheet2!A1:J1',
                               'high_point': True,
                               'low_point': True,
                               'first_point': True})
```

24.11 max, min

Specify the maximum and minimum vertical axis values:

```
worksheet.add_sparkline('F1', {'range': 'A1:E1',
                               'max': 0.5,
                               'min': -0.5})
```

As a special case you can set the maximum and minimum to be for a group of sparklines rather than one:

```
'max': 'group'
```

See "Grouped Sparklines" below.

24.12 empty_cells

Define how empty cells are handled in a sparkline:

```
worksheet.add_sparkline('F1', {'range': 'A1:E1',
                               'empty_cells': 'zero'})
```

The available options are:

- `gaps`: show empty cells as gaps (the default).
- `zero`: plot empty cells as 0.
- `connect`: Connect points with a line ("line" type sparklines only).

24.13 show_hidden

Plot data in hidden rows and columns:

```
worksheet.add_sparkline('F3', {'range': 'A3:E3',
                               'show_hidden': True})
```

Note, this option is off by default.

24.14 date_axis

Specify an alternative date axis for the sparkline. This is useful if the data being plotted isn't at fixed width intervals:

```
worksheet.add_sparkline('F3', {'range': 'A3:E3',
                               'date_axis': 'A4:E4'})
```

The number of cells in the date range should correspond to the number of cells in the data range.

24.15 series_color

It is possible to override the colour of a sparkline style using the following parameters:

```
series_color
negative_color
markers_color
first_color
last_color
high_color
low_color
```

The color should be specified as a HTML style `#rrggbb` hex value:

```
worksheet.add_sparkline('A18', {'range': 'Sheet2!A2:J2',
                                'type': 'column',
                                'series_color': '#E965E0'})
```

24.16 location

By default the sparkline location is specified by `row` and `col` in `add_sparkline()`. However, for grouped sparklines it is necessary to specify more than one cell location. The `location` parameter is used to specify a list of cells. See "Grouped Sparklines" below.

24.17 Grouped Sparklines

The `add_sparkline()` worksheet method can be used multiple times to write as many sparklines as are required in a worksheet.

However, it is sometimes necessary to group contiguous sparklines so that changes that are applied to one are applied to all. In Excel this is achieved by selecting a 3D range of cells for the data `range` and a 2D range of cells for the `location`.

In XlsxWriter, you can simulate this by passing an array refs of values to `location` and `range`:

```
worksheet.add_sparkline('A27', {'location': ['A27',   'A28',   'A29'],
                                 'range':    ['A5:J5', 'A6:J6', 'A7:J7']})
```

24.18 Sparkline examples

See *Example: Sparklines (Simple)* and *Example: Sparklines (Advanced)*.

CHAPTER

TWENTYFIVE

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')
```

25.1 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 *Working with 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)*.

CHAPTER

TWENTYSIX

WORKING WITH OUTLINES AND GROUPING

Excel allows you to group rows or columns so that they can be hidden or displayed with a single mouse click. This feature is referred to as outlines and grouping.

Outlines can reduce complex data down to a few salient sub-totals or summaries.

For example the following is a worksheet with three outlines. Rows 2-11 are grouped at level 1 and rows 2-5 and 7-10 are grouped at level 2. The lines at the left hand side are called outline level bars and the level is shown by the small numeral above the outline.

Clicking the minus sign on each of the level 2 outlines will collapse and hide the data as shown below. The minus sign changes to a plus sign to indicate that the data in the outline is hidden.

This shows the usefulness of outlines: with 2 mouse clicks we have reduced the amount of visual data down to 2 sub-totals and a master total.

Clicking on the minus sign on the level 1 outline will collapse the remaining rows as follows:

26.1 Outlines and Grouping in XlsxWriter

Grouping in `XlsxWriter` is achieved by setting the outline level via the `set_row()` and `set_column()` worksheet methods:

```
worksheet.set_row(row, height, format, options)
worksheet.set_column(first_col, last_col, width, format, options)
```

Adjacent row or columns with the same outline level are grouped together into a single outline.

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': True})

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

The following example sets an outline level of 1 for rows 1 and 2 (zero-indexed) and columns B to G. The parameters `height` and `cell_format` are assigned default values:

```
worksheet.set_row(1, None, None, {'level': 1})
worksheet.set_row(2, None, None, {'level': 1})
worksheet.set_column('B:G', None, None, {'level': 1})
```

Excel allows up to 7 outline levels. Therefore the `level` parameter should be in the range `0 <= level <= 7`.

Rows and columns can be collapsed by setting the `hidden` flag for the hidden rows/columns and setting the `collapsed` flag for the row/column that has the collapsed `'+'` symbol:

```
worksheet.set_row(1, None, None, {'level': 1, 'hidden': True})
worksheet.set_row(2, None, None, {'level': 1, 'hidden': True})
worksheet.set_row(3, None, None, {'collapsed': True})

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

Note: Setting the `collapsed` flag is particularly important for compatibility with non-Excel spreadsheets.

For a more complete examples see *Example: Outline and Grouping* and *Example: Collapsed Outline and Grouping*.

Some additional outline properties can be set via the `outline_settings()` worksheet method.

CHAPTER

TWENTYSEVEN

WORKING WITH MEMORY AND PERFORMANCE

By default XlsxWriter holds all cell data in memory. This is to allow future features when formatting is applied separately from the data.

The effect of this is that XlsxWriter can consume a lot of memory and it is possible to run out of memory when creating large files.

Fortunately, this memory usage can be reduced almost completely by using the `Workbook()` `'constant_memory'` property:

```
workbook = xlsxwriter.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 the only such feature is *Worksheet Tables*.

For larger files `'constant_memory'` mode also gives an increase in execution speed, see below.

27.1 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 of less linearly with the number of rows:

Rows	Columns	Time (s)	Memory (bytes)
200	50	0.43	2346728
400	50	0.84	4670904
800	50	1.68	8325928
1600	50	3.39	17855192
3200	50	6.82	32279672
6400	50	13.66	64862232
12800	50	27.60	128851880

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	0.37	62208
400	50	0.74	62208
800	50	1.46	62208
1600	50	2.93	62208
3200	50	5.90	62208
6400	50	11.84	62208
12800	50	23.63	62208

In the `constant_memory` mode the performance is also increased slightly.

These figures were generated using programs in the `dev/performance` directory of the XlsxWriter repo.

27.2 Benchmark of Python Excel Writers

If you wish to compare the performance of different Python Excel writing modules there is a progam called bench_excel_writers.py in the `dev/performance` directory of the XlsxWriter repo.

And here is the output for 10,000 rows x 50 columns using the latest version of the modules at the time of writing:

```
Versions:
    python      : 2.7.4
    openpyxl    : 2.1.0
    pyexcelerate: 0.6.3
    xlsxwriter  : 0.5.7
    xlwt        : 0.7.5
```

```
Dimensions:
    Rows = 10000
    Cols = 50

Times:
    pyexcelerate          :  10.11
    xlwt                  :  15.67
    xlsxwriter (optimised):  19.70
    xlsxwriter            :  23.50
    openpyxl   (optimised):  95.82
    openpyxl              :  95.90
```

As with any benchmark the results will depend on Python/module versions, CPU, RAM and Disk I/O and on the benchmark itself. So make sure to verify these results for your own setup.

CHAPTER

TWENTYEIGHT

WORKING WITH VBA MACROS

This section explains how to add a VBA file containing functions or macros to an XlsxWriter file.

Note: This feature should be considered as experimental.

28.1 The Excel XLSM file format

An Excel `xlsm` file is exactly the same as a `xlsx` file except that is contains an additional `vbaProject.bin` file which contains functions and/or macros. Excel uses a different extension to differ-

entiate between the two file formats since files containing macros are usually subject to additional security checks.

28.2 How VBA macros are included in XlsxWriter

The `vbaProject.bin` file is a binary OLE COM container. This was the format used in older `xls` versions of Excel prior to Excel 2007. Unlike all of the other components of an xlsx/xlsm file the data isn't stored in XML format. Instead the functions and macros as stored as pre-parsed binary format. As such it wouldn't be feasible to define macros and create a `vbaProject.bin` file from scratch (at least not in the remaining lifespan and interest levels of the author).

Instead a workaround is used to extract `vbaProject.bin` files from existing xlsm files and then add these to XlsxWriter files.

28.3 The vba_extract utility

The `vba_extract` utility is used to extract the `vbaProject.bin` binary from an Excel 2007+ xlsm file. The utility is included in the XlsxWriter examples directory and is also installed as a standalone executable file:

```
$ vba_extract.py macro_file.xlsm
Extracted: vbaProject.bin
```

28.4 Adding the VBA macros to a XlsxWriter file

Once the `vbaProject.bin` file has been extracted it can be added to the XlsxWriter workbook using the `add_vba_project()` method:

```
workbook.add_vba_project('./vbaProject.bin')
```

If the VBA file contains functions you can then refer to them in calculations using `write_formula()`:

```
worksheet.write_formula('A1', '=MyMortgageCalc(200000, 25)')
```

Excel files that contain functions and macros should use an `xlsm` extension or else Excel will complain and possibly not open the file:

```
workbook = xlsxwriter.Workbook('macros.xlsm')
```

It is also possible to assign a macro to a button that is inserted into a worksheet using the `insert_button()` method:

```
import xlsxwriter

# Note the file extension should be .xlsm.
```

```
workbook = xlsxwriter.Workbook('macros.xlsm')
worksheet = workbook.add_worksheet()

worksheet.set_column('A:A', 30)

# Add the VBA project binary.
workbook.add_vba_project('./vbaProject.bin')

# Show text for the end user.
worksheet.write('A3', 'Press the button to say hello.')

# Add a button tied to a macro in the VBA project.
worksheet.insert_button('B3', {'macro':   'say_hello',
                               'caption': 'Press Me',
                               'width':   80,
                               'height':  30})

workbook.close()
```

It may be necessary to specify a more explicit macro name prefixed by the workbook VBA name as follows:

```
worksheet.insert_button('B3', {'macro': 'ThisWorkbook.say_hello'})
```

See *Example: Adding a VBA macro to a Workbook* from the examples directory for a working example.

Note: Button is the only VBA Control supported by Xlsxwriter. Due to the large effort in implementation (1+ man months) it is unlikely that any other form elements will be added in the future.

28.5 Setting the VBA codenames

VBA macros generally refer to workbook and worksheet objects. If the VBA codenames aren't specified then XlsxWriter will use the Excel defaults of `ThisWorkbook` and `Sheet1`, `Sheet2` etc.

If the macro uses other codenames you can set them using the workbook and worksheet `set_vba_name()` methods as follows:

```
workbook.set_vba_name('MyWorkbook')
worksheet.set_vba_name('MySheet')
```

You can find the names that are used in the VBA editor or by unzipping the `xlsm` file and grepping the files. The following shows how to do that using libxml's xmllint to format the XML for clarity:

```
$ unzip myfile.xlsm -d myfile
$ xmllint --format find myfile -name "*.xml" | xargs | grep "Pr.*codeName"

  <workbookPr codeName="MyWorkbook" defaultThemeVersion="124226"/>
  <sheetPr codeName="MySheet"/>
```

Note: This step is particulary important for macros created with non-English versions of Excel.

28.6 What to do if it doesn't work

As stated at the start of this section this feature is experimental. The Xlsxwriter test suite contains several tests and there is a working example as shown above. However, there is no guarantee that it will work in all cases. Some effort may be required and some knowledge of VBA will certainly help. If things don't work out here are some things to try:

1. Start with a simple macro file, ensure that it works and then add complexity.
2. Try to extract the macros from an Excel 2007 file. The method should work with macros from later versions (it was also tested with Excel 2010 macros). However there may be features in the macro files of more recent version of Excel that aren't backward compatible.
3. Check the code names that macros use to refer to the workbook and worksheets (see the previous section above). In general VBA uses a code name of `ThisWorkbook` to refer to the current workbook and the sheet name (such as `Sheet1`) to refer to the worksheets. These are the defaults used by XlsxWriter. If the macro uses other names then you can specify these using the workbook and worksheet `set_vba_name()` methods:

   ```
   workbook.set_vba_name('MyWorkbook')
   worksheet.set_vba_name('MySheet')
   ```

CHAPTER

TWENTYNINE

EXAMPLES

The following are some of the examples included in the examples directory of the XlsxWriter distribution.

29.1 Example: Hello World

The simplest possible spreadsheet. This is a good place to start to see if the XlsxWriter module is installed correctly.

```
##############################################################################
#
# A hello world spreadsheet using the XlsxWriter Python module.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('hello_world.xlsx')
worksheet = workbook.add_worksheet()

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

workbook.close()
```

29.2 Example: Simple Feature Demonstration

This program is an example of writing some of the features of the XlsxWriter module.

```
##############################################################################
#
# A simple example of some of the features of the XlsxWriter Python module.
```

```
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter


# Create an new Excel file and add a worksheet.
workbook = xlsxwriter.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': True})

# 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)

# Insert an image.
worksheet.insert_image('B5', 'logo.png')

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*.

29.3 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 simple program to write some dates and times to an Excel file
# using the XlsxWriter Python module.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
from datetime import datetime
import xlsxwriter

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

# Expand the first columns so that the dates are 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 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

workbook.close()
```

29.4 Example: Adding hyperlinks

This program is an example of writing hyperlinks to a worksheet. See the `write_url()` method for more details.

```
#######################################################################
#
# Example of how to use the XlsxWriter module to write hyperlinks
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

# Create a new workbook and add a worksheet
workbook = xlsxwriter.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({
    'font_color': 'blue',
    'underline':  1
})

# Add a sample alternative link format.
red_format = workbook.add_format({
    'font_color': 'red',
```

```
    'bold':       1,
    'underline':  1,
    'font_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_url('A1', 'http://www.python.org/')  # Implicit format.
worksheet.write_url('A3', 'http://www.python.org/', url_format, string)
worksheet.write_url('A5', 'http://www.python.org/', url_format, string, tip)
worksheet.write_url('A7', 'http://www.python.org/', red_format)
worksheet.write_url('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()
```

29.5 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.

```
#######################################################################
#
# Example of how to use Python and the XlsxWriter module to write
# simple array formulas.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

# Create a new workbook and add a worksheet
workbook = xlsxwriter.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_formula('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()
```

29.6 Example: Applying Autofilters

This program is an example of using autofilters in a worksheet. See *Working with Autofilters* for more details.

```
#######################################################################
#
# 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 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.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()

    # Convert the number data from the text file.
    for i, item in enumerate(row_data):
        try:
            row_data[i] = float(item)
        except ValueError:
            pass

    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()
```

29.7 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.

```
#######################################################################
#
# 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 2013-2015, John McNamara, jmcnamara@cpan.org
#
from datetime import date, time
import xlsxwriter

workbook = xlsxwriter.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()
```

29.8 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.

```
##############################################################################
#
# 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 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.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()
```

29.9 Example: Defined names/Named ranges

Example of how to create defined names (named ranges) with XlsxWriter.

Defined names are used to define descriptive names to represent a value, a single cell or a range of cells in a workbook or worksheet. See `define_name()`.

```
    ##############################################################################
    #
    # Example of how to create defined names with the XlsxWriter Python module.
    #
    # This method is used to define a user friendly name to represent a value,
    # a single cell or a range of cells in a workbook.
    #
    # Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
    #
```

```
import xlsxwriter


workbook = xlsxwriter.Workbook('defined_name.xlsx')
worksheet1 = workbook.add_worksheet()
worksheet2 = workbook.add_worksheet()

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

# Define a local/worksheet name. Over-rides the "Sales" name above.
workbook.define_name('Sheet2!Sales', '=Sheet2!$G$1:$G$10')

# Write some text in the file and one of the defined names in a formula.
for worksheet in workbook.worksheets():
    worksheet.set_column('A:A', 45)
    worksheet.write('A1', 'This worksheet contains some defined names.')
    worksheet.write('A2', 'See Formulas -> Name Manager above.')
    worksheet.write('A3', 'Example formula in cell B3 ->')

    worksheet.write('B3', '=Exchange_rate')

workbook.close()
```

29.10 Example: Merging Cells

This program is an example of merging cells in a worksheet. See the `merge_range()` method for more details.

```
##############################################################################
#
# A simple example of merging cells with the XlsxWriter Python module.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter


# Create an new Excel file and add a worksheet.
workbook = xlsxwriter.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()
```

29.11 Example: Writing "Rich" strings with multiple formats

This program is an example of writing rich stings with multiple format to a cell in a worksheet. See the `write_rich_string()` method for more details.

```
    ##############################################################
    #
```

```
# An example of using Python and XlsxWriter to write some "rich strings",
# i.e., strings with multiple formats.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.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()
```

29.12 Example: Merging Cells with a Rich String

This program is an example of merging cells that contain a rich string.

Using the standard XlsxWriter API we can only write simple types to merged ranges so we first write a blank string to the merged range. We then overwrite the first merged cell with a rich string.

Note that we must also pass the cell format used in the merged cells format at the end

See the `merge_range()` and `write_rich_string()` methods for more details.

```
##############################################################################
#
# An  example of merging cells which contain a rich string using the
# XlsxWriter Python module.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter


# Create an new Excel file and add a worksheet.
workbook = xlsxwriter.Workbook('merge_rich_string.xlsx')
worksheet = workbook.add_worksheet()

# Set up some formats to use.
red = workbook.add_format({'color': 'red'})
blue = workbook.add_format({'color': 'blue'})
cell_format = workbook.add_format({'align': 'center',
```

```
                                        'valign': 'vcenter',
                                        'border': 1})

    # We can only write simple types to merged ranges so we write a blank string.
    worksheet.merge_range('B2:E5', "", cell_format)

    # We then overwrite the first merged cell with a rich string. Note that we
    # must also pass the cell format used in the merged cells format at the end.
    worksheet.write_rich_string('B2',
                                'This is ',
                                red, 'red',
                                ' and this is ',
                                blue, 'blue',
                                cell_format)

    workbook.close()
```

29.13 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.

```
#######################################################################
#
# An example of inserting images into a worksheet using the XlsxWriter
# Python module.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
```

```python
import xlsxwriter


# Create an new Excel file and add a worksheet.
workbook = xlsxwriter.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()
```

29.14 Example: Inserting images from a URL or byte stream into a worksheet

This program is an example of inserting images from a Python `io.BytesIO` byte stream into a worksheet.

The example byte streams are populated from a URL and from a local file.

See the `insert_image()` method for more details.

```
##############################################################################
#
# An example of inserting images from a Python BytesIO byte stream into a
# worksheet using the XlsxWriter module.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#

# Import the byte stream handler.
from io import BytesIO

# Import urlopen() for either Python 2 or 3.
try:
    from urllib.request import urlopen
except ImportError:
    from urllib2 import urlopen


import xlsxwriter

# Create the workbook and add a worksheet.
workbook  = xlsxwriter.Workbook('images_bytesio.xlsx')
worksheet = workbook.add_worksheet()
```

```
# Read an image from a remote url.
url = 'https://raw.githubusercontent.com/jmcnamara/XlsxWriter/' + \
      'master/examples/logo.png'

image_data = BytesIO(urlopen(url).read())

# Write the byte stream image to a cell. Note, the filename must be
# specified. In this case it will be read from url string.
worksheet.insert_image('B2', url, {'image_data': image_data})


# Read a local image file into a a byte stream. Note, the insert_image()
# method can do this directly. This is for illustration purposes only.
filename   = 'python.png'

image_file = open(filename, 'rb')
image_data = BytesIO(image_file.read())
image_file.close()


# Write the byte stream image to a cell. The filename must  be specified.
worksheet.insert_image('B8', filename, {'image_data': image_data})


workbook.close()
```

29.15 Example: Simple HTTP Server

Example of using Python and XlsxWriter to create an Excel XLSX file in an in memory string suitable for serving via SimpleHTTPServer or Django or with the Google App Engine.

Even though the final file will be in memory, via the StringIO object, the module uses temp files during assembly for efficiency. To avoid this on servers that don't allow temp files, for example the Google APP Engine, set the `in_memory` constructor option to `True`.

```
##############################################################################
#
# Example of using Python and XlsxWriter to create an Excel XLSX file in an
# in memory string suitable for serving via SimpleHTTPServer or Django
# or with the Google App Engine.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#

# Note: This is a Python 2 example. For Python 3 see http_server_py3.py.

import SimpleHTTPServer
import SocketServer
import StringIO

import xlsxwriter
```

```
class Handler(SimpleHTTPServer.SimpleHTTPRequestHandler):

    def do_GET(self):
        # Create an in-memory output file for the new workbook.
        output = StringIO.StringIO()

        # Even though the final file will be in memory the module uses temp
        # files during assembly for efficiency. To avoid this on servers that
        # don't allow temp files, for example the Google APP Engine, set the
        # 'in_memory' constructor option to True:
        workbook = xlsxwriter.Workbook(output, {'in_memory': True})
        worksheet = workbook.add_worksheet()

        # Write some test data.
        worksheet.write(0, 0, 'Hello, world!')

        # Close the workbook before streaming the data.
        workbook.close()

        # Rewind the buffer.
        output.seek(0)

        # Construct a server response.
        self.send_response(200)
        self.send_header('Content-Disposition', 'attachment; filename=test.xlsx')
        self.send_header('Content-type', 'application/vnd.openxmlformats-officedocumen
        self.end_headers()
        self.wfile.write(output.read())
        return


print('Server listening on port 8000...')
httpd = SocketServer.TCPServer(('', 8000), Handler)
httpd.serve_forever()
```

29.16 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.

```
#######################################################################
#
# 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                                      Center
#     &R                                      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
#
#     &[Picture]          Images                Image placeholder
#     &G                                        Same as &[Picture]
#
#     &&                  Miscellaneous         Literal ampersand &
#
# See the main XlsxWriter documentation for more information.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.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)


######################################################################
#
# Insert a header image.
#
worksheet2 = workbook.add_worksheet('Image')
header2 = '&L&G'

# Adjust the page top margin to allow space for the header image.
worksheet2.set_margins(top=1.3)

worksheet2.set_header(header2, {'image_left': 'python-200x80.png'})

worksheet2.set_column('A:A', 50)
worksheet2.write('A1', preview)


######################################################################
#
# This is an example of some of the header/footer variables.
#
```

```
worksheet3 = workbook.add_worksheet('Variables')
header3 = '&LPage &P of &N' + '&CFilename: &F' + '&RSheetname: &A'
footer3 = '&LCurrent date: &D' + '&RCurrent time: &T'

worksheet3.set_header(header3)
worksheet3.set_footer(footer3)

worksheet3.set_column('A:A', 50)
worksheet3.write('A1', preview)
worksheet3.write('A21', 'Next sheet')
worksheet3.set_h_pagebreaks([20])

######################################################################
#
# This example shows how to use more than one font
#
worksheet4 = workbook.add_worksheet('Mixed fonts')
header4 = '&C&"Courier New,Bold"Hello &"Arial,Italic"World'
footer4 = '&C&"Symbol"e&"Arial" = mc&X2'

worksheet4.set_header(header4)
worksheet4.set_footer(footer4)

worksheet4.set_column('A:A', 50)
worksheet4.write('A1', preview)

######################################################################
#
# Example of line wrapping
#
worksheet5 = workbook.add_worksheet('Word wrap')
header5 = "&CHeading 1\nHeading 2"

worksheet5.set_header(header5)

worksheet5.set_column('A:A', 50)
worksheet5.write('A1', preview)

######################################################################
#
# Example of inserting a literal ampersand &
#
worksheet6 = workbook.add_worksheet('Ampersand')
header6 = '&CCuriouser && Curiouser - Attorneys at Law'

worksheet6.set_header(header6)

worksheet6.set_column('A:A', 50)
worksheet6.write('A1', preview)

workbook.close()
```

29.17 Example: Freeze Panes and Split Panes

An example of how to create panes in a worksheet, both "freeze" panes and "split" panes. See the `freeze_panes()` and `split_panes()` methods for more details.

```
#######################################################################
#
# Example of using Python and the XlsxWriter module to create
# worksheet panes.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('panes.xlsx')

worksheet1 = workbook.add_worksheet('Panes 1')
worksheet2 = workbook.add_worksheet('Panes 2')
worksheet3 = workbook.add_worksheet('Panes 3')
worksheet4 = workbook.add_worksheet('Panes 4')

#######################################################################
#
# Set up some formatting and text to highlight the panes.
```

```
#
header_format = workbook.add_format({'bold': True,
                                     'align': 'center',
                                     'valign': 'vcenter',
                                     'fg_color': '#D7E4BC',
                                     'border': 1})

center_format = workbook.add_format({'align': 'center'})


######################################################################
#
# Example 1. Freeze pane on the top row.
#
worksheet1.freeze_panes(1, 0)

# Other sheet formatting.
worksheet1.set_column('A:I', 16)
worksheet1.set_row(0, 20)
worksheet1.set_selection('C3')


# Some text to demonstrate scrolling.
for col in range(0, 9):
    worksheet1.write(0, col, 'Scroll down', header_format)

for row in range(1, 100):
    for col in range(0, 9):
        worksheet1.write(row, col, row + 1, center_format)


######################################################################
#
# Example 2. Freeze pane on the left column.
#
worksheet2.freeze_panes(0, 1)

# Other sheet formatting.
worksheet2.set_column('A:A', 16)
worksheet2.set_selection('C3')

# Some text to demonstrate scrolling.
for row in range(0, 50):
    worksheet2.write(row, 0, 'Scroll right', header_format)
    for col in range(1, 26):
        worksheet2.write(row, col, col, center_format)


######################################################################
#
# Example 3. Freeze pane on the top row and left column.
#
worksheet3.freeze_panes(1, 1)
```

```
# Other sheet formatting.
worksheet3.set_column('A:Z', 16)
worksheet3.set_row(0, 20)
worksheet3.set_selection('C3')
worksheet3.write(0, 0, '', header_format)

# Some text to demonstrate scrolling.
for col in range(1, 26):
    worksheet3.write(0, col, 'Scroll down', header_format)

for row in range(1, 50):
    worksheet3.write(row, 0, 'Scroll right', header_format)
    for col in range(1, 26):
        worksheet3.write(row, col, col, center_format)


##########################################################################
#
# Example 4. Split pane on the top row and left column.
#
# The divisions must be specified in terms of row and column dimensions.
# The default row height is 15 and the default column width is 8.43
#
worksheet4.split_panes(15, 8.43)

# Other sheet formatting.
worksheet4.set_selection('C3')

# Some text to demonstrate scrolling.
for col in range(1, 26):
    worksheet4.write(0, col, 'Scroll', center_format)

for row in range(1, 50):
    worksheet4.write(row, 0, 'Scroll', center_format)
    for col in range(1, 26):
        worksheet4.write(row, col, col, center_format)

workbook.close()
```

29.18 Example: Worksheet Tables

Example of how to add tables to an XlsxWriter worksheet.

Tables in Excel are used to group rows and columns of data into a single structure that can be referenced in a formula or formatted collectively.

See also *Working with Worksheet Tables*.

```
##############################################################################
#
# Example of how to add tables to an XlsxWriter worksheet.
#
# Tables in Excel are used to group rows and columns of data into a single
# structure that can be referenced in a formula or formatted collectively.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('tables.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()
worksheet9 = workbook.add_worksheet()
worksheet10 = workbook.add_worksheet()
worksheet11 = workbook.add_worksheet()
worksheet12 = workbook.add_worksheet()
```

```
currency_format = workbook.add_format({'num_format': '$#,##0'})

# Some sample data for the table.
data = [
    ['Apples', 10000, 5000, 8000, 6000],
    ['Pears', 2000, 3000, 4000, 5000],
    ['Bananas', 6000, 6000, 6500, 6000],
    ['Oranges', 500, 300, 200, 700],

]


##############################################################################
#
# Example 1.
#
caption = 'Default table with no data.'

# Set the columns widths.
worksheet1.set_column('B:G', 12)

# Write the caption.
worksheet1.write('B1', caption)

# Add a table to the worksheet.
worksheet1.add_table('B3:F7')


##############################################################################
#
# Example 2.
#
caption = 'Default table with data.'

# Set the columns widths.
worksheet2.set_column('B:G', 12)

# Write the caption.
worksheet2.write('B1', caption)

# Add a table to the worksheet.
worksheet2.add_table('B3:F7', {'data': data})


##############################################################################
#
# Example 3.
#
caption = 'Table without default autofilter.'

# Set the columns widths.
worksheet3.set_column('B:G', 12)
```

```
# Write the caption.
worksheet3.write('B1', caption)

# Add a table to the worksheet.
worksheet3.add_table('B3:F7', {'autofilter': 0})

# Table data can also be written separately, as an array or individual cells.
worksheet3.write_row('B4', data[0])
worksheet3.write_row('B5', data[1])
worksheet3.write_row('B6', data[2])
worksheet3.write_row('B7', data[3])


##############################################################################
#
# Example 4.
#
caption = 'Table without default header row.'

# Set the columns widths.
worksheet4.set_column('B:G', 12)

# Write the caption.
worksheet4.write('B1', caption)

# Add a table to the worksheet.
worksheet4.add_table('B4:F7', {'header_row': 0})

# Table data can also be written separately, as an array or individual cells.
worksheet4.write_row('B4', data[0])
worksheet4.write_row('B5', data[1])
worksheet4.write_row('B6', data[2])
worksheet4.write_row('B7', data[3])


##############################################################################
#
# Example 5.
#
caption = 'Default table with "First Column" and "Last Column" options.'

# Set the columns widths.
worksheet5.set_column('B:G', 12)

# Write the caption.
worksheet5.write('B1', caption)

# Add a table to the worksheet.
worksheet5.add_table('B3:F7', {'first_column': 1, 'last_column': 1})

# Table data can also be written separately, as an array or individual cells.
worksheet5.write_row('B4', data[0])
worksheet5.write_row('B5', data[1])
```

```
worksheet5.write_row('B6', data[2])
worksheet5.write_row('B7', data[3])


##############################################################################
#
# Example 6.
#
caption = 'Table with banded columns but without default banded rows.'

# Set the columns widths.
worksheet6.set_column('B:G', 12)

# Write the caption.
worksheet6.write('B1', caption)

# Add a table to the worksheet.
worksheet6.add_table('B3:F7', {'banded_rows': 0, 'banded_columns': 1})

# Table data can also be written separately, as an array or individual cells.
worksheet6.write_row('B4', data[0])
worksheet6.write_row('B5', data[1])
worksheet6.write_row('B6', data[2])
worksheet6.write_row('B7', data[3])


##############################################################################
#
# Example 7.
#
caption = 'Table with user defined column headers'

# Set the columns widths.
worksheet7.set_column('B:G', 12)

# Write the caption.
worksheet7.write('B1', caption)

# Add a table to the worksheet.
worksheet7.add_table('B3:F7', {'data': data,
                               'columns': [{'header': 'Product'},
                                           {'header': 'Quarter 1'},
                                           {'header': 'Quarter 2'},
                                           {'header': 'Quarter 3'},
                                           {'header': 'Quarter 4'},
                                           ]})


##############################################################################
#
# Example 8.
#
caption = 'Table with user defined column headers'
```

```
# Set the columns widths.
worksheet8.set_column('B:G', 12)

# Write the caption.
worksheet8.write('B1', caption)

# Formula to use in the table.
formula = '=SUM(Table8[@[Quarter 1]:[Quarter 4]])'

# Add a table to the worksheet.
worksheet8.add_table('B3:G7', {'data': data,
                               'columns': [{'header': 'Product'},
                                           {'header': 'Quarter 1'},
                                           {'header': 'Quarter 2'},
                                           {'header': 'Quarter 3'},
                                           {'header': 'Quarter 4'},
                                           {'header': 'Year',
                                            'formula': formula},
                                           ]})


##############################################################################
#
# Example 9.
#
caption = 'Table with totals row (but no caption or totals).'

# Set the columns widths.
worksheet9.set_column('B:G', 12)

# Write the caption.
worksheet9.write('B1', caption)

# Formula to use in the table.
formula = '=SUM(Table9[@[Quarter 1]:[Quarter 4]])'

# Add a table to the worksheet.
worksheet9.add_table('B3:G8', {'data': data,
                               'total_row': 1,
                               'columns': [{'header': 'Product'},
                                           {'header': 'Quarter 1'},
                                           {'header': 'Quarter 2'},
                                           {'header': 'Quarter 3'},
                                           {'header': 'Quarter 4'},
                                           {'header': 'Year',
                                            'formula': formula
                                            },
                                           ]})


##############################################################################
#
# Example 10.
```

```
#
caption = 'Table with totals row with user captions and functions.'

# Set the columns widths.
worksheet10.set_column('B:G', 12)

# Write the caption.
worksheet10.write('B1', caption)

# Options to use in the table.
options = {'data': data,
           'total_row': 1,
           'columns': [{'header': 'Product', 'total_string': 'Totals'},
                       {'header': 'Quarter 1', 'total_function': 'sum'},
                       {'header': 'Quarter 2', 'total_function': 'sum'},
                       {'header': 'Quarter 3', 'total_function': 'sum'},
                       {'header': 'Quarter 4', 'total_function': 'sum'},
                       {'header': 'Year',
                        'formula': '=SUM(Table10[@[Quarter 1]:[Quarter 4]])',
                        'total_function': 'sum'
                        },
                       ]}

# Add a table to the worksheet.
worksheet10.add_table('B3:G8', options)


##############################################################################
#
# Example 11.
#
caption = 'Table with alternative Excel style.'

# Set the columns widths.
worksheet11.set_column('B:G', 12)

# Write the caption.
worksheet11.write('B1', caption)

# Options to use in the table.
options = {'data': data,
           'style': 'Table Style Light 11',
           'total_row': 1,
           'columns': [{'header': 'Product', 'total_string': 'Totals'},
                       {'header': 'Quarter 1', 'total_function': 'sum'},
                       {'header': 'Quarter 2', 'total_function': 'sum'},
                       {'header': 'Quarter 3', 'total_function': 'sum'},
                       {'header': 'Quarter 4', 'total_function': 'sum'},
                       {'header': 'Year',
                        'formula': '=SUM(Table11[@[Quarter 1]:[Quarter 4]])',
                        'total_function': 'sum'
                        },
                       ]}
```

```
# Add a table to the worksheet.
worksheet11.add_table('B3:G8', options)


##############################################################################
#
# Example 12.
#
caption = 'Table with column formats.'

# Set the columns widths.
worksheet12.set_column('B:G', 12)

# Write the caption.
worksheet12.write('B1', caption)

# Options to use in the table.
options = {'data': data,
           'total_row': 1,
           'columns': [{'header': 'Product', 'total_string': 'Totals'},
                       {'header': 'Quarter 1',
                        'total_function': 'sum',
                        'format': currency_format,
                        },
                       {'header': 'Quarter 2',
                        'total_function': 'sum',
                        'format': currency_format,
                        },
                       {'header': 'Quarter 3',
                        'total_function': 'sum',
                        'format': currency_format,
                        },
                       {'header': 'Quarter 4',
                        'total_function': 'sum',
                        'format': currency_format,
                        },
                       {'header': 'Year',
                        'formula': '=SUM(Table12[@[Quarter 1]:[Quarter 4]])',
                        'total_function': 'sum',
                        'format': currency_format,
                        },
                       ]}

# Add a table to the worksheet.
worksheet12.add_table('B3:G8', options)

workbook.close()
```

29.19 Example: Sparklines (Simple)

Example of how to add sparklines to a XlsxWriter worksheet.

Sparklines are small charts that fit in a single cell and are used to show trends in data.

See the *Working with Sparklines* method for more details.

```
##############################################################################
#
# Example of how to add sparklines to a Python XlsxWriter file.
#
# Sparklines are small charts that fit in a single cell and are
# used to show trends in data.
#
# See sparklines2.py for examples of more complex sparkline formatting.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('sparklines1.xlsx')
worksheet = workbook.add_worksheet()
```

```
# Some sample data to plot.
data = [
    [-2, 2, 3, -1, 0],
    [30, 20, 33, 20, 15],
    [1, -1, -1, 1, -1],
]


# Write the sample data to the worksheet.
worksheet.write_row('A1', data[0])
worksheet.write_row('A2', data[1])
worksheet.write_row('A3', data[2])


# Add a line sparkline (the default) with markers.
worksheet.add_sparkline('F1', {'range': 'Sheet1!A1:E1',
                               'markers': True})


# Add a column sparkline with non-default style.
worksheet.add_sparkline('F2', {'range': 'Sheet1!A2:E2',
                               'type': 'column',
                               'style': 12})


# Add a win/loss sparkline with negative values highlighted.
worksheet.add_sparkline('F3', {'range': 'Sheet1!A3:E3',
                               'type': 'win_loss',
                               'negative_points': True})

workbook.close()
```

29.20 Example: Sparklines (Advanced)

This example shows the majority of options that can be applied to sparklines.

Sparklines are small charts that fit in a single cell and are used to show trends in data.

See the *Working with Sparklines* method for more details.

sparklines2.xlsx
Home
Layout
Tables
Charts
SmartArt
A48
A
B
Sparkline
Description
A default "line" sparkline.
A default "column" sparkline.
A default "win/loss" sparkline.
Line with markers.
Line with high and low points.
Line with first and last point markers.
Line with negative point markers.
Line with axis.
Column with default style (1).
Column with style 2.
Column with style 3.
Column with style 4.
Column with style 5.
Column with style 6.
Column with a user defined colour.
A win/loss sparkline.
A win/loss sparkline with negative point
A left to right column (the default).
A right to left column.
Growth
Sparkline and text in one cell.
A grouped sparkline. Changes are applie
Sheet1
Sheet2
Normal View
Ready

```
##############################################################################
#
# Example of how to add sparklines to an XlsxWriter file with Python.
#
# Sparklines are small charts that fit in a single cell and are
# used to show trends in data. This example shows the majority of
# options that can be applied to sparklines.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('sparklines2.xlsx')
worksheet1 = workbook.add_worksheet()
worksheet2 = workbook.add_worksheet()
bold = workbook.add_format({'bold': True})
row = 1

# Set the columns widths to make the output clearer.
worksheet1.set_column('A:A', 14)
worksheet1.set_column('B:B', 50)
worksheet1.set_zoom(150)

# Headings.
worksheet1.write('A1', 'Sparkline', bold)
worksheet1.write('B1', 'Description', bold)


##############################################################################
#
text = 'A default "line" sparkline.'

worksheet1.add_sparkline('A2', {'range': 'Sheet2!A1:J1'})

worksheet1.write(row, 1, text)
row += 1


##############################################################################
#
text = 'A default "column" sparkline.'

worksheet1.add_sparkline('A3', {'range': 'Sheet2!A2:J2',
                                'type': 'column'})

worksheet1.write(row, 1, text)
row += 1


##############################################################################
#
text = 'A default "win/loss" sparkline.'
```

```
    worksheet1.add_sparkline('A4', {'range': 'Sheet2!A3:J3',
                                    'type': 'win_loss'})

    worksheet1.write(row, 1, text)
    row += 2


    ##############################################################################
    #
    text = 'Line with markers.'

    worksheet1.add_sparkline('A6', {'range': 'Sheet2!A1:J1',
                                    'markers': True})

    worksheet1.write(row, 1, text)
    row += 1


    ##############################################################################
    #
    text = 'Line with high and low points.'

    worksheet1.add_sparkline('A7', {'range': 'Sheet2!A1:J1',
                                    'high_point': True,
                                    'low_point': True})

    worksheet1.write(row, 1, text)
    row += 1


    ##############################################################################
    #
    text = 'Line with first and last point markers.'

    worksheet1.add_sparkline('A8', {'range': 'Sheet2!A1:J1',
                                    'first_point': True,
                                    'last_point': True})

    worksheet1.write(row, 1, text)
    row += 1


    ##############################################################################
    #
    text = 'Line with negative point markers.'

    worksheet1.add_sparkline('A9', {'range': 'Sheet2!A1:J1',
                                    'negative_points': True})

    worksheet1.write(row, 1, text)
    row += 1
```

```
    ##############################################################################
    #
    text = 'Line with axis.'

    worksheet1.add_sparkline('A10', {'range': 'Sheet2!A1:J1',
                                     'axis': True})

    worksheet1.write(row, 1, text)
    row += 2


    ##############################################################################
    #
    text = 'Column with default style (1).'

    worksheet1.add_sparkline('A12', {'range': 'Sheet2!A2:J2',
                                     'type': 'column'})

    worksheet1.write(row, 1, text)
    row += 1


    ##############################################################################
    #
    text = 'Column with style 2.'

    worksheet1.add_sparkline('A13', {'range': 'Sheet2!A2:J2',
                                     'type': 'column',
                                     'style': 2})

    worksheet1.write(row, 1, text)
    row += 1


    ##############################################################################
    #
    text = 'Column with style 3.'

    worksheet1.add_sparkline('A14', {'range': 'Sheet2!A2:J2',
                                     'type': 'column',
                                     'style': 3})

    worksheet1.write(row, 1, text)
    row += 1


    ##############################################################################
    #
    text = 'Column with style 4.'

    worksheet1.add_sparkline('A15', {'range': 'Sheet2!A2:J2',
                                     'type': 'column',
                                     'style': 4})
```

```
    worksheet1.write(row, 1, text)
    row += 1


    ##############################################################################
    #
    text = 'Column with style 5.'

    worksheet1.add_sparkline('A16', {'range': 'Sheet2!A2:J2',
                                     'type': 'column',
                                     'style': 5})

    worksheet1.write(row, 1, text)
    row += 1


    ##############################################################################
    #
    text = 'Column with style 6.'

    worksheet1.add_sparkline('A17', {'range': 'Sheet2!A2:J2',
                                     'type': 'column',
                                     'style': 6})

    worksheet1.write(row, 1, text)
    row += 1


    ##############################################################################
    #
    text = 'Column with a user defined colour.'

    worksheet1.add_sparkline('A18', {'range': 'Sheet2!A2:J2',
                                     'type': 'column',
                                     'series_color': '#E965E0'})

    worksheet1.write(row, 1, text)
    row += 2


    ##############################################################################
    #
    text = 'A win/loss sparkline.'

    worksheet1.add_sparkline('A20', {'range': 'Sheet2!A3:J3',
                                     'type': 'win_loss'})

    worksheet1.write(row, 1, text)
    row += 1


    ##############################################################################
    #
```

```
text = 'A win/loss sparkline with negative points highlighted.'

worksheet1.add_sparkline('A21', {'range': 'Sheet2!A3:J3',
                                 'type': 'win_loss',
                                 'negative_points': True})

worksheet1.write(row, 1, text)
row += 2


##############################################################################
#
text = 'A left to right column (the default).'

worksheet1.add_sparkline('A23', {'range': 'Sheet2!A4:J4',
                                 'type': 'column',
                                 'style': 20})

worksheet1.write(row, 1, text)
row += 1


##############################################################################
#
text = 'A right to left column.'

worksheet1.add_sparkline('A24', {'range': 'Sheet2!A4:J4',
                                 'type': 'column',
                                 'style': 20,
                                 'reverse': True})

worksheet1.write(row, 1, text)
row += 1


##############################################################################
#
text = 'Sparkline and text in one cell.'

worksheet1.add_sparkline('A25', {'range': 'Sheet2!A4:J4',
                                 'type': 'column',
                                 'style': 20})

worksheet1.write(row, 0, 'Growth')
worksheet1.write(row, 1, text)
row += 2


##############################################################################
#
text = 'A grouped sparkline. Changes are applied to all three.'

worksheet1.add_sparkline('A27', {'location': ['A27', 'A28', 'A29'],
```

```
                                    'range': ['Sheet2!A5:J5',
                                              'Sheet2!A6:J6',
                                              'Sheet2!A7:J7'],
                                    'markers': True})

    worksheet1.write(row, 1, text)
    row += 1


    ##########################################################################
    #
    # Create a second worksheet with data to plot.
    #
    worksheet2.set_column('A:J', 11)

    data = [

        # Simple line data.
        [-2, 2, 3, -1, 0, -2, 3, 2, 1, 0],

        # Simple column data.
        [30, 20, 33, 20, 15, 5, 5, 15, 10, 15],

        # Simple win/loss data.
        [1, 1, -1, -1, 1, -1, 1, 1, 1, -1],

        # Unbalanced histogram.
        [5, 6, 7, 10, 15, 20, 30, 50, 70, 100],

        # Data for the grouped sparkline example.
        [-2, 2, 3, -1, 0, -2, 3, 2, 1, 0],
        [3, -1, 0, -2, 3, 2, 1, 0, 2, 1],
        [0, -2, 3, 2, 1, 0, 1, 2, 3, 1],

    ]

    # Write the sample data to the worksheet.
    worksheet2.write_row('A1', data[0])
    worksheet2.write_row('A2', data[1])
    worksheet2.write_row('A3', data[2])
    worksheet2.write_row('A4', data[3])
    worksheet2.write_row('A5', data[4])
    worksheet2.write_row('A6', data[5])
    worksheet2.write_row('A7', data[6])

    workbook.close()
```

29.21 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*.

```
##############################################################################
#
# An example of writing cell comments to a worksheet using Python and
# XlsxWriter.
#
# For more advanced comment options see comments2.py.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('comments1.xlsx')
worksheet = workbook.add_worksheet()

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

workbook.close()
```

29.22 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*.

```
#######################################################################
#
# An example of writing cell comments to a worksheet using Python and
# XlsxWriter.
#
# Each of the worksheets demonstrates different features of cell comments.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.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()
```

29.23 Example: Outline and Grouping

Examples of how use XlsxWriter to generate Excel outlines and grouping. See also *Working with Outlines and Grouping*.

```
##############################################################################
#
# Example of how use Python and XlsxWriter to generate Excel outlines and
# grouping.
#
# Excel allows you to group rows or columns so that they can be hidden or
# displayed with a single mouse click. This feature is referred to as outlines.
#
# Outlines can reduce complex data down to a few salient sub-totals or
# summaries.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

# Create a new workbook and add some worksheets
workbook = xlsxwriter.Workbook('outline.xlsx')
worksheet1 = workbook.add_worksheet('Outlined Rows')
worksheet2 = workbook.add_worksheet('Collapsed Rows')
worksheet3 = workbook.add_worksheet('Outline Columns')
worksheet4 = workbook.add_worksheet('Outline levels')

# Add a general format
bold = workbook.add_format({'bold': 1})
```

```
##############################################################################
#
# Example 1: A worksheet with outlined rows. It also includes SUBTOTAL()
# functions so that it looks like the type of automatic outlines that are
# generated when you use the Excel Data->SubTotals menu item.
#
# For outlines the important parameters are 'level' and 'hidden'. Rows with
# the same 'level' are grouped together. The group will be collapsed if
# 'hidden' is enabled. The parameters 'height' and 'cell_format' are assigned
# default values if they are None.
#
worksheet1.set_row(1, None, None, {'level': 2})
worksheet1.set_row(2, None, None, {'level': 2})
worksheet1.set_row(3, None, None, {'level': 2})
worksheet1.set_row(4, None, None, {'level': 2})
worksheet1.set_row(5, None, None, {'level': 1})

worksheet1.set_row(6, None, None, {'level': 2})
worksheet1.set_row(7, None, None, {'level': 2})
worksheet1.set_row(8, None, None, {'level': 2})
worksheet1.set_row(9, None, None, {'level': 2})
worksheet1.set_row(10, None, None, {'level': 1})

# Adjust the column width for clarity
worksheet1.set_column('A:A', 20)

# Add the data, labels and formulas
worksheet1.write('A1', 'Region', bold)
worksheet1.write('A2', 'North')
worksheet1.write('A3', 'North')
worksheet1.write('A4', 'North')
worksheet1.write('A5', 'North')
worksheet1.write('A6', 'North Total', bold)

worksheet1.write('B1', 'Sales', bold)
worksheet1.write('B2', 1000)
worksheet1.write('B3', 1200)
worksheet1.write('B4', 900)
worksheet1.write('B5', 1200)
worksheet1.write('B6', '=SUBTOTAL(9,B2:B5)', bold)

worksheet1.write('A7', 'South')
worksheet1.write('A8', 'South')
worksheet1.write('A9', 'South')
worksheet1.write('A10', 'South')
worksheet1.write('A11', 'South Total', bold)

worksheet1.write('B7', 400)
worksheet1.write('B8', 600)
worksheet1.write('B9', 500)
worksheet1.write('B10', 600)
worksheet1.write('B11', '=SUBTOTAL(9,B7:B10)', bold)
```

```
worksheet1.write('A12', 'Grand Total', bold)
worksheet1.write('B12', '=SUBTOTAL(9,B2:B10)', bold)


##############################################################################
#
# Example 2: A worksheet with outlined rows. This is the same as the
# previous example except that the rows are collapsed.
# Note: We need to indicate the rows that contains the collapsed symbol '+'
# with the optional parameter, 'collapsed'. The group will be then be
# collapsed if 'hidden' is True.
#
worksheet2.set_row(1, None, None, {'level': 2, 'hidden': True})
worksheet2.set_row(2, None, None, {'level': 2, 'hidden': True})
worksheet2.set_row(3, None, None, {'level': 2, 'hidden': True})
worksheet2.set_row(4, None, None, {'level': 2, 'hidden': True})
worksheet2.set_row(5, None, None, {'level': 1, 'hidden': True})

worksheet2.set_row(6, None, None, {'level': 2, 'hidden': True})
worksheet2.set_row(7, None, None, {'level': 2, 'hidden': True})
worksheet2.set_row(8, None, None, {'level': 2, 'hidden': True})
worksheet2.set_row(9, None, None, {'level': 2, 'hidden': True})
worksheet2.set_row(10, None, None, {'level': 1, 'hidden': True})
worksheet2.set_row(11, None, None, {'collapsed': True})

# Adjust the column width for clarity
worksheet2.set_column('A:A', 20)

# Add the data, labels and formulas
worksheet2.write('A1', 'Region', bold)
worksheet2.write('A2', 'North')
worksheet2.write('A3', 'North')
worksheet2.write('A4', 'North')
worksheet2.write('A5', 'North')
worksheet2.write('A6', 'North Total', bold)

worksheet2.write('B1', 'Sales', bold)
worksheet2.write('B2', 1000)
worksheet2.write('B3', 1200)
worksheet2.write('B4', 900)
worksheet2.write('B5', 1200)
worksheet2.write('B6', '=SUBTOTAL(9,B2:B5)', bold)

worksheet2.write('A7', 'South')
worksheet2.write('A8', 'South')
worksheet2.write('A9', 'South')
worksheet2.write('A10', 'South')
worksheet2.write('A11', 'South Total', bold)

worksheet2.write('B7', 400)
worksheet2.write('B8', 600)
worksheet2.write('B9', 500)
worksheet2.write('B10', 600)
```

```
worksheet2.write('B11', '=SUBTOTAL(9,B7:B10)', bold)

worksheet2.write('A12', 'Grand Total', bold)
worksheet2.write('B12', '=SUBTOTAL(9,B2:B10)', bold)


##############################################################################
#
# Example 3: Create a worksheet with outlined columns.
#
data = [
    ['Month', 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Total'],
    ['North', 50, 20, 15, 25, 65, 80, '=SUM(B2:G2)'],
    ['South', 10, 20, 30, 50, 50, 50, '=SUM(B3:G3)'],
    ['East', 45, 75, 50, 15, 75, 100, '=SUM(B4:G4)'],
    ['West', 15, 15, 55, 35, 20, 50, '=SUM(B5:G5)']]

# Add bold format to the first row.
worksheet3.set_row(0, None, bold)

# Set column formatting and the outline level.
worksheet3.set_column('A:A', 10, bold)
worksheet3.set_column('B:G', 5, None, {'level': 1})
worksheet3.set_column('H:H', 10)

# Write the data and a formula
for row, data_row in enumerate(data):
    worksheet3.write_row(row, 0, data_row)

worksheet3.write('H6', '=SUM(H2:H5)', bold)


##############################################################################
#
# Example 4: Show all possible outline levels.
#
levels = [
    'Level 1', 'Level 2', 'Level 3', 'Level 4', 'Level 5', 'Level 6',
    'Level 7', 'Level 6', 'Level 5', 'Level 4', 'Level 3', 'Level 2',
    'Level 1']

worksheet4.write_column('A1', levels)

worksheet4.set_row(0, None, None, {'level': 1})
worksheet4.set_row(1, None, None, {'level': 2})
worksheet4.set_row(2, None, None, {'level': 3})
worksheet4.set_row(3, None, None, {'level': 4})
worksheet4.set_row(4, None, None, {'level': 5})
worksheet4.set_row(5, None, None, {'level': 6})
worksheet4.set_row(6, None, None, {'level': 7})
worksheet4.set_row(7, None, None, {'level': 6})
worksheet4.set_row(8, None, None, {'level': 5})
worksheet4.set_row(9, None, None, {'level': 4})
```

```
worksheet4.set_row(10, None, None, {'level': 3})
worksheet4.set_row(11, None, None, {'level': 2})
worksheet4.set_row(12, None, None, {'level': 1})

workbook.close()
```

29.24 Example: Collapsed Outline and Grouping

Examples of how use XlsxWriter to generate Excel outlines and grouping. These examples focus mainly on collapsed outlines. See also *Working with Outlines and Grouping*.

```
#######################################################################
#
# Example of how to use Python and XlsxWriter to generate Excel outlines and
# grouping.
#
# These examples focus mainly on collapsed outlines. See also the
# outlines.py example program for more general examples.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter
```

```
# Create a new workbook and add some worksheets
workbook = xlsxwriter.Workbook('outline_collapsed.xlsx')
worksheet1 = workbook.add_worksheet('Outlined Rows')
worksheet2 = workbook.add_worksheet('Collapsed Rows 1')
worksheet3 = workbook.add_worksheet('Collapsed Rows 2')
worksheet4 = workbook.add_worksheet('Collapsed Rows 3')
worksheet5 = workbook.add_worksheet('Outline Columns')
worksheet6 = workbook.add_worksheet('Collapsed Columns')

# Add a general format
bold = workbook.add_format({'bold': 1})


# This function will generate the same data and sub-totals on each worksheet.
# Used in the first 4 examples.
#
def create_sub_totals(worksheet):
    # Adjust the column width for clarity.
    worksheet.set_column('A:A', 20)

    # Add the data, labels and formulas.
    worksheet.write('A1', 'Region', bold)
    worksheet.write('A2', 'North')
    worksheet.write('A3', 'North')
    worksheet.write('A4', 'North')
    worksheet.write('A5', 'North')
    worksheet.write('A6', 'North Total', bold)

    worksheet.write('B1', 'Sales', bold)
    worksheet.write('B2', 1000)
    worksheet.write('B3', 1200)
    worksheet.write('B4', 900)
    worksheet.write('B5', 1200)
    worksheet.write('B6', '=SUBTOTAL(9,B2:B5)', bold)

    worksheet.write('A7', 'South')
    worksheet.write('A8', 'South')
    worksheet.write('A9', 'South')
    worksheet.write('A10', 'South')
    worksheet.write('A11', 'South Total', bold)

    worksheet.write('B7', 400)
    worksheet.write('B8', 600)
    worksheet.write('B9', 500)
    worksheet.write('B10', 600)
    worksheet.write('B11', '=SUBTOTAL(9,B7:B10)', bold)

    worksheet.write('A12', 'Grand Total', bold)
    worksheet.write('B12', '=SUBTOTAL(9,B2:B10)', bold)

##############################################################################
#
# Example 1: A worksheet with outlined rows. It also includes SUBTOTAL()
```

```
# functions so that it looks like the type of automatic outlines that are
# generated when you use the Excel Data->SubTotals menu item.
#
worksheet1.set_row(1, None, None, {'level': 2})
worksheet1.set_row(2, None, None, {'level': 2})
worksheet1.set_row(3, None, None, {'level': 2})
worksheet1.set_row(4, None, None, {'level': 2})
worksheet1.set_row(5, None, None, {'level': 1})

worksheet1.set_row(6, None, None, {'level': 2})
worksheet1.set_row(7, None, None, {'level': 2})
worksheet1.set_row(8, None, None, {'level': 2})
worksheet1.set_row(9, None, None, {'level': 2})
worksheet1.set_row(10, None, None, {'level': 1})

# Write the sub-total data that is common to the row examples.
create_sub_totals(worksheet1)


##############################################################################
#
# Example 2: Create a worksheet with collapsed outlined rows.
# This is the same as the example 1  except that the all rows are collapsed.
# Note: We need to indicate the rows that contains the collapsed symbol '+'
# with the optional parameter, 'collapsed'.
#
worksheet2.set_row(1, None, None, {'level': 2, 'hidden': True})
worksheet2.set_row(2, None, None, {'level': 2, 'hidden': True})
worksheet2.set_row(3, None, None, {'level': 2, 'hidden': True})
worksheet2.set_row(4, None, None, {'level': 2, 'hidden': True})
worksheet2.set_row(5, None, None, {'level': 1, 'hidden': True})

worksheet2.set_row(6, None, None, {'level': 2, 'hidden': True})
worksheet2.set_row(7, None, None, {'level': 2, 'hidden': True})
worksheet2.set_row(8, None, None, {'level': 2, 'hidden': True})
worksheet2.set_row(9, None, None, {'level': 2, 'hidden': True})
worksheet2.set_row(10, None, None, {'level': 1, 'hidden': True})

worksheet2.set_row(11, None, None, {'collapsed': True})

# Write the sub-total data that is common to the row examples.
create_sub_totals(worksheet2)


##############################################################################
#
# Example 3: Create a worksheet with collapsed outlined rows.
# Same as the example 1  except that the two sub-totals are collapsed.
#
worksheet3.set_row(1, None, None, {'level': 2, 'hidden': True})
worksheet3.set_row(2, None, None, {'level': 2, 'hidden': True})
worksheet3.set_row(3, None, None, {'level': 2, 'hidden': True})
worksheet3.set_row(4, None, None, {'level': 2, 'hidden': True})
```

```
worksheet3.set_row(5, None, None, {'level': 1, 'collapsed': True})

worksheet3.set_row(6, None, None, {'level': 2, 'hidden': True})
worksheet3.set_row(7, None, None, {'level': 2, 'hidden': True})
worksheet3.set_row(8, None, None, {'level': 2, 'hidden': True})
worksheet3.set_row(9, None, None, {'level': 2, 'hidden': True})
worksheet3.set_row(10, None, None, {'level': 1, 'collapsed': True})

# Write the sub-total data that is common to the row examples.
create_sub_totals(worksheet3)


##############################################################################
#
# Example 4: Create a worksheet with outlined rows.
# Same as the example 1  except that the two sub-totals are collapsed.
#
worksheet4.set_row(1, None, None, {'level': 2, 'hidden': True})
worksheet4.set_row(2, None, None, {'level': 2, 'hidden': True})
worksheet4.set_row(3, None, None, {'level': 2, 'hidden': True})
worksheet4.set_row(4, None, None, {'level': 2, 'hidden': True})
worksheet4.set_row(5, None, None, {'level': 1, 'hidden': True,
                                   'collapsed': True})

worksheet4.set_row(6, None, None, {'level': 2, 'hidden': True})
worksheet4.set_row(7, None, None, {'level': 2, 'hidden': True})
worksheet4.set_row(8, None, None, {'level': 2, 'hidden': True})
worksheet4.set_row(9, None, None, {'level': 2, 'hidden': True})
worksheet4.set_row(10, None, None, {'level': 1, 'hidden': True,
                                    'collapsed': True})

worksheet4.set_row(11, None, None, {'collapsed': True})


# Write the sub-total data that is common to the row examples.
create_sub_totals(worksheet4)


##############################################################################
#
# Example 5: Create a worksheet with outlined columns.
#
data = [
    ['Month', 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Total'],
    ['North', 50, 20, 15, 25, 65, 80, '=SUM(B2:G2)'],
    ['South', 10, 20, 30, 50, 50, 50, '=SUM(B3:G3)'],
    ['East', 45, 75, 50, 15, 75, 100, '=SUM(B4:G4)'],
    ['West', 15, 15, 55, 35, 20, 50, '=SUM(B5:G5)']]

# Add bold format to the first row.
worksheet5.set_row(0, None, bold)

# Set column formatting and the outline level.
```

```
worksheet5.set_column('A:A', 10, bold)
worksheet5.set_column('B:G', 5, None, {'level': 1})
worksheet5.set_column('H:H', 10)

# Write the data and a formula.
for row, data_row in enumerate(data):
    worksheet5.write_row(row, 0, data_row)

worksheet5.write('H6', '=SUM(H2:H5)', bold)

##############################################################################
#
# Example 6: Create a worksheet with collapsed outlined columns.
# This is the same as the previous example except with collapsed columns.
#

# Reuse the data from the previous example.

# Add bold format to the first row.
worksheet6.set_row(0, None, bold)

# Set column formatting and the outline level.
worksheet6.set_column('A:A', 10, bold)
worksheet6.set_column('B:G', 5, None, {'level': 1, 'hidden': True})
worksheet6.set_column('H:H', 10, None, {'collapsed': True})

# Write the data and a formula.
for row, data_row in enumerate(data):
    worksheet6.write_row(row, 0, data_row)

worksheet6.write('H6', '=SUM(H2:H5)', bold)

workbook.close()
```

29.25 Example: Setting Document Properties

This program is an example setting document properties. See the `set_properties()` workbook method for more details.

```
##############################################################################
#
# An example of adding document properties to a XlsxWriter file.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.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.")

    workbook.close()
```

29.26 Example: Simple Unicode with Python 2

To write Unicode text in UTF-8 to a xlsxwriter file in Python 2:

1. Encode the file as UTF-8.
2. Include a "coding" directive at the start of the file.
3. Use u'' to indicate a Unicode string.

29.27 Example: Simple Unicode with Python 3

To write Unicode text in UTF-8 to a xlsxwriter file in Python 3:

1. Encode the file as UTF-8.

29.28 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 take 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.

```
##############################################################################
#
# 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 2013-2015, John McNamara, jmcnamara@cpan.org
#
import codecs
import xlsxwriter

# 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 = xlsxwriter.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()
```

29.29 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 take 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.

```
##############################################################################
#
# 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 2013-2015, John McNamara, jmcnamara@cpan.org
#
import codecs
import xlsxwriter

# 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 = xlsxwriter.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()
```

29.30 Example: Setting Worksheet Tab Colours

This program is an example of setting worksheet tab colours. See the `set_tab_color()` method for more details.

```
#######################################################################
#
# Example of how to set Excel worksheet tab colours using Python
# and the XlsxWriter module..
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.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()
```

29.31 Example: Diagonal borders in cells

Example of how to set diagonal borders in a cell.

See `set_diag_border()`, `set_diag_type()` and `set_diag_border()` for details.

```
##############################################################################
#
# A simple formatting example that demonstrates how to add diagonal cell
# borders with XlsxWriter.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('diag_border.xlsx')
worksheet = workbook.add_worksheet()

format1 = workbook.add_format({'diag_type': 1})
format2 = workbook.add_format({'diag_type': 2})
format3 = workbook.add_format({'diag_type': 3})

format4 = workbook.add_format({
    'diag_type': 3,
    'diag_border': 7,
    'diag_color': 'red',
})
```

```
worksheet.write('B3', 'Text', format1)
worksheet.write('B6', 'Text', format2)
worksheet.write('B9', 'Text', format3)
worksheet.write('B12', 'Text', format4)

workbook.close()
```

29.32 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.

```
#######################################################################
#
# Example of cell locking and formula hiding in an Excel worksheet
# using Python and the XlsxWriter module.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter
```

```
workbook = xlsxwriter.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()
```

29.33 Example: Hiding Worksheets

This program is an example of how to hide a worksheet using the `hide()` method.

```
#######################################################################
#
# Example of how to hide a worksheet with XlsxWriter.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('hide_sheet.xlsx')
worksheet1 = workbook.add_worksheet()
worksheet2 = workbook.add_worksheet()
worksheet3 = workbook.add_worksheet()

worksheet1.set_column('A:A', 30)
worksheet2.set_column('A:A', 30)
worksheet3.set_column('A:A', 30)

# Hide Sheet2. It won't be visible until it is unhidden in Excel.
worksheet2.hide()

worksheet1.write('A1', 'Sheet2 is hidden')
worksheet2.write('A1', "Now it's my turn to find you!")
worksheet3.write('A1', 'Sheet2 is hidden')
```

```
workbook.close()
```

29.34 Example: Hiding Rows and Columns

This program is an example of how to hide rows and columns in XlsxWriter.

An individual row can be hidden using the `set_row()` method:

```
worksheet.set_row(0, None, None, {'hidden': True})
```

However, in order to hide a large number of rows, for example all the rows after row 8, we need to use an Excel optimisation to hide rows without setting each one, (of approximately 1 million rows). To do this we use the `set_default_row()` method.

Columns don't require this optimisation and can be hidden using `set_column()`.

```
##############################################################################
#
# Example of how to hide rows and columns in XlsxWriter. In order to
# hide rows without setting each one, (of approximately 1 million rows),
# Excel uses an optimisation to hide all rows that don't have data.
#
```

```
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('hide_row_col.xlsx')
worksheet = workbook.add_worksheet()

# Write some data.
worksheet.write('D1', 'Some hidden columns.')
worksheet.write('A8', 'Some hidden rows.')

# Hide all rows without data.
worksheet.set_default_row(hide_unused_rows=True)

# Set the height of empty rows that we do want to display even if it is
# the default height.
for row in range(1, 7):
    worksheet.set_row(row, 15)

# Columns can be hidden explicitly. This doesn't increase the file size..
worksheet.set_column('G:XFD', None, None, {'hidden': True})

workbook.close()
```

29.35 Example: Adding a VBA macro to a Workbook

This program is an example of how to add a button connected to a VBA macro to a worksheet.

See *Working with VBA Macros* for more details.

```
#######################################################################
#
# An example of adding macros to an XlsxWriter file using a VBA project
# file extracted from an existing Excel xlsm file.
#
# The vba_extract.py utility supplied with XlsxWriter can be used to extract
# the vbaProject.bin file.
#
# An embedded macro is connected to a form button on the worksheet.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

# Note the file extension should be .xlsm.
workbook = xlsxwriter.Workbook('macros.xlsm')
worksheet = workbook.add_worksheet()

worksheet.set_column('A:A', 30)

# Add the VBA project binary.
workbook.add_vba_project('./vbaProject.bin')
```

```
# Show text for the end user.
worksheet.write('A3', 'Press the button to say hello.')

# Add a button tied to a macro in the VBA project.
worksheet.insert_button('B3',{'macro': 'say_hello',
                              'caption': 'Press Me',
                              'width': 80,
                              'height': 30})

workbook.close()
```

CHAPTER

THIRTY

CHART EXAMPLES

The following are some of the examples included in the examples directory of the XlsxWriter distribution.

30.1 Example: Chart (Simple)

Example of a simple column chart with 3 data series:

See the *The Chart Class* and *Working with Charts* for more details.

```
#######################################################################
#
# An example of a simple Excel chart with Python and XlsxWriter.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('chart.xlsx')
worksheet = workbook.add_worksheet()

# Create a new Chart object.
```

```
chart = workbook.add_chart({'type': 'column'})

# Write some data to add to plot on the chart.
data = [
    [1, 2, 3, 4, 5],
    [2, 4, 6, 8, 10],
    [3, 6, 9, 12, 15],
]

worksheet.write_column('A1', data[0])
worksheet.write_column('B1', data[1])
worksheet.write_column('C1', data[2])

# Configure the charts. In simplest case we just add some data series.
chart.add_series({'values': '=Sheet1!$A$1:$A$5'})
chart.add_series({'values': '=Sheet1!$B$1:$B$5'})
chart.add_series({'values': '=Sheet1!$C$1:$C$5'})

# Insert the chart into the worksheet.
worksheet.insert_chart('A7', chart)

workbook.close()
```

30.2 Example: Area Chart

Example of creating Excel Area charts.

Chart 1 in the following example is a default area chart:

Chart 2 is a stacked area chart:

Chart 3 is a percentage stacked area chart:

```
#######################################################################
#
# An example of creating Excel Area charts with Python and XlsxWriter.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_area.xlsx')
worksheet = workbook.add_worksheet()
bold = workbook.add_format({'bold': 1})

# Add the worksheet data that the charts will refer to.
headings = ['Number', 'Batch 1', 'Batch 2']
data = [
    [2, 3, 4, 5, 6, 7],
    [40, 40, 50, 30, 25, 50],
```

```
    [30, 25, 30, 10, 5, 10],
]

worksheet.write_row('A1', headings, bold)
worksheet.write_column('A2', data[0])
worksheet.write_column('B2', data[1])
worksheet.write_column('C2', data[2])

######################################################################
#
# Create an area chart.
#
chart1 = workbook.add_chart({'type': 'area'})

# Configure the first series.
chart1.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure a second series. Note use of alternative syntax to define ranges.
chart1.add_series({
    'name':       ['Sheet1', 0, 2],
    'categories': ['Sheet1', 1, 0, 6, 0],
    'values':     ['Sheet1', 1, 2, 6, 2],
})

# Add a chart title and some axis labels.
chart1.set_title ({'name': 'Results of sample analysis'})
chart1.set_x_axis({'name': 'Test number'})
chart1.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart1.set_style(11)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D2', chart1, {'x_offset': 25, 'y_offset': 10})

######################################################################
#
# Create a stacked area chart sub-type.
#
chart2 = workbook.add_chart({'type': 'area', 'subtype': 'stacked'})

# Configure the first series.
chart2.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure second series.
```

```
chart2.add_series({
    'name':       '=Sheet1!$C$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$C$2:$C$7',
})

# Add a chart title and some axis labels.
chart2.set_title ({'name': 'Stacked Chart'})
chart2.set_x_axis({'name': 'Test number'})
chart2.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart2.set_style(12)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D18', chart2, {'x_offset': 25, 'y_offset': 10})

#######################################################################
#
# Create a percent stacked area chart sub-type.
#
chart3 = workbook.add_chart({'type': 'area', 'subtype': 'percent_stacked'})

# Configure the first series.
chart3.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure second series.
chart3.add_series({
    'name':       '=Sheet1!$C$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$C$2:$C$7',
})

# Add a chart title and some axis labels.
chart3.set_title ({'name': 'Percent Stacked Chart'})
chart3.set_x_axis({'name': 'Test number'})
chart3.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart3.set_style(13)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D34', chart3, {'x_offset': 25, 'y_offset': 10})

workbook.close()
```

30.3 Example: Bar Chart

Example of creating Excel Bar charts.

Chart 1 in the following example is a default bar chart:

Chart 2 is a stacked bar chart:

Chart 3 is a percentage stacked bar chart:

```
#######################################################################
#
# An example of creating Excel Bar charts with Python and XlsxWriter.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_bar.xlsx')
worksheet = workbook.add_worksheet()
bold = workbook.add_format({'bold': 1})

# Add the worksheet data that the charts will refer to.
headings = ['Number', 'Batch 1', 'Batch 2']
data = [
    [2, 3, 4, 5, 6, 7],
    [10, 40, 50, 20, 10, 50],
    [30, 60, 70, 50, 40, 30],
]

worksheet.write_row('A1', headings, bold)
worksheet.write_column('A2', data[0])
worksheet.write_column('B2', data[1])
worksheet.write_column('C2', data[2])

#######################################################################
#
# Create a new bar chart.
#
chart1 = workbook.add_chart({'type': 'bar'})

# Configure the first series.
chart1.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
```

```
})

# Configure a second series. Note use of alternative syntax to define ranges.
chart1.add_series({
    'name':       ['Sheet1', 0, 2],
    'categories': ['Sheet1', 1, 0, 6, 0],
    'values':     ['Sheet1', 1, 2, 6, 2],
})

# Add a chart title and some axis labels.
chart1.set_title ({'name': 'Results of sample analysis'})
chart1.set_x_axis({'name': 'Test number'})
chart1.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart1.set_style(11)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D2', chart1, {'x_offset': 25, 'y_offset': 10})

#######################################################################
#
# Create a stacked chart sub-type.
#
chart2 = workbook.add_chart({'type': 'bar', 'subtype': 'stacked'})

# Configure the first series.
chart2.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure second series.
chart2.add_series({
    'name':       '=Sheet1!$C$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$C$2:$C$7',
})

# Add a chart title and some axis labels.
chart2.set_title ({'name': 'Stacked Chart'})
chart2.set_x_axis({'name': 'Test number'})
chart2.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart2.set_style(12)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D18', chart2, {'x_offset': 25, 'y_offset': 10})

#######################################################################
#
```

```
# Create a percentage stacked chart sub-type.
#
chart3 = workbook.add_chart({'type': 'bar', 'subtype': 'percent_stacked'})

# Configure the first series.
chart3.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure second series.
chart3.add_series({
    'name':       '=Sheet1!$C$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$C$2:$C$7',
})

# Add a chart title and some axis labels.
chart3.set_title ({'name': 'Percent Stacked Chart'})
chart3.set_x_axis({'name': 'Test number'})
chart3.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart3.set_style(13)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D34', chart3, {'x_offset': 25, 'y_offset': 10})

workbook.close()
```

30.4 Example: Column Chart

Example of creating Excel Column charts.

Chart 1 in the following example is a default column chart:

Chart 2 is a stacked column chart:

Chart 3 is a percentage stacked column chart:

```
#######################################################################
#
# An example of creating Excel Column charts with Python and XlsxWriter.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_column.xlsx')
worksheet = workbook.add_worksheet()
bold = workbook.add_format({'bold': 1})

# Add the worksheet data that the charts will refer to.
headings = ['Number', 'Batch 1', 'Batch 2']
data = [
    [2, 3, 4, 5, 6, 7],
    [10, 40, 50, 20, 10, 50],
    [30, 60, 70, 50, 40, 30],
]

worksheet.write_row('A1', headings, bold)
worksheet.write_column('A2', data[0])
worksheet.write_column('B2', data[1])
worksheet.write_column('C2', data[2])

#######################################################################
#
# Create a new column chart.
#
chart1 = workbook.add_chart({'type': 'column'})

# Configure the first series.
chart1.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
```

```
})

# Configure a second series. Note use of alternative syntax to define ranges.
chart1.add_series({
    'name':       ['Sheet1', 0, 2],
    'categories': ['Sheet1', 1, 0, 6, 0],
    'values':     ['Sheet1', 1, 2, 6, 2],
})

# Add a chart title and some axis labels.
chart1.set_title ({'name': 'Results of sample analysis'})
chart1.set_x_axis({'name': 'Test number'})
chart1.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart1.set_style(11)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D2', chart1, {'x_offset': 25, 'y_offset': 10})

#######################################################################
#
# Create a stacked chart sub-type.
#
chart2 = workbook.add_chart({'type': 'column', 'subtype': 'stacked'})

# Configure the first series.
chart2.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure second series.
chart2.add_series({
    'name':       '=Sheet1!$C$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$C$2:$C$7',
})

# Add a chart title and some axis labels.
chart2.set_title ({'name': 'Stacked Chart'})
chart2.set_x_axis({'name': 'Test number'})
chart2.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart2.set_style(12)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D18', chart2, {'x_offset': 25, 'y_offset': 10})

#######################################################################
#
```

```
# Create a percentage stacked chart sub-type.
#
chart3 = workbook.add_chart({'type': 'column', 'subtype': 'percent_stacked'})

# Configure the first series.
chart3.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure second series.
chart3.add_series({
    'name':       '=Sheet1!$C$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$C$2:$C$7',
})

# Add a chart title and some axis labels.
chart3.set_title ({'name': 'Percent Stacked Chart'})
chart3.set_x_axis({'name': 'Test number'})
chart3.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart3.set_style(13)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D34', chart3, {'x_offset': 25, 'y_offset': 10})

workbook.close()
```

30.5 Example: Line Chart

Example of creating an Excel line chart. The X axis of a line chart is a category axis with fixed point spacing. For a line chart with arbitrary point spacing see the Scatter chart type.

Chart 1 in the following example is:

```
#######################################################################
#
# An example of creating Excel Line charts with Python and XlsxWriter.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_line.xlsx')
worksheet = workbook.add_worksheet()
bold = workbook.add_format({'bold': 1})

# Add the worksheet data that the charts will refer to.
headings = ['Number', 'Batch 1', 'Batch 2']
data = [
    [2, 3, 4, 5, 6, 7],
    [10, 40, 50, 20, 10, 50],
    [30, 60, 70, 50, 40, 30],
]

worksheet.write_row('A1', headings, bold)
worksheet.write_column('A2', data[0])
worksheet.write_column('B2', data[1])
worksheet.write_column('C2', data[2])

# Create a new chart object. In this case an embedded chart.
chart1 = workbook.add_chart({'type': 'line'})

# Configure the first series.
chart1.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure second series. Note use of alternative syntax to define ranges.
```

```
chart1.add_series({
    'name':       ['Sheet1', 0, 2],
    'categories': ['Sheet1', 1, 0, 6, 0],
    'values':     ['Sheet1', 1, 2, 6, 2],
})

# Add a chart title and some axis labels.
chart1.set_title ({'name': 'Results of sample analysis'})
chart1.set_x_axis({'name': 'Test number'})
chart1.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style. Colors with white outline and shadow.
chart1.set_style(10)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D2', chart1, {'x_offset': 25, 'y_offset': 10})


workbook.close()
```

30.6 Example: Pie Chart

Example of creating Excel Pie charts. Chart 1 in the following example is:

Chart 2 shows how to set segment colours.

It is possible to define chart colours for most types of XlsxWriter charts via the `add_series()` method. However, Pie charts are a special case since each segment is represented as a point and as such it is necessary to assign formatting to each point in the series.

```
#######################################################################
#
# An example of creating Excel Pie charts with Python and XlsxWriter.
#
# The demo also shows how to set segment colours. It is possible to
# define chart colors for most types of XlsxWriter charts
# via the add_series() method. However, Pie/Doughnut charts are a special
# case since each segment is represented as a point so it is necessary to
# assign formatting to each point in the series.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_pie.xlsx')

worksheet = workbook.add_worksheet()
bold = workbook.add_format({'bold': 1})

# Add the worksheet data that the charts will refer to.
headings = ['Category', 'Values']
data = [
    ['Apple', 'Cherry', 'Pecan'],
    [60, 30, 10],
]

worksheet.write_row('A1', headings, bold)
worksheet.write_column('A2', data[0])
worksheet.write_column('B2', data[1])

#######################################################################
#
# Create a new chart object.
#
chart1 = workbook.add_chart({'type': 'pie'})
```

```
# Configure the series. Note the use of the list syntax to define ranges:
chart1.add_series({
    'name':       'Pie sales data',
    'categories': ['Sheet1', 1, 0, 3, 0],
    'values':     ['Sheet1', 1, 1, 3, 1],
})

# Add a title.
chart1.set_title({'name': 'Popular Pie Types'})

# Set an Excel chart style. Colors with white outline and shadow.
chart1.set_style(10)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('C2', chart1, {'x_offset': 25, 'y_offset': 10})

######################################################################
#
# Create a Pie chart with user defined segment colors.
#

# Create an example Pie chart like above.
chart2 = workbook.add_chart({'type': 'pie'})

# Configure the series and add user defined segment colours.
chart2.add_series({
    'name': 'Pie sales data',
    'categories': '=Sheet1!A2:A4',
    'values':     '=Sheet1!B2:B4',
    'points': [
        {'fill': {'color': '#5ABA10'}},
        {'fill': {'color': '#FE110E'}},
        {'fill': {'color': '#CA5C05'}},
    ],
})

# Add a title.
chart2.set_title({'name': 'Pie Chart with user defined colors'})

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('C18', chart2, {'x_offset': 25, 'y_offset': 10})

######################################################################
#
# Create a Pie chart with rotation of the segments.
#

# Create an example Pie chart like above.
chart3 = workbook.add_chart({'type': 'pie'})

# Configure the series.
chart3.add_series({
    'name': 'Pie sales data',
```

```
    'categories': '=Sheet1!A2:A4',
    'values':     '=Sheet1!B2:B4',
})

# Add a title.
chart3.set_title({'name': 'Pie Chart with segment rotation'})

# Change the angle/rotation of the first segment.
chart3.set_rotation(90)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('C34', chart3, {'x_offset': 25, 'y_offset': 10})

workbook.close()
```

30.7 Example: Doughnut Chart

Example of creating Excel Doughnut charts. Chart 1 in the following example is:

Chart 4 shows how to set segment colours and other options.

It is possible to define chart colours for most types of XlsxWriter charts via the `add_series()` method. However, Pie/Doughnut charts are a special case since each segment is represented as a point and as such it is necessary to assign formatting to each point in the series.

```
#######################################################################
#
# An example of creating Excel Doughnut charts with Python and XlsxWriter.
#
# The demo also shows how to set segment colours. It is possible to
# define chart colors for most types of XlsxWriter charts
# via the add_series() method. However, Pie/Doughnut charts are a special
# case since each segment is represented as a point so it is necessary to
# assign formatting to each point in the series.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_doughnut.xlsx')

worksheet = workbook.add_worksheet()
bold = workbook.add_format({'bold': 1})

# Add the worksheet data that the charts will refer to.
headings = ['Category', 'Values']
data = [
    ['Glazed', 'Chocolate', 'Cream'],
    [50, 35, 15],
]

worksheet.write_row('A1', headings, bold)
worksheet.write_column('A2', data[0])
worksheet.write_column('B2', data[1])

#######################################################################
#
# Create a new chart object.
#
chart1 = workbook.add_chart({'type': 'doughnut'})

# Configure the series. Note the use of the list syntax to define ranges:
chart1.add_series({
    'name':       'Doughnut sales data',
    'categories': ['Sheet1', 1, 0, 3, 0],
    'values':     ['Sheet1', 1, 1, 3, 1],
})

# Add a title.
chart1.set_title({'name': 'Popular Doughnut Types'})

# Set an Excel chart style. Colors with white outline and shadow.
chart1.set_style(10)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('C2', chart1, {'x_offset': 25, 'y_offset': 10})

#######################################################################
```

```
#
# Create a Doughnut chart with user defined segment colors.
#

# Create an example Doughnut chart like above.
chart2 = workbook.add_chart({'type': 'doughnut'})

# Configure the series and add user defined segment colours.
chart2.add_series({
    'name': 'Doughnut sales data',
    'categories': '=Sheet1!A2:A4',
    'values':     '=Sheet1!B2:B4',
    'points': [
        {'fill': {'color': '#FA58D0'}},
        {'fill': {'color': '#61210B'}},
        {'fill': {'color': '#F5F6CE'}},
    ],
})

# Add a title.
chart2.set_title({'name': 'Doughnut Chart with user defined colors'})

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('C18', chart2, {'x_offset': 25, 'y_offset': 10})

######################################################################
#
# Create a Doughnut chart with rotation of the segments.
#

# Create an example Doughnut chart like above.
chart3 = workbook.add_chart({'type': 'doughnut'})

# Configure the series.
chart3.add_series({
    'name': 'Doughnut sales data',
    'categories': '=Sheet1!A2:A4',
    'values':     '=Sheet1!B2:B4',
})

# Add a title.
chart3.set_title({'name': 'Doughnut Chart with segment rotation'})

# Change the angle/rotation of the first segment.
chart3.set_rotation(90)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('C34', chart3, {'x_offset': 25, 'y_offset': 10})


######################################################################
#
# Create a Doughnut chart with user defined hole size and other options.
```

```
#

# Create an example Doughnut chart like above.
chart4 = workbook.add_chart({'type': 'doughnut'})

# Configure the series.
chart4.add_series({
    'name': 'Doughnut sales data',
    'categories': '=Sheet1!A2:A4',
    'values':     '=Sheet1!B2:B4',
    'points': [
        {'fill': {'color': '#FA58D0'}},
        {'fill': {'color': '#61210B'}},
        {'fill': {'color': '#F5F6CE'}},
    ],
})

# Set a 3D style.
chart4.set_style(26)

# Add a title.
chart4.set_title({'name': 'Doughnut Chart with options applied'})

# Change the angle/rotation of the first segment.
chart4.set_rotation(28)

# Change the hole size.
chart4.set_hole_size(33)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('C50', chart4, {'x_offset': 25, 'y_offset': 10})


workbook.close()
```

30.8 Example: Scatter Chart

Example of creating Excel Scatter charts.

Chart 1 in the following example is a default scatter chart:

Chart 2 is a scatter chart with straight lines and markers:

Chart 3 is a scatter chart with straight lines and no markers:

Chart 4 is a scatter chart with smooth lines and markers:

Chart 5 is a scatter chart with smooth lines and no markers:

```
#######################################################################
#
# An example of creating Excel Scatter charts with Python and XlsxWriter.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_scatter.xlsx')
worksheet = workbook.add_worksheet()
bold = workbook.add_format({'bold': 1})

# Add the worksheet data that the charts will refer to.
headings = ['Number', 'Batch 1', 'Batch 2']
data = [
    [2, 3, 4, 5, 6, 7],
    [10, 40, 50, 20, 10, 50],
    [30, 60, 70, 50, 40, 30],
]

worksheet.write_row('A1', headings, bold)
worksheet.write_column('A2', data[0])
worksheet.write_column('B2', data[1])
worksheet.write_column('C2', data[2])


#######################################################################
#
# Create a new scatter chart.
#
chart1 = workbook.add_chart({'type': 'scatter'})

# Configure the first series.
chart1.add_series({
    'name': '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
```

```
    'values': '=Sheet1!$B$2:$B$7',
})

# Configure second series. Note use of alternative syntax to define ranges.
chart1.add_series({
    'name':       ['Sheet1', 0, 2],
    'categories': ['Sheet1', 1, 0, 6, 0],
    'values':     ['Sheet1', 1, 2, 6, 2],
})

# Add a chart title and some axis labels.
chart1.set_title ({'name': 'Results of sample analysis'})
chart1.set_x_axis({'name': 'Test number'})
chart1.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart1.set_style(11)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D2', chart1, {'x_offset': 25, 'y_offset': 10})

#######################################################################
#
# Create a scatter chart sub-type with straight lines and markers.
#
chart2 = workbook.add_chart({'type': 'scatter',
                             'subtype': 'straight_with_markers'})

# Configure the first series.
chart2.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure second series.
chart2.add_series({
    'name':       '=Sheet1!$C$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$C$2:$C$7',
})

# Add a chart title and some axis labels.
chart2.set_title ({'name': 'Straight line with markers'})
chart2.set_x_axis({'name': 'Test number'})
chart2.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart2.set_style(12)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D18', chart2, {'x_offset': 25, 'y_offset': 10})
```

```
    ######################################################################
    #
    # Create a scatter chart sub-type with straight lines and no markers.
    #
    chart3 = workbook.add_chart({'type': 'scatter',
                                 'subtype': 'straight'})

    # Configure the first series.
    chart3.add_series({
        'name':       '=Sheet1!$B$1',
        'categories': '=Sheet1!$A$2:$A$7',
        'values':     '=Sheet1!$B$2:$B$7',
    })

    # Configure second series.
    chart3.add_series({
        'name':       '=Sheet1!$C$1',
        'categories': '=Sheet1!$A$2:$A$7',
        'values':     '=Sheet1!$C$2:$C$7',
    })

    # Add a chart title and some axis labels.
    chart3.set_title ({'name': 'Straight line'})
    chart3.set_x_axis({'name': 'Test number'})
    chart3.set_y_axis({'name': 'Sample length (mm)'})

    # Set an Excel chart style.
    chart3.set_style(13)

    # Insert the chart into the worksheet (with an offset).
    worksheet.insert_chart('D34', chart3, {'x_offset': 25, 'y_offset': 10})

    ######################################################################
    #
    # Create a scatter chart sub-type with smooth lines and markers.
    #
    chart4 = workbook.add_chart({'type': 'scatter',
                                 'subtype': 'smooth_with_markers'})

    # Configure the first series.
    chart4.add_series({
        'name':       '=Sheet1!$B$1',
        'categories': '=Sheet1!$A$2:$A$7',
        'values':     '=Sheet1!$B$2:$B$7',
    })

    # Configure second series.
    chart4.add_series({
        'name':       '=Sheet1!$C$1',
        'categories': '=Sheet1!$A$2:$A$7',
        'values':     '=Sheet1!$C$2:$C$7',
    })
```

```
# Add a chart title and some axis labels.
chart4.set_title ({'name': 'Smooth line with markers'})
chart4.set_x_axis({'name': 'Test number'})
chart4.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart4.set_style(14)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D51', chart4, {'x_offset': 25, 'y_offset': 10})

#######################################################################
#
# Create a scatter chart sub-type with smooth lines and no markers.
#
chart5 = workbook.add_chart({'type': 'scatter',
                             'subtype': 'smooth'})

# Configure the first series.
chart5.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure second series.
chart5.add_series({
    'name':       '=Sheet1!$C$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$C$2:$C$7',
})

# Add a chart title and some axis labels.
chart5.set_title ({'name': 'Smooth line'})
chart5.set_x_axis({'name': 'Test number'})
chart5.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart5.set_style(15)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D66', chart5, {'x_offset': 25, 'y_offset': 10})

workbook.close()
```

30.9 Example: Radar Chart

Example of creating Excel Column charts.

Chart 1 in the following example is a default radar chart:

Chart 2 in the following example is a radar chart with markers:

Chart 3 in the following example is a filled radar chart:

```
#######################################################################
#
# An example of creating Excel Radar charts with Python and XlsxWriter.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_radar.xlsx')
worksheet = workbook.add_worksheet()
bold = workbook.add_format({'bold': 1})

# Add the worksheet data that the charts will refer to.
headings = ['Number', 'Batch 1', 'Batch 2']
data = [
    [2, 3, 4, 5, 6, 7],
    [30, 60, 70, 50, 40, 30],
    [25, 40, 50, 30, 50, 40],
]

worksheet.write_row('A1', headings, bold)
worksheet.write_column('A2', data[0])
worksheet.write_column('B2', data[1])
worksheet.write_column('C2', data[2])

#######################################################################
#
# Create a new radar chart.
#
chart1 = workbook.add_chart({'type': 'radar'})

# Configure the first series.
chart1.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
```

```
})

# Configure second series. Note use of alternative syntax to define ranges.
chart1.add_series({
    'name':       ['Sheet1', 0, 2],
    'categories': ['Sheet1', 1, 0, 6, 0],
    'values':     ['Sheet1', 1, 2, 6, 2],
})

# Add a chart title and some axis labels.
chart1.set_title ({'name': 'Results of sample analysis'})
chart1.set_x_axis({'name': 'Test number'})
chart1.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart1.set_style(11)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D2', chart1, {'x_offset': 25, 'y_offset': 10})

#######################################################################
#
# Create a radar chart with markers chart sub-type.
#
chart2 = workbook.add_chart({'type': 'radar', 'subtype': 'with_markers'})

# Configure the first series.
chart2.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure second series.
chart2.add_series({
    'name':       '=Sheet1!$C$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$C$2:$C$7',
})

# Add a chart title and some axis labels.
chart2.set_title ({'name': 'Radar Chart With Markers'})
chart2.set_x_axis({'name': 'Test number'})
chart2.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart2.set_style(12)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D18', chart2, {'x_offset': 25, 'y_offset': 10})

#######################################################################
#
```

```
# Create a filled radar chart sub-type.
#
chart3 = workbook.add_chart({'type': 'radar', 'subtype': 'filled'})

# Configure the first series.
chart3.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure second series.
chart3.add_series({
    'name':       '=Sheet1!$C$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$C$2:$C$7',
})

# Add a chart title and some axis labels.
chart3.set_title ({'name': 'Filled Radar Chart'})
chart3.set_x_axis({'name': 'Test number'})
chart3.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart3.set_style(13)

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D34', chart3, {'x_offset': 25, 'y_offset': 10})

workbook.close()
```

30.10 Example: Stock Chart

Example of creating and Excel HiLow-Close Stock chart.

Chart 1 in the following example is:

```
#######################################################################
#
# An example of creating Excel Stock charts with Python and XlsxWriter.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
from datetime import datetime
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_stock.xlsx')
worksheet = workbook.add_worksheet()

bold = workbook.add_format({'bold': 1})
date_format = workbook.add_format({'num_format': 'dd/mm/yyyy'})

chart = workbook.add_chart({'type': 'stock'})

# Add the worksheet data that the charts will refer to.
headings = ['Date', 'High', 'Low', 'Close']
data = [
    ['2007-01-01', '2007-01-02', '2007-01-03', '2007-01-04', '2007-01-05'],
    [27.2, 25.03, 19.05, 20.34, 18.5],
    [23.49, 19.55, 15.12, 17.84, 16.34],
    [25.45, 23.05, 17.32, 20.45, 17.34],
]

worksheet.write_row('A1', headings, bold)

for row in range(5):
    date = datetime.strptime(data[0][row], "%Y-%m-%d")

    worksheet.write(row + 1, 0, date, date_format)
    worksheet.write(row + 1, 1, data[1][row])
    worksheet.write(row + 1, 2, data[2][row])
    worksheet.write(row + 1, 3, data[3][row])
```

```
worksheet.set_column('A:D', 11)

# Add a series for each of the High-Low-Close columns.
chart.add_series({
    'categories': '=Sheet1!$A$2:$A$6',
    'values': '=Sheet1!$B$2:$B$6',
})

chart.add_series({
    'categories': '=Sheet1!$A$2:$A$6',
    'values':     '=Sheet1!$C$2:$C$6',
})

chart.add_series({
    'categories': '=Sheet1!$A$2:$A$6',
    'values': '=Sheet1!$D$2:$D$6',
})

# Add a chart title and some axis labels.
chart.set_title ({'name': 'High-Low-Close'})
chart.set_x_axis({'name': 'Date'})
chart.set_y_axis({'name': 'Share price'})

worksheet.insert_chart('E9', chart)

workbook.close()
```

30.11 Example: Secondary Axis Chart

Example of creating an Excel Line chart with a secondary axis. Note, the primary and secondary chart type must be the same.

```
#######################################################################
#
# An example of creating an Excel Line chart with a secondary axis
# using Python and XlsxWriter.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_secondary_axis.xlsx')
worksheet = workbook.add_worksheet()
bold = workbook.add_format({'bold': 1})

# Add the worksheet data that the charts will refer to.
headings = ['Aliens', 'Humans']
data = [
    [2, 3, 4, 5, 6, 7],
    [10, 40, 50, 20, 10, 50],
]

worksheet.write_row('A1', headings, bold)
worksheet.write_column('A2', data[0])
worksheet.write_column('B2', data[1])


# Create a new chart object. In this case an embedded chart.
chart = workbook.add_chart({'type': 'line'})

# Configure a series with a secondary axis
chart.add_series({
    'name':   '=Sheet1!$A$1',
    'values': '=Sheet1!$A$2:$A$7',
    'y2_axis': 1,
})

chart.add_series({
    'name':   '=Sheet1!$B$1',
    'values': '=Sheet1!$B$2:$B$7',
})

chart.set_legend({'position': 'right'})

# Add a chart title and some axis labels.
chart.set_title({'name': 'Survey results'})
chart.set_x_axis({'name': 'Days', })
chart.set_y_axis({'name': 'Population', 'major_gridlines': {'visible': 0}})
chart.set_y2_axis({'name': 'Laser wounds'})

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D2', chart, {'x_offset': 25, 'y_offset': 10})

workbook.close()
```

30.12 Example: Date Axis Chart

Date Category Axes are a special case of Category axes in Excel which give them some of the properties of Values axes.

For example, Excel doesn't normally allow minimum and maximum values to be set for category axes. However, date axes are an exception.

In XlsxWriter Date Category Axes are set using the `date_axis` option in `set_x_axis()` or `set_y_axis()`:

```
chart.set_x_axis({'date_axis': True})
```

If used, the `min` and `max` values should be set as Excel times or dates.

```
######################################################################
#
# An example of creating an Excel charts with a date axis using
# Python and XlsxWriter.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#

from datetime import date
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_date_axis.xlsx')

worksheet = workbook.add_worksheet()
chart = workbook.add_chart({'type': 'line'})
date_format = workbook.add_format({'num_format': 'dd/mm/yyyy'})

# Widen the first column to display the dates.
worksheet.set_column('A:A', 12)

# Some data to be plotted in the worksheet.
dates = [date(2013, 1, 1),
         date(2013, 1, 2),
         date(2013, 1, 3),
         date(2013, 1, 4),
```

```
            date(2013, 1, 5),
            date(2013, 1, 6),
            date(2013, 1, 7),
            date(2013, 1, 8),
            date(2013, 1, 9),
            date(2013, 1, 10)]

values = [10, 30, 20, 40, 20, 60, 50, 40, 30, 30]

# Write the date to the worksheet.
worksheet.write_column('A1', dates, date_format)
worksheet.write_column('B1', values)

# Add a series to the chart.
chart.add_series({
    'categories': '=Sheet1!$A$1:$A$10',
    'values': '=Sheet1!$B$1:$B$10',
})

# Configure the X axis as a Date axis and set the max and min limits.
chart.set_x_axis({
    'date_axis': True,
    'min': date(2013, 1, 2),
    'max': date(2013, 1, 9),
})

# Turn off the legend.
chart.set_legend({'none': True})

# Insert the chart into the worksheet.
worksheet.insert_chart('D2', chart)

workbook.close()
```

30.13 Example: Charts with Data Tables

Example of creating charts with data tables.

Chart 1 in the following example is a column chart with default data table:

Chart 2 is a column chart with default data table with legend keys:

```
#######################################################################
#
# An example of creating Excel Column charts with data tables using
# Python and XlsxWriter.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_data_table.xlsx')
worksheet = workbook.add_worksheet()
bold = workbook.add_format({'bold': 1})

# Add the worksheet data that the charts will refer to.
headings = ['Number', 'Batch 1', 'Batch 2']
data = [
    [2, 3, 4, 5, 6, 7],
```

```
    [10, 40, 50, 20, 10, 50],
    [30, 60, 70, 50, 40, 30],
]

worksheet.write_row('A1', headings, bold)
worksheet.write_column('A2', data[0])
worksheet.write_column('B2', data[1])
worksheet.write_column('C2', data[2])


######################################################################
#
# Create a column chart with a data table.
#
chart1 = workbook.add_chart({'type': 'column'})

# Configure the first series.
chart1.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure second series. Note use of alternative syntax to define ranges.
chart1.add_series({
    'name':       ['Sheet1', 0, 2],
    'categories': ['Sheet1', 1, 0, 6, 0],
    'values':     ['Sheet1', 1, 2, 6, 2],
})

# Add a chart title and some axis labels.
chart1.set_title({'name': 'Chart with Data Table'})
chart1.set_x_axis({'name': 'Test number'})
chart1.set_y_axis({'name': 'Sample length (mm)'})

# Set a default data table on the X-Axis.
chart1.set_table()

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D2', chart1, {'x_offset': 25, 'y_offset': 10})

######################################################################
#
# Create a column chart with a data table and legend keys.
#
chart2 = workbook.add_chart({'type': 'column'})

# Configure the first series.
chart2.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})
```

```
# Configure second series.
chart2.add_series({
    'name':       '=Sheet1!$C$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$C$2:$C$7',
})

# Add a chart title and some axis labels.
chart2.set_title({'name': 'Data Table with legend keys'})
chart2.set_x_axis({'name': 'Test number'})
chart2.set_y_axis({'name': 'Sample length (mm)'})

# Set a data table on the X-Axis with the legend keys shown.
chart2.set_table({'show_keys': True})

# Hide the chart legend since the keys are shown on the data table.
chart2.set_legend({'position': 'none'})

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D18', chart2, {'x_offset': 25, 'y_offset': 10})

workbook.close()
```

30.14 Example: Charts with Data Tools

A demo of an various Excel chart data tools that are available via an XlsxWriter chart. These include, Trendlines, Data Labels, Error Bars, Drop Lines, High-Low Lines and Up-Down Bars.

Chart 1 in the following example is a chart with trendlines:

Chart 2 is a chart with data labels and markers:

Chart 3 is a chart with error bars:

Chart 4 is a chart with up-down bars:

Chart 5 is a chart with hi-low lines:

Chart 6 is a chart with drop lines:

```
#######################################################################
#
# A demo of an various Excel chart data tools that are available via
# an XlsxWriter chart.
#
# These include, Trendlines, Data Labels, Error Bars, Drop Lines,
# High-Low Lines and Up-Down Bars.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('chart_data_tools.xlsx')
worksheet = workbook.add_worksheet()
bold = workbook.add_format({'bold': 1})

# Add the worksheet data that the charts will refer to.
headings = ['Number', 'Data 1', 'Data 2']
data = [
    [2, 3, 4, 5, 6, 7],
    [10, 40, 50, 20, 10, 50],
    [30, 60, 70, 50, 40, 30],
]

worksheet.write_row('A1', headings, bold)
worksheet.write_column('A2', data[0])
worksheet.write_column('B2', data[1])
worksheet.write_column('C2', data[2])


#######################################################################
#
# Trendline example.
#
# Create a Line chart.
chart1 = workbook.add_chart({'type': 'line'})
```

```
# Configure the first series with a polynomial trendline.
chart1.add_series({
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
    'trendline': {
        'type': 'polynomial',
        'order': 3,
    },
})

# Configure the second series with a moving average trendline.
chart1.add_series({
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$C$2:$C$7',
    'trendline': {'type': 'linear'},
})

# Add a chart title. and some axis labels.
chart1.set_title({'name': 'Chart with Trendlines'})

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D2', chart1, {'x_offset': 25, 'y_offset': 10})

######################################################################
#
# Data Labels and Markers example.
#
# Create a Line chart.
chart2 = workbook.add_chart({'type': 'line'})

# Configure the first series.
chart2.add_series({
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
    'data_labels': {'value': 1},
    'marker': {'type': 'automatic'},
})

# Configure the second series.
chart2.add_series({
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$C$2:$C$7',
})

# Add a chart title. and some axis labels.
chart2.set_title({'name': 'Chart with Data Labels and Markers'})

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D18', chart2, {'x_offset': 25, 'y_offset': 10})

######################################################################
#
# Error Bars example.
```

```
#
# Create a Line chart.
chart3 = workbook.add_chart({'type': 'line'})

# Configure the first series.
chart3.add_series({
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
    'y_error_bars': {'type': 'standard_error'},
})

# Configure the second series.
chart3.add_series({
    'categories': '=Sheet1!$A$2:$A$7',
    'values': '=Sheet1!$C$2:$C$7',
})

# Add a chart title. and some axis labels.
chart3.set_title({'name': 'Chart with Error Bars'})

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D34', chart3, {'x_offset': 25, 'y_offset': 10})

######################################################################
#
# Up-Down Bars example.
#
# Create a Line chart.
chart4 = workbook.add_chart({'type': 'line'})

# Add the Up-Down Bars.
chart4.set_up_down_bars()

# Configure the first series.
chart4.add_series({
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure the second series.
chart4.add_series({
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$C$2:$C$7',
})

# Add a chart title. and some axis labels.
chart4.set_title({'name': 'Chart with Up-Down Bars'})

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D50', chart4, {'x_offset': 25, 'y_offset': 10})

######################################################################
#
```

```
# High-Low Lines example.
#
# Create a Line chart.
chart5 = workbook.add_chart({'type': 'line'})

# Add the High-Low lines.
chart5.set_high_low_lines()

# Configure the first series.
chart5.add_series({
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure the second series.
chart5.add_series({
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$C$2:$C$7',
})

# Add a chart title. and some axis labels.
chart5.set_title({'name': 'Chart with High-Low Lines'})

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D66', chart5, {'x_offset': 25, 'y_offset': 10})

######################################################################
#
# Drop Lines example.
#
# Create a Line chart.
chart6 = workbook.add_chart({'type': 'line'})

# Add Drop Lines.
chart6.set_drop_lines()

# Configure the first series.
chart6.add_series({
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure the second series.
chart6.add_series({
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$C$2:$C$7',
})

# Add a chart title. and some axis labels.
chart6.set_title({'name': 'Chart with Drop Lines'})

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart('D82', chart6, {'x_offset': 25, 'y_offset': 10})
```

```
workbook.close()
```

30.15 Example: Chartsheet

Example of creating an Excel Bar chart on a *chartsheet*.

```
#######################################################################
#
# An example of creating an Excel chart in a chartsheet with Python
# and XlsxWriter.
#
# Copyright 2013-2015, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook('chartsheet.xlsx')

# Add a worksheet to hold the data.
worksheet = workbook.add_worksheet()

# Add a chartsheet. A worksheet that only holds a chart.
chartsheet = workbook.add_chartsheet()

# Add a format for the headings.
bold = workbook.add_format({'bold': 1})

# Add the worksheet data that the charts will refer to.
```

```
headings = ['Number', 'Batch 1', 'Batch 2']
data = [
    [2, 3, 4, 5, 6, 7],
    [10, 40, 50, 20, 10, 50],
    [30, 60, 70, 50, 40, 30],
]

worksheet.write_row('A1', headings, bold)
worksheet.write_column('A2', data[0])
worksheet.write_column('B2', data[1])
worksheet.write_column('C2', data[2])


# Create a new bar chart.
chart1 = workbook.add_chart({'type': 'bar'})

# Configure the first series.
chart1.add_series({
    'name':       '=Sheet1!$B$1',
    'categories': '=Sheet1!$A$2:$A$7',
    'values':     '=Sheet1!$B$2:$B$7',
})

# Configure a second series. Note use of alternative syntax to define ranges.
chart1.add_series({
    'name':       ['Sheet1', 0, 2],
    'categories': ['Sheet1', 1, 0, 6, 0],
    'values':     ['Sheet1', 1, 2, 6, 2],
})

# Add a chart title and some axis labels.
chart1.set_title ({'name': 'Results of sample analysis'})
chart1.set_x_axis({'name': 'Test number'})
chart1.set_y_axis({'name': 'Sample length (mm)'})

# Set an Excel chart style.
chart1.set_style(11)

# Add the chart to the chartsheet.
chartsheet.set_chart(chart1)

workbook.close()
```

CHAPTER

THIRTYONE

ALTERNATIVE MODULES FOR HANDLING EXCEL FILES

The following are some Python alternatives to XlsxWriter.

31.1 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.

31.2 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.

31.3 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.

CHAPTER

THIRTYTWO

KNOWN ISSUES AND BUGS

This section lists known issues and bugs and gives some information on how to submit bug reports.

32.1 "Content is Unreadable. Open and Repair"

Very, very occasionally you may see an Excel warning when opening an XlsxWriter file like:

> Excel could not open file.xlsx because some content is unreadable. Do you want to open and repair this workbook.

This ominous sounding message is Excel's default warning for any validation error in the XML used for the components of the XLSX file.

If you encounter an issue like this you should open an issue on GitHub with a program to replicate the issue (see below) or send one of the failing output files to the *Author*.

32.2 "Exception caught in workbook destructor. Explicit close() may be required"

The following exception, or similar, can occur if the `close()` method isn't used at the end of the program:

```
Exception Exception: Exception('Exception caught in workbook destructor.
Explicit close() may be required for workbook.',)
in <bound method Workbook.__del__ of <xlsxwriter.workbook.Workbookobject
at 0x103297d50>>
```

Note, it is possible that this exception will also be raised as part of another exception that occurs during workbook destruction. In either case ensure that there is an explicit `workbook.close()` in the program.

32.3 Formulas displayed as #NAME? until edited

Excel 2010 and 2013 added functions which weren't defined in the original file specification. These functions are referred to as *future* functions. Examples of these functions are `ACOT`,

`CHISQ.DIST.RT` , `CONFIDENCE.NORM`, `STDEV.P`, `STDEV.S` and `WORKDAY.INTL`. The full list is given in the MS XLSX extensions documentation on future functions.

When written using `write_formula()` these functions need to be fully qualified with the `_xlfn.` prefix as they are shown in the MS XLSX documentation link above. For example:

```
worksheet.write_formula('A1', '=_xlfn.STDEV.S(B1:B10)')
```

32.4 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)
```

32.5 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.

32.6 Images not displayed correctly in Excel 2001 for Mac and non-Excel applications

Images inserted into worksheets via `insert_image()` may not display correctly in Excel 2011 for Mac and non-Excel applications such as OpenOffice and LibreOffice. Specifically the images may looked stretched or squashed.

This is not specifically an XlsxWriter issue. It also occurs with files created in Excel 2007 and Excel 2010.

CHAPTER

THIRTYTHREE

REPORTING BUGS

Here are some tips on reporting bugs in XlsxWriter.

33.1 Upgrade to the latest version of the module

The bug you are reporting may already be fixed in the latest version of the module. You can check which version of XlsxWriter that you are using as follows:

```
python -c 'import xlsxwriter; print(xlsxwriter.__version__)'
```

Check the *Changes in XlsxWriter* section to see what has changed in the latest versions.

33.2 Read the documentation

Read or search the XlsxWriter documentation to see if the issue you are encountering is already explained.

33.3 Look at the example programs

There are many *Examples* in the distribution. Try to identify an example program that corresponds to your query and adapt it to use as a bug report.

33.4 Use the official XlsxWriter Issue tracker on GitHub

The official XlsxWriter Issue tracker is on GitHub.

33.5 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. It is generally 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 paste large non-relevant sections of your program.

A sample bug report is shown below. This format helps to analyse and respond to the bug report more quickly.

> **Issue with SOMETHING**
>
> I am using XlsxWriter to do SOMETHING but it 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:
>
> ```
> import xlsxwriter
>
> workbook = xlsxwriter.Workbook('hello.xlsx')
> worksheet = workbook.add_worksheet()
>
> worksheet.write('A1', 'Hello world')
>
> workbook.close()
> ```

See also how How to create a Minimal, Complete, and Verifiable example from StackOverflow.

CHAPTER

THIRTYFOUR

FREQUENTLY ASKED QUESTIONS

The section outlines some answers to frequently asked questions.

34.1 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.

34.2 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)
```

34.3 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.

34.4 Q. Is feature X supported or will it be supported?

All supported features are documented.

34.5 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 your write it.

34.6 Q. Do people actually ask these questions frequently, or at all?

Apart from this question, yes.

CHAPTER

THIRTYFIVE

CHANGES IN XLSXWRITER

This section shows changes and bug fixes in the XlsxWriter module.

35.1 Release 0.6.7 - March 1 2015

- Added option to add function value in worksheet `add_table()`. Issue #216.
- Fix for A1 row/col numbers below lower bound. Issue #212.

35.2 Release 0.6.6 - January 16 2015

- Fix for incorrect shebang line in *vba_extract.py* packaged in wheel. Issue #211.
- Added docs and example for diagonal cell border. See *Example: Diagonal borders in cells*.

35.3 Release 0.6.5 - December 31 2014

- Added worksheet quoting for chart names in lists. Issue #205.
- Added docs on how to find and set VBA codenames. Issue #202.
- Fix Python3 issue with unused charts. Issue #200.
- Enabled warning for missing category is scatter chart. Issue #197.
- Fix for upper chart style limit. Increased the chart style limit from 42 to the correct 48. Issue #192.
- Raise warning if a chart is inserted more than once. Issue #184.

35.4 Release 0.6.4 - November 15 2014

- Fix for issue where fonts applied to data labels raised exception. Issue #179.
- Added option to allow explicit text axis types for charts, similar to date axes. Issue #178.

- Fix for issue where the bar/column chart gap and overlap weren't applied to the secondary axis. Issue #177.

35.5 Release 0.6.3 - November 6 2014

- Added support for adding VBA macros to workbooks. See *Working with VBA Macros*. Issue #126.

35.6 Release 0.6.2 - November 1 2014

- Added chart axis line and fill properties. Issue #88.

35.7 Release 0.6.1 - October 29 2014

- Added chart specific handling of data label positions since not all positions are available for all chart types. Issue #170.
- Added number formatting (issue #130), font handling, separator and legend key for data labels. See *Chart series option: Data Labels*
- Fix for non-quoted worksheet names containing spaces and non-alphanumeric characters. Issue #167.

35.8 Release 0.6.0 - October 15 2014

- Added option to add images to headers and footers. See *Example: Adding Headers and Footers to Worksheets*. Issue #133.
- Fixed issue where non 96dpi images weren't scaled properly in Excel. Issue #164.
- Added option to not scale header/footer with page. See `set_header()`. Issue #134.

35.9 Release 0.5.9 - October 11 2014

- Removed `egg_base` requirement from `setup.cfg` which was preventing installation on Windows. Issue #162.
- Fix for issue where X axis title formula was overwritten by the Y axis title. Issue #161.

35.10 Release 0.5.8 - September 28 2014

- Added support for Doughnut charts. Feature request #157.
- Added support for wheel packages. Feature request #156.
- Made the exception handling in `write()` clearer for unsupported types so that it raises a more accurate TypeError instead of a ValueError. Issue #153.

35.11 Release 0.5.7 - August 13 2014

- Added support for `insert_image()` images from byte streams to allow images from URLs and other sources. Feature request #118.
- Added `write_datetime()` support for datetime.timedelta. Feature request #128.

35.12 Release 0.5.6 - July 22 2014

- Fix for spurious exception message when `close()` isn't used. Issue #131.
- Fix for formula string values that look like numbers. Issue #122.
- Clarify `print_area()` documentation for complete row/column ranges. Issue #139.
- Fix for unicode strings in data validation lists. Issue #135.

35.13 Release 0.5.5 - May 6 2014

- Fix for incorrect chart offsets in `insert_chart()` and `set_size()`.

35.14 Release 0.5.4 - May 4 2014

- Added image positioning option to `insert_image()` to control how images are moved in relation to surrounding cells. Issue #117.
- Fix for chart `error_bar` exceptions. Issue #115.
- Added clearer reporting of nested exceptions in `write()` methods. Issue #108.
- Added support for `inside_base` data label position in charts.

35.15 Release 0.5.3 - February 20 2014

- Added checks and warnings for data validation limits. Issue #89.

- Added option to add hyperlinks to images. Thanks to Paul Tax.
- Added Python 3 Http server example. Thanks to Krystian Rosinski.
- Added `set_calc_mode()` method to control automatic calculation of formulas when worksheet is opened. Thanks to Chris Tompkinson.
- Added `use_zip64()` method to allow ZIP64 extensions when writing very large files.
- Fix to handle '0' and other number like strings as number formats. Issue #103.
- Fix for missing images in `in_memory` mode. Issue #102.

35.16 Release 0.5.2 - December 31 2013

- Added date axis handling to charts. See *Example: Date Axis Chart*. Issue #73.
- Added support for non-contiguous chart ranges. Issue #44.
- Fix for low byte and control characters in strings. Issue #86.
- Fix for chart titles with exclamation mark. Issue #83.
- Fix to remove duplicate `set_column()` entries. Issue #82.

35.17 Release 0.5.1 - December 2 2013

- Added interval unit option for category axes. Issue #69.
- Fix for axis name font rotation.
- Fix for several minor issues with Pie chart legends.

35.18 Release 0.5.0 - November 17 2013

- Added *Chartsheets* to allow single worksheet charts. Issue #10.

35.19 Release 0.4.9 - November 17 2013

- Added *chart object positioning and sizing* to allow positioning of plotarea, legend, title and axis names. Issue #66.
- Added `set_title()` `none` option to turn off automatic titles.
- Improved `define_name()` name validation.
- Fix to prevent modification of user parameters in `condtional_format()`.

35.20 Release 0.4.8 - November 13 2013

- Added `in_memory` `Workbook()` constructor option to allow XlsxWriter to work on Google App Engine. Issue #28.

35.21 Release 0.4.7 - November 9 2013

- Added fix for markers on non-marker scatter charts. Issue #62.
- Added custom error bar option. Thanks to input from Alex Birmingham.
- Changed Html docs to Bootstap theme.
- Added *Example: Merging Cells with a Rich String*.

35.22 Release 0.4.6 - October 23 2013

- Added font formatting to chart legends.

35.23 Release 0.4.5 - October 21 2013

- Added `position_axis` chart axis option.
- Added optional list handling for chart names.

35.24 Release 0.4.4 - October 16 2013

- Documented use of *cell utility* functions.
- Fix for tables added in non-sequential order. Closes #51 reported by calfzhou.

35.25 Release 0.4.3 - September 12 2013

- Fix for comments overlying columns with non-default width. Issue #45.

35.26 Release 0.4.2 - August 30 2013

- Added a default blue underline hyperlink format for `write_url()`.
- Added `Workbook()` constructor options `strings_to_formulas` and `strings_to_urls` to override default conversion of strings in write().

35.27 Release 0.4.1 - August 28 2013

- Fix for charts and images that cross rows and columns that are hidden or formatted but which don't have size changes. Issue #42 reported by Kristian Stobbe.

35.28 Release 0.4.0 - August 26 2013

- Added more generic support for JPEG files. Issue #40 reported by Simon Breuss.
- Fix for harmless Python 3 installation warning. Issue #41 reported by James Reeves.

35.29 Release 0.3.9 - August 24 2013

- Added fix for minor issue with `insert_image()` for images that extend over several cells.
- Added fix to ensure formula calculation on load regardless of Excel version.

35.30 Release 0.3.8 - August 23 2013

- Added handling for Decimal(), Fraction() and other float types to the `write()` function.
- Added Python 2.5 and Jython support. Thanks to Jonas Diemer for the patch.

35.31 Release 0.3.7 - August 16 2013

- Added `write_boolean()` function to write Excel boolean values. Issue #37. Also added explicit handling of Python bool values to the `write()` function.
- Changed `Workbook()` constructor option `strings_to_numbers` default option to False so that there is no implicit conversion of numbers in strings to numbers. The previous behaviour can be obtained by setting the constructor option to True. **Note** This is a backward incompatibility.

35.32 Release 0.3.6 - July 26 2013

- Simplified import based on a suggestion from John Yeung. Issue #26.
- Fix for NAN/INF converted to invalid numbers in write(). Issue #30.
- Added `Workbook()` constructor option `strings_to_numbers` to override default conversion of number strings to numbers in write().
- Added `Workbook()` constructor option `default_date_format` to allow a default date format string to be set. Issue #5.

35.33 Release 0.3.5 - June 28 2013

- Reverted back to using codecs for file encoding (versions <= 0.3.1) to avoid numerous UTF-8 issues in Python2/3.

35.34 Release 0.3.4 - June 27 2013

- Added Chart line smoothing option. Thanks to Dieter Vandenbussche.
- Added Http Server example (*Example: Simple HTTP Server*). Thanks to Alexander Afanasiev.
- Fixed inaccurate column width calculation. Closes #27. Thanks to John Yeung.
- Added chart axis font rotation.

35.35 Release 0.3.3 - June 10 2013

- Minor packaging fixes #14 and #19.
- Fixed explicit UTF-8 file encoding for Python 3. PR from Alexandr Shadchin, #15.
- Fixed invalid string formatting resulted in misleading stacktrace. PR from Andrei Korostelev, #21.

35.36 Release 0.3.2 - May 1 2013

- Speed optimisations. The module is now 10-15% faster on average.

35.37 Release 0.3.1 - April 27 2013

- Added chart support. See the *The Chart Class*, *Working with Charts* and *Chart Examples*.

35.38 Release 0.3.0 - April 7 2013

- Added worksheet sparklines. See *Working with Sparklines*, *Example: Sparklines (Simple)* and *Example: Sparklines (Advanced)*

35.39 Release 0.2.9 - April 7 2013

- Added worksheet tables. See *Working with Worksheet Tables* and *Example: Worksheet Tables*.

- Tested with the new Python stable releases 2.7.4 and 3.3.1. All tests now pass in the following versions:
  - Python 2.6
  - Python 2.7.2
  - Python 2.7.3
  - Python 2.7.4
  - Python 3.1
  - Python 3.2
  - Python 3.3.0
  - Python 3.3.1
- There are now over 700 unit tests including more than 170 tests that compare against the output of Excel.

35.40 Release 0.2.8 - April 4 2013

- Added worksheet outlines and grouping. See *Working with Outlines and Grouping*.

35.41 Release 0.2.7 - April 3 2013

- Added `set_default_row()` method. See *Example: Hiding Rows and Columns*.
- Added hide_row_col.py, hide_sheet.py and text_indent.py examples.

35.42 Release 0.2.6 - April 1 2013

- Added `freeze_panes()` and `split_panes()` methods. See *Example: Freeze Panes and Split Panes* .
- Added `set_selection()` method to select worksheet cell or range of cells.

35.43 Release 0.2.5 - April 1 2013

- Added additional `Workbook()` parameters `'tmpdir'` and `'date_1904'`.

35.44 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.

35.45 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.

35.46 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.

35.47 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.

35.48 Release 0.2.0 - March 24 2013

- Added conditional formatting. See the `conditional_format()` method, *Working with Conditional Formatting* and *Example: Conditional Formatting*.

35.49 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

35.50 Release 0.1.8 - March 18 2013

- Fixed Python 3 support.

35.51 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*.

35.52 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.

35.53 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.

35.54 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*.

35.55 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.

35.56 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.

35.57 Release 0.1.1 - March 3 2013

- Added the `write_url()` worksheet method for writing hyperlinks to a worksheet.

35.58 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

35.59 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.

35.60 Release 0.0.8 - February 26 2013

- Added the `merge_range()` method to merge worksheet cells.

35.61 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()`

35.62 Release 0.0.6 - February 22 2013

- Added page setup method.
  - `print_row_col_headers()`

35.63 Release 0.0.5 - February 21 2013

- Added page setup methods.
  - `repeat_rows()`
  - `repeat_columns()`

35.64 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()`

35.65 Release 0.0.3 - February 19 2013

- Added page setup method.
  - `set_margins()`

35.66 Release 0.0.2 - February 18 2013

- Added page setup methods.
  - `set_landscape()`
  - `set_portrait()`
  - `set_page_view()`
  - `set_paper()`
  - `print_across()`

35.67 Release 0.0.1 - February 17 2013

- First public release.

CHAPTER

THIRTYSIX

AUTHOR

XlsxWriter was written by John McNamara.

- GitHub
- Twitter @jmcnamara13
- Coderwall
- Ohloh

You can contact me at jmcnamara@cpan.org.

36.1 Sponsorship

If your company would like to sponsor a feature then let me know.

36.2 Donations

If you would like to donate to the XlsxWriter project to keep it active you can do so via PayPal.

CHAPTER

THIRTYSEVEN

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.

A

activate() (built-in function), 64
add_chart() (built-in function), 25
add_chartsheet() (built-in function), 26
add_format() (built-in function), 24
add_series() (built-in function), 113
add_sparkline() (built-in function), 60
add_table() (built-in function), 60
add_vba_project() (built-in function), 30
add_worksheet() (built-in function), 24
autofilter() (built-in function), 69

C

center_horizontally() (built-in function), 81
center_vertically() (built-in function), 81
close() (built-in function), 27
conditional_format() (built-in function), 58

D

data_validation() (built-in function), 57
define_name() (built-in function), 29

F

filter_column() (built-in function), 70
filter_column_list() (built-in function), 71
fit_to_pages() (built-in function), 89
freeze_panes() (built-in function), 72

G

get_name() (built-in function), 64

H

hide() (built-in function), 65
hide_gridlines() (built-in function), 87
hide_zero() (built-in function), 74

I

insert_button() (built-in function), 55
insert_chart() (built-in function), 54
insert_image() (built-in function), 51

M

merge_range() (built-in function), 67

O

outline_settings() (built-in function), 76

P

print_across() (built-in function), 89
print_area() (built-in function), 88
print_row_col_headers() (built-in function), 88
protect() (built-in function), 75

R

repeat_columns() (built-in function), 87
repeat_rows() (built-in function), 87
right_to_left() (built-in function), 74

S

select() (built-in function), 65
set_align() (built-in function), 99
set_bg_color() (built-in function), 103
set_bold() (built-in function), 95
set_border() (built-in function), 104
set_border_color() (built-in function), 106
set_bottom() (built-in function), 105
set_bottom_color() (built-in function), 106
set_calc_mode() (built-in function), 31
set_center_across() (built-in function), 100
set_chart() (built-in function), 136
set_chartarea() (built-in function), 126
set_column() (built-in function), 49
set_comments_author() (built-in function), 63
set_default_row() (built-in function), 76
set_diag_border() (built-in function), 107
set_diag_color() (built-in function), 108

set_diag_type() (built-in function), 108
set_drop_lines() (built-in function), 130
set_fg_color() (built-in function), 104
set_first_sheet() (built-in function), 66
set_font_color() (built-in function), 94
set_font_name() (built-in function), 93
set_font_script() (built-in function), 96
set_font_size() (built-in function), 94
set_font_strikeout() (built-in function), 95
set_footer() (built-in function), 86
set_h_pagebreaks() (built-in function), 91
set_header() (built-in function), 82
set_hidden() (built-in function), 99
set_high_low_lines() (built-in function), 131
set_hole_size() (built-in function), 133
set_indent() (built-in function), 101
set_italic() (built-in function), 95
set_landscape() (built-in function), 79
set_left() (built-in function), 106
set_left_color() (built-in function), 107
set_legend() (built-in function), 123
set_locked() (built-in function), 99
set_margins() (built-in function), 81
set_num_format() (built-in function), 96
set_page_view() (built-in function), 79
set_paper() (built-in function), 80
set_pattern() (built-in function), 103
set_plotarea() (built-in function), 127
set_portrait() (built-in function), 79
set_print_scale() (built-in function), 90
set_properties() (built-in function), 27
set_right() (built-in function), 106
set_right_color() (built-in function), 107
set_rotation() (built-in function), 101
set_row() (built-in function), 47
set_selection() (built-in function), 71
set_shrink() (built-in function), 102
set_size() (built-in function), 122
set_start_page() (built-in function), 90
set_style() (built-in function), 128
set_tab_color() (built-in function), 74
set_table() (built-in function), 128
set_text_justlast() (built-in function), 103
set_text_wrap() (built-in function), 101
set_title() (built-in function), 122
set_top() (built-in function), 105
set_top_color() (built-in function), 107
set_underline() (built-in function), 95
set_up_down_bars() (built-in function), 129
set_v_pagebreaks() (built-in function), 91
set_vba_name() (built-in function), 30
set_x2_axis() (built-in function), 121
set_x_axis() (built-in function), 115
set_y2_axis() (built-in function), 121
set_y_axis() (built-in function), 121
set_zoom() (built-in function), 74
show_blanks_as() (built-in function), 132
show_comments() (built-in function), 63
show_hidden_data() (built-in function), 132
split_panes() (built-in function), 73

U

use_zip64() (built-in function), 31

W

Workbook() (built-in function), 21
worksheets() (built-in function), 31
write() (built-in function), 33
write_array_formula() (built-in function), 39
write_blank() (built-in function), 40
write_boolean() (built-in function), 41
write_column() (built-in function), 47
write_comment() (built-in function), 61
write_datetime() (built-in function), 41
write_formula() (built-in function), 38
write_number() (built-in function), 37
write_rich_string() (built-in function), 44
write_row() (built-in function), 46
write_string() (built-in function), 36
write_url() (built-in function), 42

X

xl_cell_to_rowcol() (built-in function), 146
xl_col_to_name() (built-in function), 146
xl_range() (built-in function), 146
xl_range_abs() (built-in function), 147
xl_rowcol_to_cell() (built-in function), 145