Creating Excel files with Python and XlsxWriter

Release 3.1.7

John McNamara

October 09, 2023

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/GIF/BMP/WMF/EMF images.

- Rich multi-format strings.

- Cell comments.

- Textboxes.

- Integration with Pandas and Polars.

- Memory optimization mode for writing large files.

It supports Python 3.4+ and PyPy3 and uses standard libraries only.

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 2-Clause *License* and the source code is available on GitHub.

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

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:

```
$ pip install XlsxWriter

# Or to a non system dir:
$ pip install --user XlsxWriter
```

2.1.2 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
$ 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/main.tar.gz

$ tar zxvf main.tar.gz
$ cd XlsxWriter-main/
$ python setup.py install
```

2.1.3 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
$ python setup.py install
```

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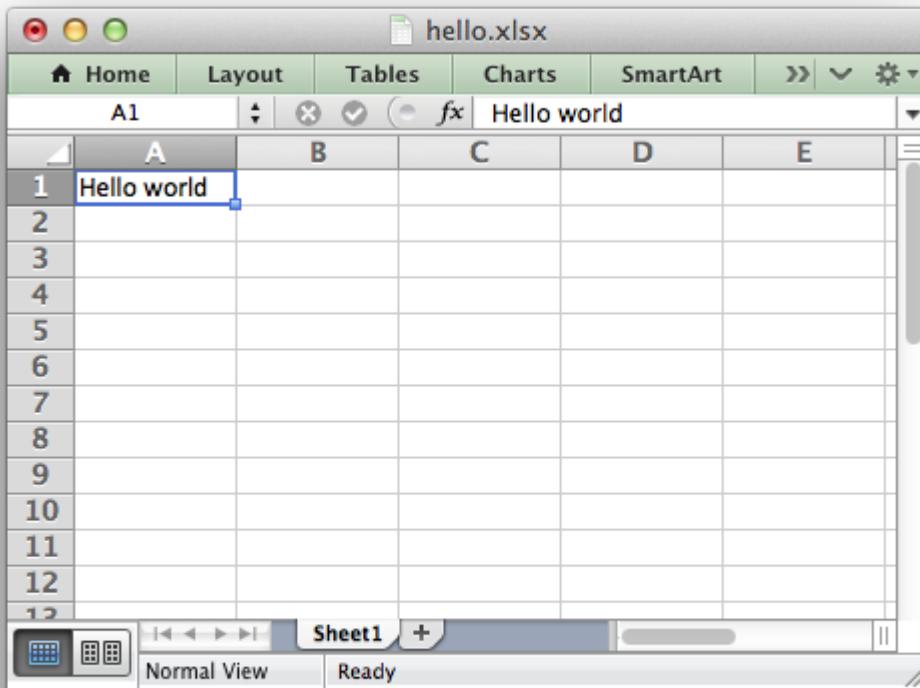
If the installation went correctly you can create a small sample program like the following to verify that the module works correctly:

```python
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.

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:

```python
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)')
```
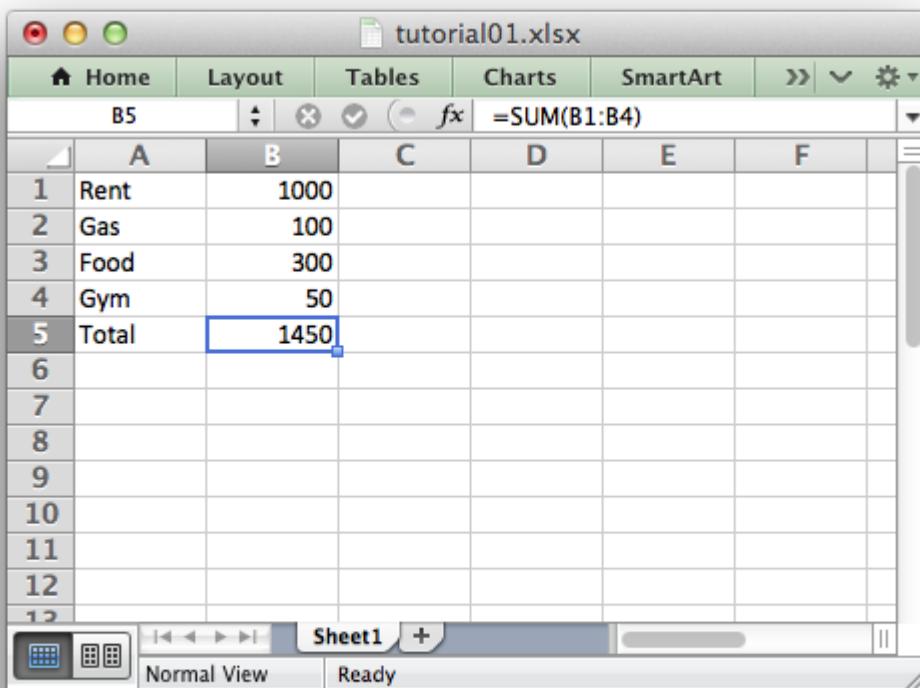
```
workbook.close()
```

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()
```

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

In the next sections we will see how we can use the XlsxWriter module to add formatting and other Excel features.
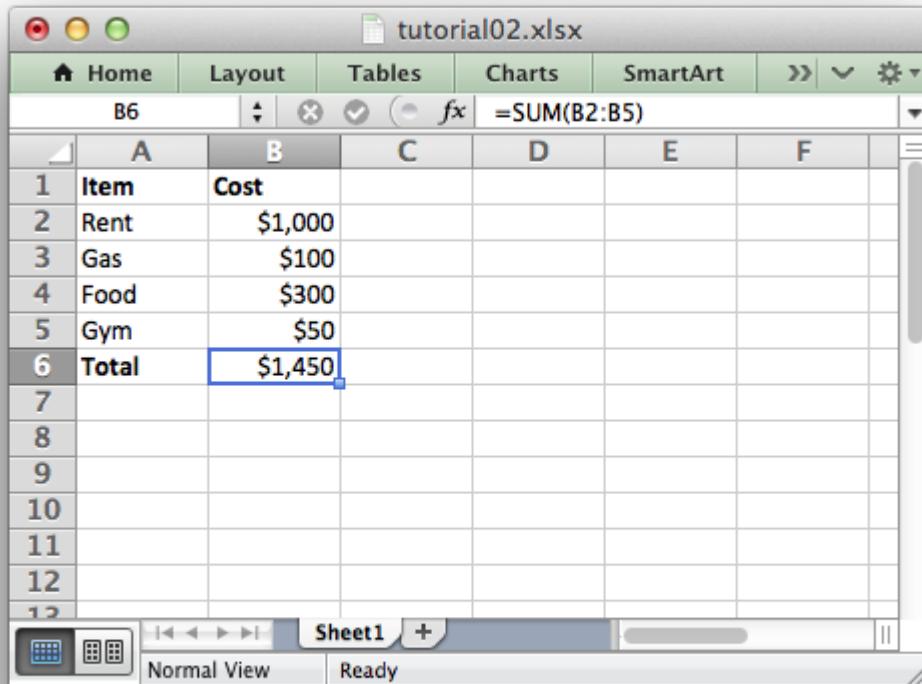
TUTORIAL 2: ADDING FORMATTING TO THE XLSX FILE

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

This converted the required data into an Excel file but it looked a little bare. In order to make the information clearer we would like to add some simple formatting, like this:

The 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:

```python
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* section.

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

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

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

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

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

Like this:

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

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

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

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

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

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

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

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

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

The corresponding spreadsheet will look like this:

The 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:

```python
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*.

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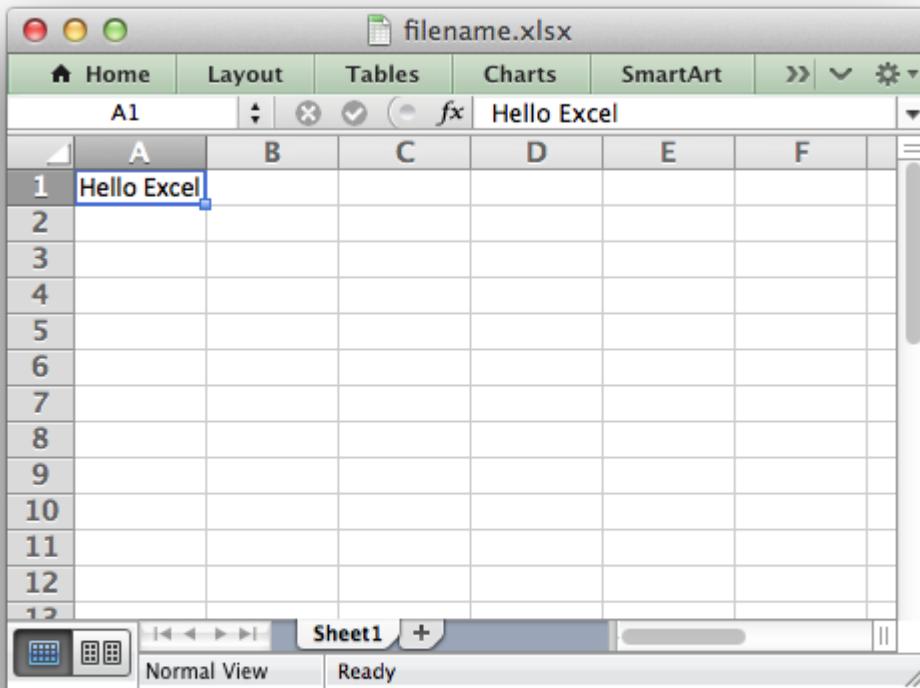
The `Workbook()` constructor is used to create a new Excel workbook with a given filename:

```python
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. For this reason the `add_table()` and `merge_range()` Worksheet methods don't work in this mode.

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

- **tmpdir**: `XlsxWriter` stores workbook data in 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 `tmpdir` 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, set the `in_memory` constructor option to

True:

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

This option overrides the `constant_memory` option.

Note: This option used to be the recommended way of deploying XlsxWriter on Google APP Engine since it didn't support a `/tmp` directory. However, the Python 3 Runtime Environment in Google App Engine supports a *filesystem with read/write access to /tmp* which means this option isn't required.

- **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`. To enable this option use:

```
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`. To disable this option use:

```
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`. To disable this option use:

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

- **use_future_functions**: Enable the use of newer Excel "future" functions without having to prefix them with with `_xlfn.`. The default is `False`. To enable this option use:

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

See also *Formulas added in Excel 2010 and later*.

- **max_url_length**: Set the maximum length for hyperlinks in worksheets. The default is 2079 and the minimum is 255. Versions of Excel prior to Excel 2015 limited hyperlink links and anchor/locations to 255 characters each. Versions after that support urls up to 2079 characters. XlsxWriter versions >= 1.2.3 support the new longer limit by default. However, a lower or user defined limit can be set via the `max_url_length` option:

```
workbook = xlsxwriter.Workbook(filename, {'max_url_length': 255})
```

- **nan_inf_to_errors**: Enable the *worksheet*.`write()` and `write_number()` methods to convert `nan`, `inf` and `-inf` to Excel errors. Excel doesn't handle NAN/INF as numbers so as a workaround they are mapped to formulas that yield the error codes #NUM! and #DIV/0!. The default is `False`. To enable this option use:

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

- **remove_timezone**: Excel doesn't support timezones in datetimes/times so there isn't any fail-safe way that XlsxWriter can map a Python timezone aware datetime into an Excel date-time in functions such as `write_datetime()`. As such the user should convert and re-move the timezones in some way that makes sense according to their requirements. Al-ternatively the `remove_timezone` option can be used to strip the timezone from datetime values. The default is `False`. To enable this option use:

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

See also *Timezone Handling in XlsxWriter*.

- **use_zip64**: Use ZIP64 extensions when writing the xlsx file zip container to allow files greater than 4 GB. This is the same as calling `use_zip64()` after creating the Workbook object. This constructor option is just syntactic sugar to make the use of the option more explicit. The following are equivalent:

```
workbook = xlsxwriter.Workbook(filename, {'use_zip64': True})

# Same as:
workbook = xlsxwriter.Workbook(filename)
workbook.use_zip64()
```

See the note about the Excel warning caused by using this option in `use_zip64()`.

- **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.

The `Workbook()` method also works using the `with` context manager. In which case it doesn't need an explicit *close()* statement:

```
with xlsxwriter.Workbook('hello_world.xlsx') as workbook:
    worksheet = workbook.add_worksheet()

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

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

```
from io import BytesIO

output = BytesIO()
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([*name*])

Add a new worksheet to a workbook.

> **Parameters name** (*string*) – Optional worksheet name, defaults to Sheet1, etc.
>
> **Return type** A *worksheet* object.
>
> **Raises**
>
> - **DuplicateWorksheetName** – if a duplicate worksheet name is used.
>
> - **InvalidWorksheetName** – if an invalid worksheet name is used.
>
> - **ReservedWorksheetName** – if a reserved worksheet name is used.
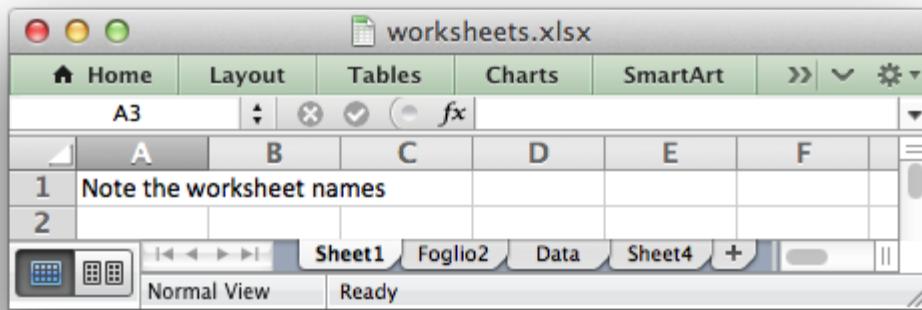
The `add_worksheet()` method adds a new worksheet to a workbook.

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 name parameter is optional. If it is not specified, or blank, 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:

- It must be less than 32 characters. This error will raise a `InvalidWorksheetName` exception.

- It cannot contain any of the characters: [ ] : * ? / \. This error will raise a `InvalidWorksheetName` exception.

- It cannot begin or end with an apostrophe. This error will raise a `InvalidWorksheetName` exception.

- You cannot use the same, case insensitive, name for more than one worksheet. This error will raise a `DuplicateWorksheetName` exception.

- You should not use the Excel reserved name "History", or case insensitive variants as this is restricted in English, and other, versions of Excel.

The rules for worksheet names in Excel are explained in the Microsoft Office documentation on how to Rename a 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* section 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)
name    (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.

- name

  Set the name for the chart sheet:

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

  The name property is optional and if it isn't supplied it will default to `Chart1`, `Chart2`, etc. The name must be a valid Excel chart name.

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()
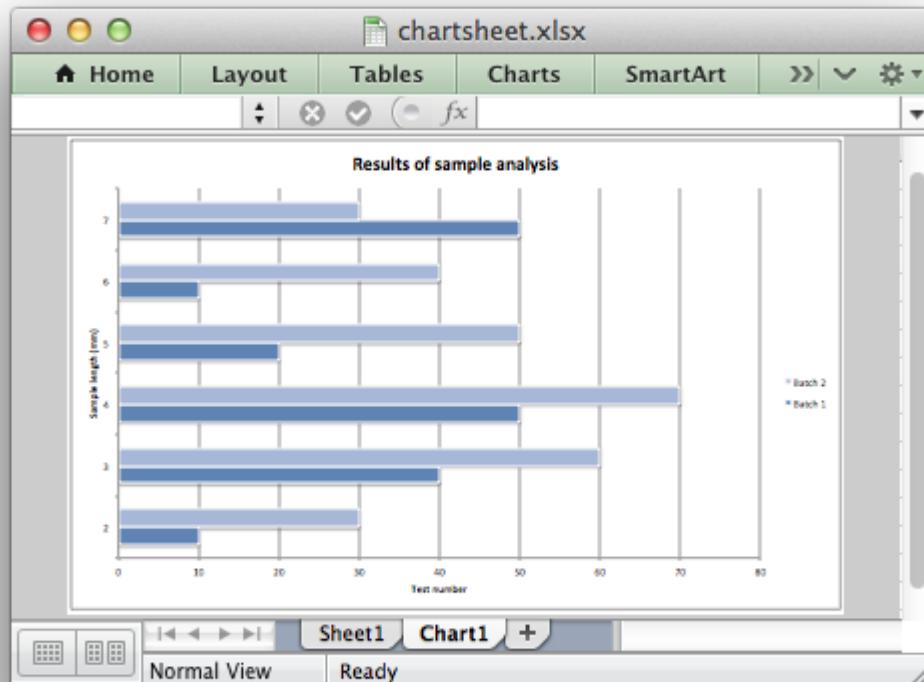
add_chartsheet($[sheetname]$)
>   Add a new add_chartsheet to a workbook.

>>   **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. See add_worksheet() for the limitation on Excel worksheet names.

6.6 workbook.close()

close()
> Close the Workbook object and write the XLSX file.

> > **Raises**

> > > • **FileCreateError** – if there is a file or permissions error during writing.

> > > • **DuplicateTableName** – if a duplicate worksheet table name was added.

> > > • **EmptyChartSeries** – if a chart is added without a data series.

> > > • **UndefinedImageSize** – if an image doesn't contain height/width data.

> > > • **UnsupportedImageFormat** – if an image type isn't supported.

> > > • **FileSizeError** – if the filesize would require ZIP64 extensions.

The workbook close() method writes all data to the xlsx file and closes it:

```
workbook.close()
```

This is a required method call to close and write the xlsxwriter file, unless you are using the with context manager, see below.

The Workbook object also works using the with context manager. In which case it doesn't need an explicit close() statement:

```
With xlsxwriter.Workbook('hello_world.xlsx') as workbook:
    worksheet = workbook.add_worksheet()

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

The workbook will close automatically when exiting the scope of the with statement.

The most common exception during close() is FileCreateError which is generally caused by a write permission error. On Windows this usually occurs if the file being created is already open in Excel. This exception can be caught in a try block where you can instruct the user to close the open file before overwriting it:

```
while True:
    try:
        workbook.close()
    except xlsxwriter.exceptions.FileCreateError as e:
        decision = input("Exception caught in workbook.close(): %s\n"
                         "Please close the file if it is open in Excel.\n"
                         "Try to write file again? [Y/n]: " % e)
        if decision != 'n':
            continue

    break
```

The `close()` method can only write a file once. It doesn't behave like a save method and it cannot be called multiple times to write a file at different stages. If it is called more than once it will raise a `UserWarning` in order to help avoid issues where a file is closed within a loop or at the wrong scope level.

See also *Example: Catch exception on closing*.

6.7 workbook.set_size()

set_size(*width*, *height*)
> Set the size of a workbook window.

>> **Parameters**

>>> • **width** (*int*) – Width of the window in pixels.

>>> • **height** (*int*) – Height of the window in pixels.

The `set_size()` method can be used to set the size of a workbook window:

```
workbook.set_size(1200, 800)
```

The Excel window size was used in Excel 2007 to define the width and height of a workbook window within the Multiple Document Interface (MDI). In later versions of Excel for Windows this interface was dropped. This method is currently only useful when setting the window size in Excel for Mac 2011. The units are pixels and the default size is 1073 x 644.

Note, this doesn't equate exactly to the Excel for Mac pixel size since it is based on the original Excel 2007 for Windows sizing. Some trial and error may be required to get an exact size.
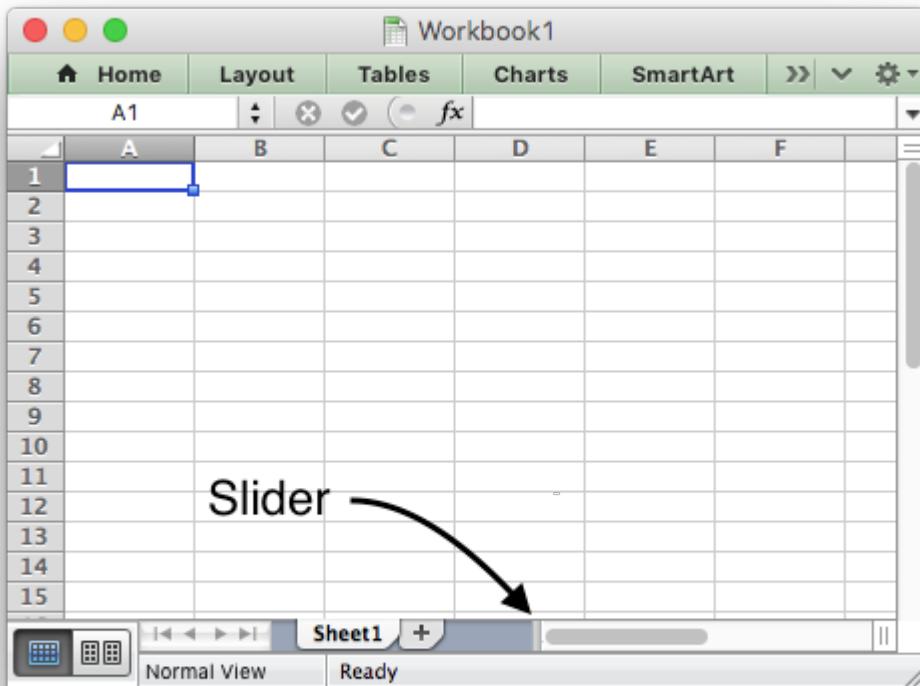
6.8 workbook.tab_ratio()

set_tab_ratio(*tab_ratio*)
> Set the ratio between the worksheet tabs and the horizontal slider.

>> **Parameters tab_ratio** (*float*) – The tab ratio between 0 and 100.

The `set_tab_ratio()` method can be used to set the ratio between worksheet tabs and the horizontal slider at the bottom of a workbook. This can be increased to give more room to the tabs or reduced to increase the size of the horizontal slider:

The default value in Excel is 60. It can be changed as follows:

```
workbook.set_tab_ratio(75)
```

6.9 workbook.set_properties()

set_properties(*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.
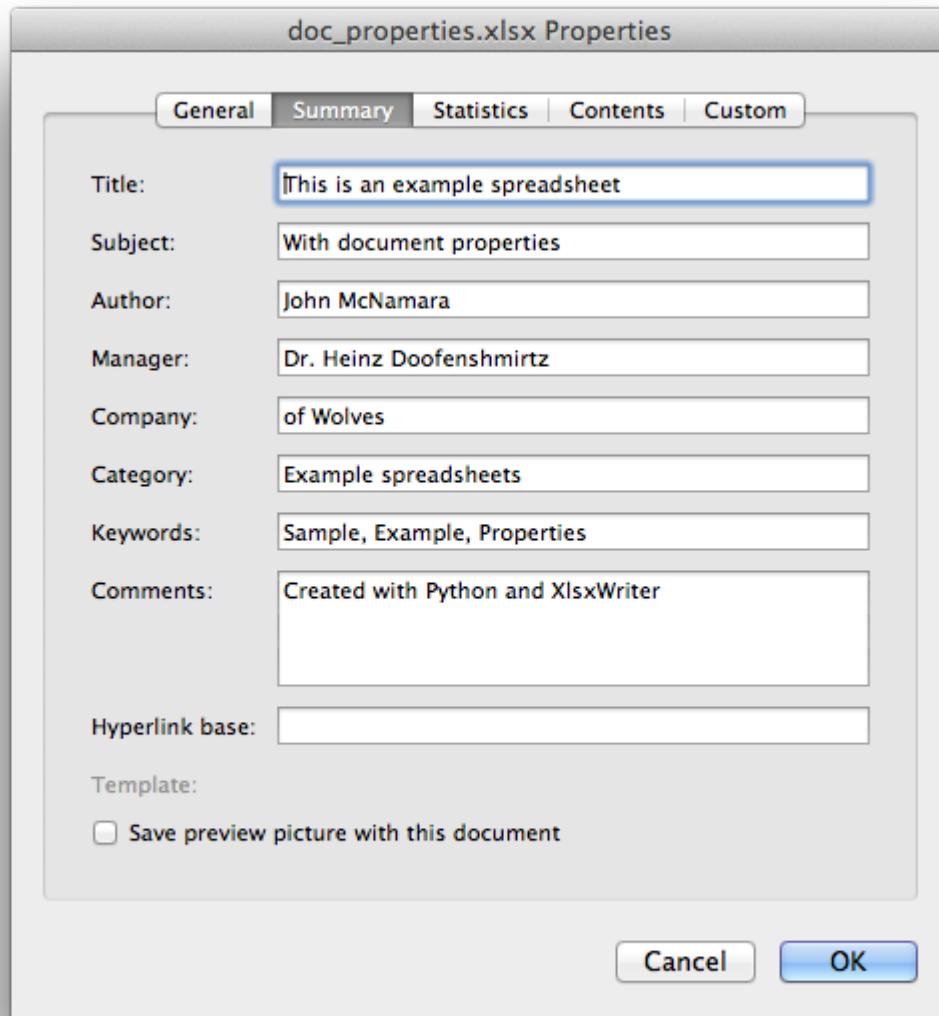
The properties that can be set are:

- title
- subject
- author
- manager

- company

- category

- keywords

- comments

- status

- hyperlink_base

- `created` - the file creation date as a `datetime.date` object.

The properties are all optional and should be passed in dictionary format as follows:

```python
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',
    'created':  datetime.date(2018, 1, 1),
    'comments': 'Created with Python and XlsxWriter'})
```

See also *Example: Setting Document Properties*.

6.10 workbook.set_custom_property()

set_custom_property(*name, value*[, *property_type*])
Set a custom document property.

Parameters
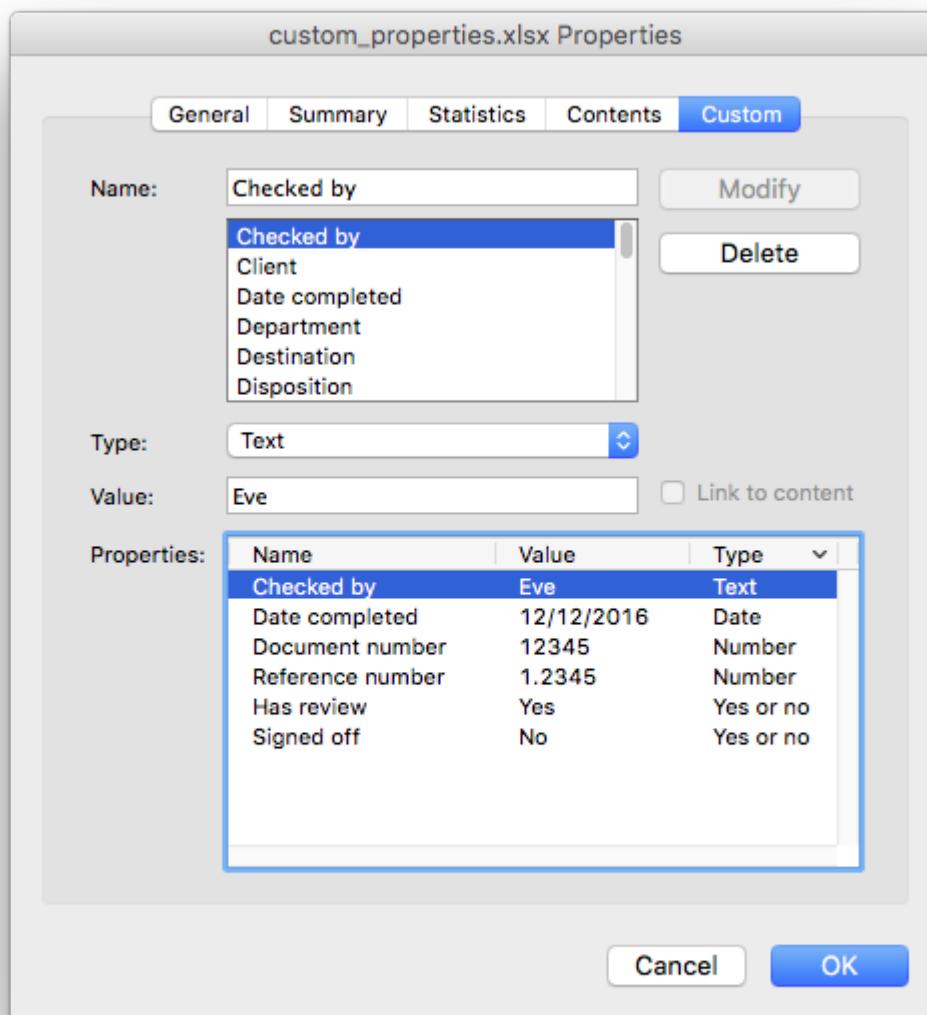
- **name** (*string*) – The name of the custom property.

- **value** – The value of the custom property (various types).

- **property_type** (*string*) – The type of the property. Optional.

The `set_custom_property()` method can be used to set one or more custom document properties not covered by the standard properties in the `set_properties()` method above.

For example:

```
date = datetime.strptime('2016-12-12', '%Y-%m-%d')

workbook.set_custom_property('Checked by',       'Eve')
workbook.set_custom_property('Date completed',   date)
workbook.set_custom_property('Document number',  12345)
workbook.set_custom_property('Reference number', 1.2345)
workbook.set_custom_property('Has review',       True)
workbook.set_custom_property('Signed off',       False)
```

Date parameters should be `datetime.datetime` objects.

The optional `property_type` parameter can be used to set an explicit type for the custom property, just like in Excel. The available types are:

```
text
date
number
bool
```

However, in almost all cases the type will be inferred correctly from the Python type, like in the example above.

Note: the `name` and `value` parameters are limited to 255 characters by Excel.

6.11 workbook.define_name()

define_name()
>   Create a defined name in the workbook to use as a variable.
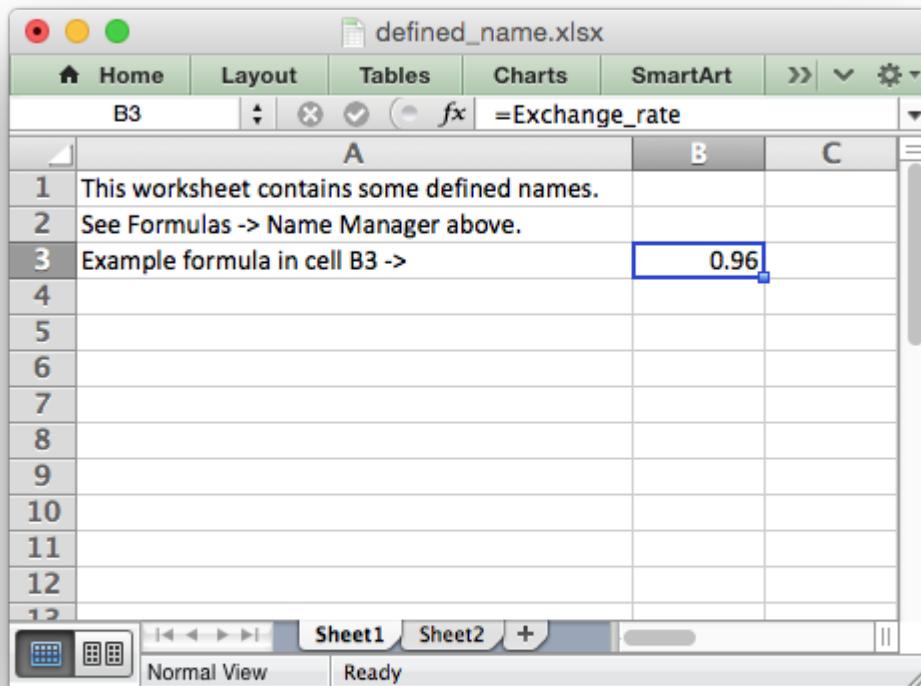
>   **Parameters**

>   >   • **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:

```python
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 Microsoft Office documentation on how to Define and use names in formulas.

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

6.12 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 name doesn't have to be `vbaProject.bin`. Any suitable path/name for an existing VBA bin file will do.

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.

6.13 workbook.add_signed_vba_project()

**add_signed_vba_project(vba_project,
signature [,
project_is_stream, [
signature_is_stream]]):**

    Add a vbaProject binary and a vbaProjectSignature binary to the Excel workbook.

> **Parameters**
>
> > - **vba_project** – The vbaProject binary file name.
> >
> > - **signature** – The vbaProjectSignature binary file name.
> >
> > - **project_is_stream** (*bool*) – The vba_project is an in memory byte stream.
> >
> > - **signature_is_stream** (*bool*) – The signature is an in memory byte stream.

The `add_signed_vba_project()` method can be used to add digitally signed macros or functions to a workbook. The method adds a binary VBA project file and a binary VBA project signature file that have been extracted from an existing Excel xlsm file with digitally signed macros:

```
workbook.add_signed_vba_project('./vbaProject.bin', './vbaProjectSignature.bin')
```

Only one `vbaProject.bin` file can be added per workbook. The name doesn't have to be `vbaProject.bin`. Any suitable path/name for an existing VBA bin file will do. The same applies for `vbaProjectSignature.bin`.

The `project_is_stream` (`signature_is_stream`, resp.) parameter is used to indicate that `vba_project` (`signature`, resp.) 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.

6.14 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.

6.15 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:

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

6.16 workbook.get_worksheet_by_name()

get_worksheet_by_name(*name*)
> Return a worksheet object in the workbook using the sheetname.

> > **Parameters name** (*string*) – Name of worksheet that you wish to retrieve.

> > **Return type** A *worksheet* object.

The `get_worksheet_by_name()` method returns the worksheet or chartsheet object with the given name or None if it isn't found:

```python
worksheet = workbook.get_worksheet_by_name('Sheet1')
```

6.17 workbook.get_default_url_format()

get_default_url_format()
> Return a format object.

> > **Return type** A *format* object.

The `get_default_url_format()` method gets a copy of the default url format used when a user defined format isn't specified with `write_url()`. The format is the hyperlink style defined by Excel for the default theme:

```
url_format = workbook.get_default_url_format()
```

6.18 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.

6.19 workbook.use_zip64()

`use_zip64`()
> Allow ZIP64 extensions when writing the xlsx file zip container.

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

Note: When using the `use_zip64()` option the zip file created by the Python standard library `zipfile.py` may cause Excel to issue a warning about repairing the file. This warning is annoying but harmless. The "repaired" file will contain all of the data written by XlsxWriter, only the zip container will be changed.

6.20 workbook.read_only_recommended()
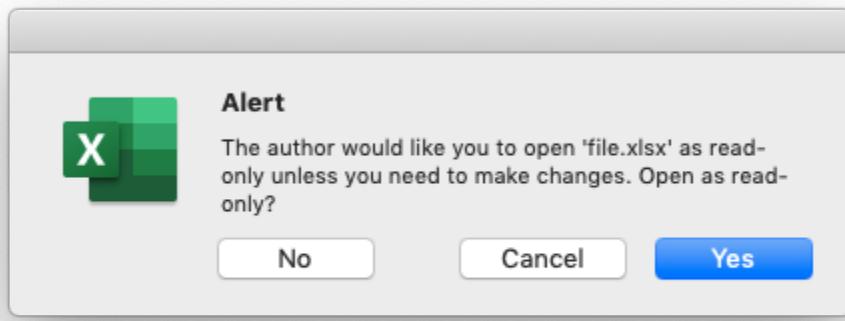
`read_only_recommended`()
> Add a recommendation to open the file in "read-only" mode.

This method can be used to set the Excel "Read-only Recommended" option that is available when saving a file. This presents the user of the file with an option to open it in "read-only" mode. This means that any changes to the file can't be saved back to the same file and must be saved to a new file. It can be set as follows:

```python
import xlsxwriter

workbook = xlsxwriter.Workbook('file.xlsx')
```

```
worksheet = workbook.add_worksheet()

workbook.read_only_recommended()

workbook.close()
```

Which will raise a dialog like the following when opening the file:

THE WORKSHEET CLASS

The worksheet class represents an Excel worksheet. It handles operations such as writing data to cells or formatting worksheet layout.
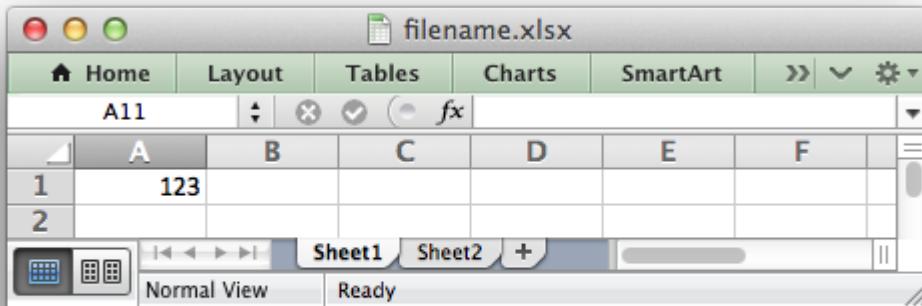
A worksheet object isn't instantiated directly. Instead a new worksheet is created by calling the `add_worksheet()` method from a `Workbook()` object:

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

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

worksheet1.write('A1', 123)

workbook.close()
```


XlsxWriter supports Excels worksheet limits of 1,048,576 rows by 16,384 columns.

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.
>
> **Returns** 0: Success.
>
> **Returns** -1: Row or column is out of worksheet bounds.
>
> **Returns** Other values from the called write methods.

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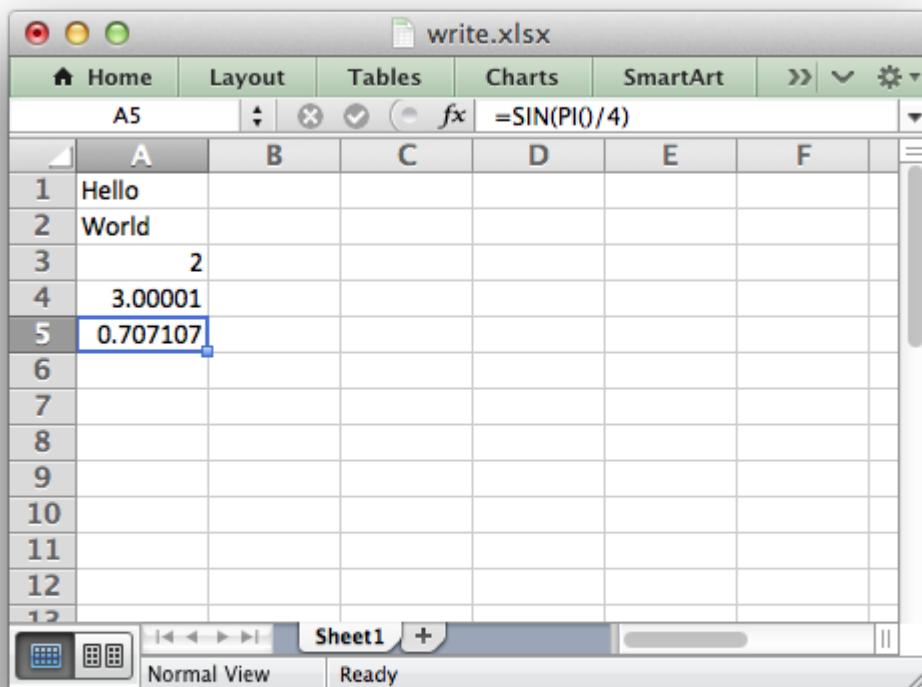
If none of the above types are matched the value is evaluated with `float()` to see if it corresponds to a user defined float type. If it does then it is written using `write_number()`.

Finally, if none of these rules are matched then a `TypeError` exception is raised. However, it is also possible to handle additional, user defined, data types using the `add_write_handler()` method explained below and in *Writing user defined types*.

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.add_write_handler()

add_write_handler(*user_type*, *user_function*)
    Add a callback function to the write() method to handle user define types.

        Parameters

                • **user_type** (*type*) – The user type() to match on.

                • **user_function** (*types.FunctionType*) – The user defined function to write
                  the type data.

As explained above, the write() method maps basic Python types to corresponding Excel types. If you want to write an unsupported type then you can either avoid write() and map the user type in your code to one of the more specific write methods or you can extend it using the add_write_handler() method.

For example, say you wanted to automatically write uuid values as strings using write() you would start by creating a function that takes the uuid, converts it to a string and then writes it using write_string():

```
def write_uuid(worksheet, row, col, uuid, format=None):
    string_uuid = str(uuid)
    return worksheet.write_string(row, col, string_uuid, format)
```
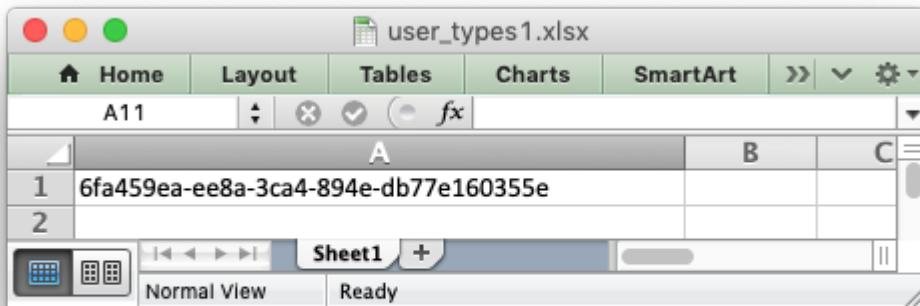
You could then add a handler that matches the uuid type and calls your user defined function:

```
#                                   match,     action()
worksheet.add_write_handler(uuid.UUID, write_uuid)
```

Then you can use write() without further modification:

```
my_uuid = uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')

# Write the UUID. This would raise a TypeError without the handler.
worksheet.write('A1', my_uuid)
```


Multiple callback functions can be added using `add_write_handler()` but only one callback action is allowed per type. However, it is valid to use the same callback function for different types:

```
worksheet.add_write_handler(int,   test_number_range)
worksheet.add_write_handler(float, test_number_range)
```

See *Writing user defined types* for more details on how this feature works and how to write callback functions, and also the following examples:

- *Example: Writing User Defined Types (1)*
- *Example: Writing User Defined Types (2)*
- *Example: Writing User Defined types (3)*

7.3 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.

  Returns 0: Success.

  Returns -1: Row or column is out of worksheet bounds.

> **Returns** -2: String truncated to 32k characters.

The `write_string()` method writes a string to the cell specified by `row` and `column`:
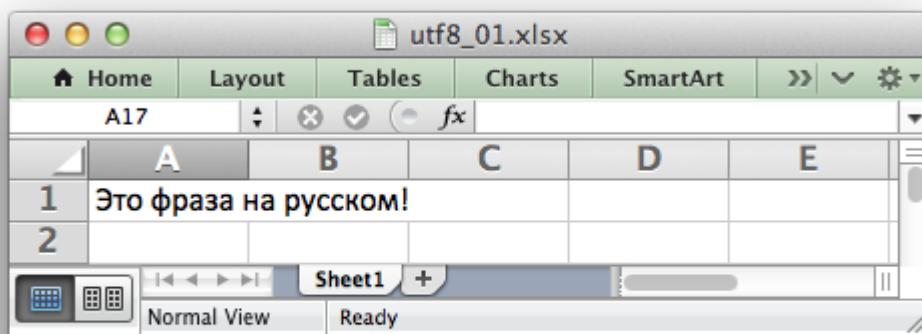
```
worksheet.write_string(0, 0, 'Your text here')
worksheet.write_string('A2', 'or here')
```

Both row-column and A1 style notation are supported, as shown above. 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 is UTF-8 encoded:

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


See *Example: Simple Unicode with Python 3* for a more complete example.

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. See *Example: Unicode - Polish in UTF-8* and *Example: Unicode - Shift JIS*.

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.

7.4 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.

Returns 0: Success.

Returns -1: Row or column is out of worksheet bounds.

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)
```

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

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 unless the nan_inf_to_errors Workbook() option is used.

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.

7.5 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.

- **value** – Optional result. The value if the formula was calculated.

Returns 0: Success.

Returns -1: Row or column is out of worksheet bounds.

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")')
```

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

Array formulas are also supported:

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

See also the `write_array_formula()` method below.

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 result of the formula using the optional `value` parameter. This is occasionally necessary when working with non-Excel applications that don't calculate the result of the formula:

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

See *Formula Results* for more details.

Excel stores formulas in US style formatting regardless of the Locale or Language of the Excel version:

```
worksheet.write_formula('A1', '=SUM(1, 2, 3)')    # OK
worksheet.write_formula('A2', '=SOMME(1, 2, 3)')  # French. Error on load.
```

See *Non US Excel functions and syntax* for a full explanation.

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`. In XlsxWriter these require a prefix:

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

See *Formulas added in Excel 2010 and later* for a detailed explanation and full list of functions that are affected.

7.6 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.

- **value** – Optional result. The value if the formula was calculated.

Returns 0: Success.

Returns -1: Row or column is out of worksheet bounds.

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

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

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)}')
```

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 result of the formula (see discussion of formulas and the `value` parameter for the `write_formula()` method above). However, using this parameter only writes a single value to the upper left cell in the result array. See *Formula Results* for more details.

See also *Example: Array formulas*.

7.7 worksheet.write_dynamic_array_formula()

write_dynamic_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.
- **value** – Optional result. The value if the formula was calculated.

        Returns 0: Success.
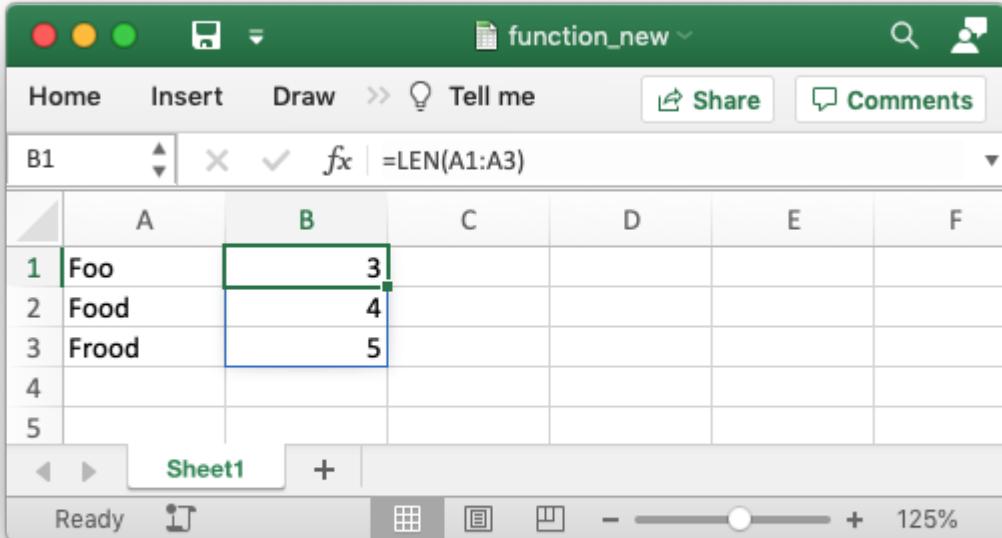
        Returns -1: Row or column is out of worksheet bounds.

The `write_dynamic_array_formula()` method writes an dynamic array formula to a cell range. Dynamic array formulas are explained in detail in *Dynamic Array support*.

The syntax of `write_dynamic_array_formula()` is the same as `write_array_formula()`, shown above, except that you don't need to add {} braces:

```
worksheet.write_dynamic_array_formula('B1:B3', '=LEN(A1:A3)')
```

Which gives the following result:

It is also possible to specify the first cell of the range to get the same results:

```
worksheet.write_dynamic_array_formula('B1:B1', '=LEN(A1:A3)')
```

See also *Example: Dynamic array formulas*.

7.8 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.
>
> **Returns** 0: Success.
>
> **Returns** -1: Row or column is out of worksheet bounds.

Write a blank cell specified by `row` and `column`:

```
worksheet.write_blank(0, 0, None, cell_format)
worksheet.write_blank('A2', None, cell_format)
```

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

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, cell_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.

7.9 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.

>> **Returns** 0: Success.

>> **Returns** -1: Row or column is out of worksheet bounds.

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, as shown above. 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.

7.10 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.

Returns 0: Success.

Returns -1: Row or column is out of worksheet bounds.

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)
worksheet.write_datetime('A2', datetime, date_format)
```

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

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

There are many ways to create datetime objects, for example the `date-time.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 de-fault_date_format option.

See *Working with Dates and Time* for more details and also *Timezone Handling in XlsxWriter*.

7.11 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 the Excel hy-perlink style.

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

- **tip** (*string*) – An optional tooltip.

Returns 0: Success.

Returns -1: Row or column is out of worksheet bounds.

Returns -2: String longer than 32k characters.

Returns -3: Url longer than Excel limit of 2079 characters.

Returns -4: Exceeds Excel limit of 65,530 urls per worksheet.

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:

```
worksheet.write_url(0, 0, 'https://www.python.org/')
worksheet.write_url('A2', 'https://www.python.org/')
```

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

The `cell_format` parameter is used to apply formatting to the cell. This parameter is optional and the default Excel hyperlink style will be used if it isn't specified. If required you can access the default url format using the Workbook `get_default_url_format()` method:

```
url_format = workbook.get_default_url_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', 'https://www.python.org/')
worksheet.write_url('A3', 'mailto:jmcnamara@cpan.org')
```

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

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

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

```
worksheet.write_url('A1', 'https://www.python.org', string='Python home')
```

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', 'https://www.python.org/')

# Overwrite the URL string with a formula. The cell will still be a link.
# Note the use of the default url format for consistency with other links.
url_format = workbook.get_default_url_format()
worksheet.write_formula('A1', '=1+1', url_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:

```
# Link to a cell on the current worksheet.
worksheet.write_url('A1',  'internal:Sheet2!A1')

# Link to a cell on another worksheet.
worksheet.write_url('A2',  'internal:Sheet2!A1:B2')

# Worksheet names with spaces should be single quoted like in Excel.
worksheet.write_url('A3',  "internal:'Sales Data'!A1")

# Link to another Excel workbook.
worksheet.write_url('A4', r'external:c:\temp\foo.xlsx')

# Link to a worksheet cell in another workbook.
worksheet.write_url('A5', r'external:c:\foo.xlsx#Sheet2!A1')

# Link to a worksheet in another workbook with a relative link.
worksheet.write_url('A7', r'external:..\foo.xlsx#Sheet2!A1')

# Link to a worksheet in another workbook with a network link.
worksheet.write_url('A8', 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.

Note: Versions of Excel prior to Excel 2015 limited hyperlink links and anchor/locations to 255 characters each. Versions after that support urls up to 2079 characters. XlsxWriter versions >= 1.2.3 support this longer limit by default. However, a lower or user defined limit can be set via the `max_url_length` property in the `Workbook()` constructor.

7.12 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** (*list*) – String and format pairs.

　　　　　　　　• **cell_format** (*Format*) – Optional Format object.

　　　　Returns 0: Success.

　　　　Returns -1: Row or column is out of worksheet bounds.

　　　　Returns -2: String longer than 32k characters.

　　　　Returns -3: 2 consecutive formats used.
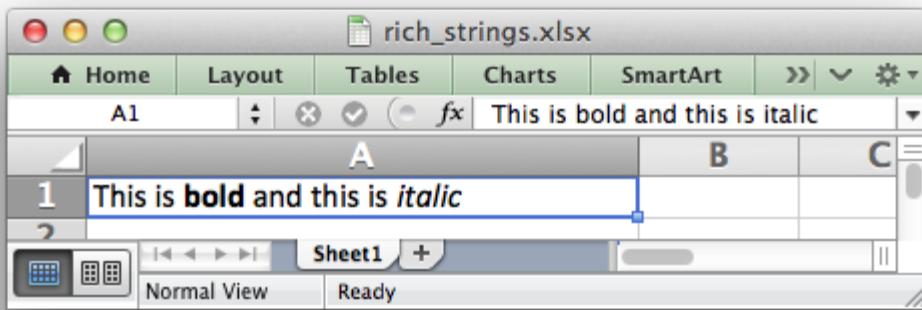
　　　　Returns -4: Empty string used.

　　　　Returns -5: Insufficient parameters.

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')
```

Both row-column and A1 style notation are supported. The following are equivalent:

```
worksheet.write_rich_string(0, 0, 'This is ', bold, 'bold')
worksheet.write_rich_string('A1', 'This is ', bold, 'bold')
```

See *Working with Cell Notation* for more details.

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')
```

If you have formats and segments in a list you can add them like this, using the standard Python list unpacking syntax:

```
segments = ['This is ', bold, 'bold', ' and this is ', blue, 'blue']
worksheet.write_rich_string('A9', *segments)
```

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)
```

Note: Excel doesn't allow the use of two consecutive formats in a rich string or an empty string fragment. For either of these conditions a warning is raised and the input to `write_rich_string()` is ignored.

Also, the maximum string size supported by Excel is 32,767 characters. If the rich string exceeds this limit a warning is raised and the input to `write_rich_string()` is ignored.

See also *Example: Writing "Rich" strings with multiple formats* and *Example: Merging Cells with a Rich String*.

7.13 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.

Returns 0: Success.

Returns Other: Error return value of the `write()` method.

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])
```

Both row-column and A1 style notation are supported. The following are equivalent:

```
worksheet.write_row(0, 0, data)
worksheet.write_row('A1', data)
```

See *Working with Cell Notation* for more details.

7.14 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.
>
> **Returns** 0: Success.
>
> **Returns** Other: Error return value of the `write()` method.

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])
```

Both row-column and A1 style notation are supported. The following are equivalent:

```
    worksheet.write_column(0, 0, data)
    worksheet.write_column('A1', data)
```

See *Working with Cell Notation* for more details.

7.15 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, in character units.

     • **cell_format** (*Format*) – Optional Format object.

     • **options** (*dict*) – Optional row parameters: hidden, level, collapsed.

   Returns 0: Success.

   Returns -1: Row is out of worksheet bounds.

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 height is specified in character units. To specify the height in pixels use the `set_row_pixels()` method.

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 default row 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})
```

7.16 worksheet.set_row_pixels()

set_row_pixels(*row*, *height*, *cell_format*, *options*)
     Set properties for a row of cells, with the row height in pixels.

Parameters

- **row** (*int*) – The worksheet row (zero indexed).

- **height** (*float*) – The row height, in pixels.

- **cell_format** (*Format*) – Optional Format object.

- **options** (*dict*) – Optional row parameters: hidden, level, collapsed.

Returns 0: Success.

Returns -1: Row is out of worksheet bounds.

The `set_row_pixels()` method is identical to `set_row()` except that the height can be set in pixels instead of Excel character units:

```
worksheet.set_row_pixels(0, 18)  # Same as 24 in character units.
```

All other parameters and options are the same as `set_row()`. See the documentation on `set_row()` for more details.

7.17 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 first_col.

- **width** (*float*) – The width of the column(s), in character units.

- **cell_format** (*Format*) – Optional Format object.

- **options** (*dict*) – Optional parameters: hidden, level, collapsed.

Returns 0: Success.

Returns -1: Column is out of worksheet bounds.

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` parameter sets the column width in the same units used by Excel which is: the number of characters in the default font. The default width is 8.43 in the default font of Calibri 11. The actual relationship between a string width and a column width in Excel is complex. See the following explanation of column widths from the Microsoft support documentation for more details. To set the width in pixels use the `set_column_pixels()` method.

See also the `autofit()` method for simulated autofitting of column widths.

As usual the `cell_format` *Format* parameter is optional. If you wish to set the format without changing the default column 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})
```

7.18 worksheet.set_column_pixels()

set_column_pixels(*first_col*, *last_col*, *width*, *cell_format*, *options*)
    Set properties for one or more columns of cells, with the width in pixels.

   Parameters

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

   Returns 0: Success.

   Returns -1: Column is out of worksheet bounds.

The set_column_pixels() method is identical to set_column() except that the width can be set in pixels instead of Excel character units:

```
worksheet.set_column_pixels(5, 5, 75)  # Same as 10 character units.
```

All other parameters and options are the same as `set_column()`. See the documentation on `set_column()` for more details.

7.19 worksheet.autofit()
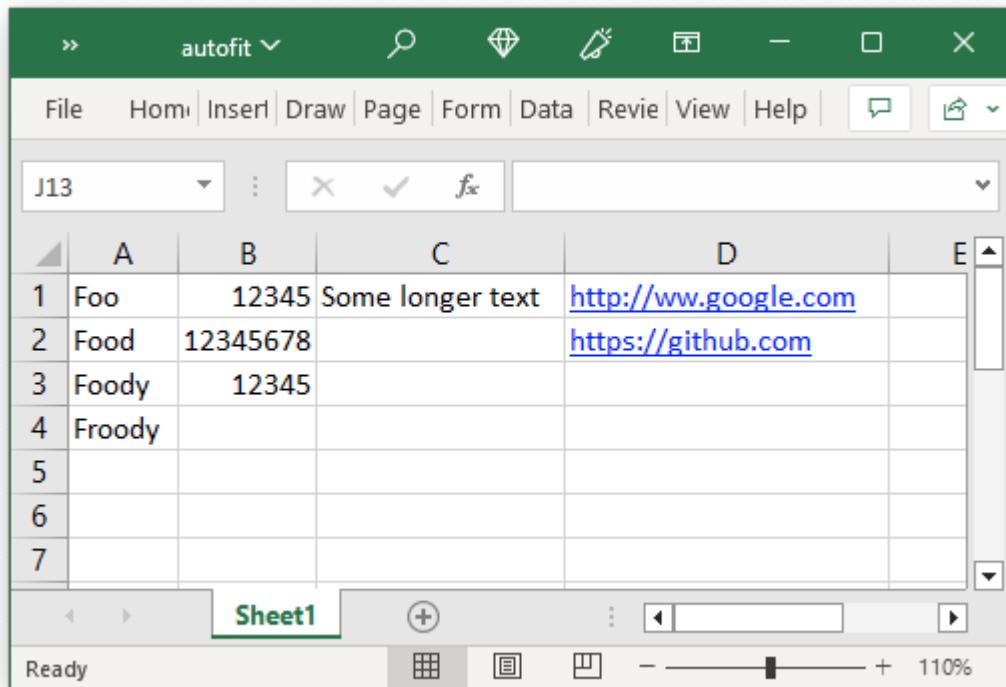
autofit()

Simulates autofit for column widths.

> **Returns** Nothing.

The `autofit()` method can be used to simulate autofitting column widths based on the largest string/number in the column:

```
worksheet.autofit()
```

See *Example: Autofitting columns*

There is no option in the xlsx file format that can be used to say "autofit columns on loading". Auto-fitting of columns is something that Excel does at runtime when it has access to all of the worksheet information as well as the Windows functions for calculating display areas based on fonts and formatting.

The `worksheet.autofit()` method simulates this behavior by calculating string widths using metrics taken from Excel. As such there are some limitations to be aware of when using this method:

- It is a simulated method and may not be accurate in all cases.

- It is based on the default font and font size of Calibri 11. It will not give accurate results for other fonts or font sizes.

This isn't perfect but for most cases it should be sufficient and if not you can set your own widths, see below.

The `autofit()` method won't override a user defined column width set with `set_column()` or `set_column_pixels()` if it is greater than the autofit value. This allows the user to set a minimum width value for a column.

You can also call `set_column()` and `set_column_pixels()` after `autofit()` to override any of the calculated values.

7.20 worksheet.insert_image()

insert_image(*row*, *col*, *filename*[, *options*])
> Insert an image in a worksheet cell.
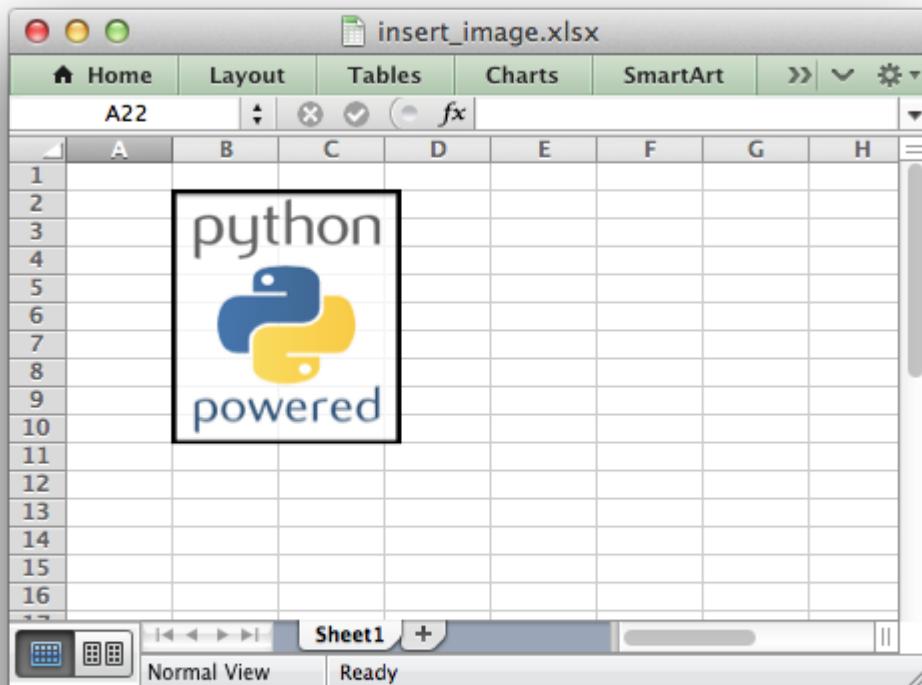
> **Parameters**
>> • **row** (*int*) – The cell row (zero indexed).
>>
>> • **col** (*int*) – The cell column (zero indexed).
>>
>> • **filename** – Image filename (with path if required).
>>
>> • **options** (*dict*) – Optional parameters for image position, scale and url.

> **Returns** 0: Success.

> **Returns** -1: Row or column is out of worksheet bounds.

This method can be used to insert a image into a worksheet. The image can be in PNG, JPEG, GIF, BMP, WMF or EMF format (see the notes about BMP and EMF below):

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


Both row-column and A1 style notation are supported. The following are equivalent:

```
worksheet.insert_image(1, 1, 'python.png')
worksheet.insert_image('B2', 'python.png')
```

See *Working with Cell Notation* for more details.

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,
    'object_position': 2,
    'image_data':      None,
    'url':             None,
    'description':     None,
    'decorative':      False,
}
```

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 optional mouseover tooltip for images with hyperlinks:

```
worksheet.insert_image('B4', 'python.png', {'url': 'https://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:
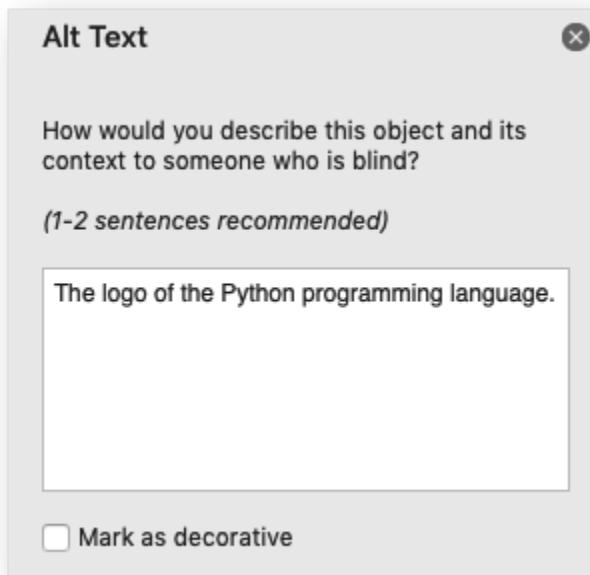
```
url = 'https://python.org/logo.png'
image_data = io.BytesIO(urllib2.urlopen(url).read())

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 used by Excel as a default description field (see below). However, it can be a blank string if the description isn't required. In the previous example the filename/description is extracted from the URL string. See also *Example: Inserting images from a URL or byte stream into a worksheet*.

The `description` field can be used to specify a description or "alt text" string for the image. In general this would be used to provide a text description of the image to help accessibility. It is an optional parameter and defaults to the filename of the image. It can be used as follows:

```
worksheet.insert_image('B3', 'python.png',
                       {'description': 'The logo of the Python programming language.'}
```


The optional `decorative` parameter is also used to help accessibility. It is used to mark the image as decorative, and thus uninformative, for automated screen readers. As in Excel, if this parameter is in use the `description` field isn't written. It is used as follows:

```
worksheet.insert_image('B3', 'python.png', {'decorative': True})
```

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

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

Where `object_position` 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.

4. Same as Option 1 to "move and size with cells" except XlsxWriter applies hidden cells after the image is inserted.

See *Working with Object Positioning* for more detailed information about the positioning and scaling of images within a worksheet.

Note:

- 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 color, bitmaps.

- EMF images can have very small differences in width and height when compared to Excel files. Despite a lot of effort and testing it wasn't possible to exactly match Excel's calculations for handling the dimensions of EMF files. However, the differences are small (< 1%) and in general aren't visible.

See also *Example: Inserting images into a worksheet*.

7.21 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.

> **Returns** 0: Success.
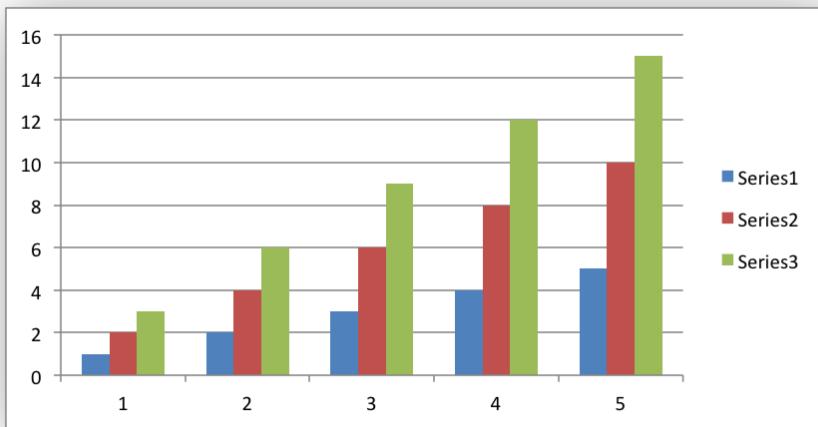
> **Returns** -1: Row or column is out of worksheet bounds.

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*.

Both row-column and A1 style notation are supported. The following are equivalent:

```
worksheet.insert_chart(4, 1, chart)
worksheet.insert_chart('B5', chart)
```

See *Working with Cell Notation* for more details.

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,
    'object_position': 1,
    'description':     None,
    'decorative':      False,
}
```

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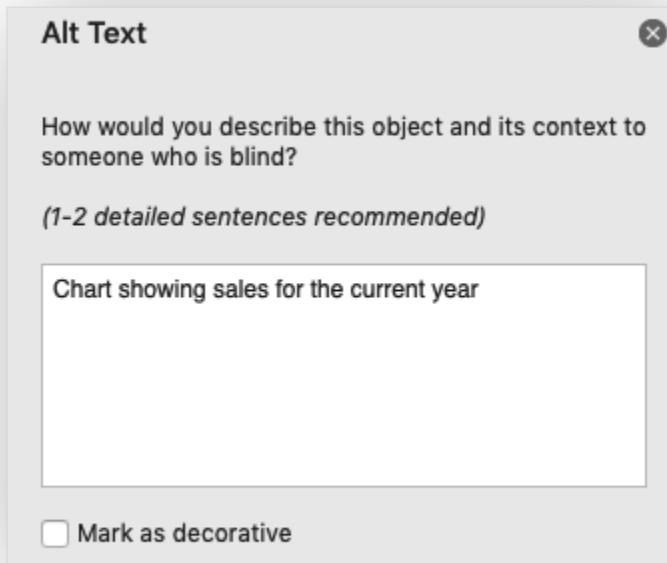
```
worksheet.insert_chart('B5', chart, {'x_scale': 0.5, 'y_scale': 0.5})
```

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

The `description` field can be used to specify a description or "alt text" string for the chart. In

---

general this would be used to provide a text description of the chart to help accessibility. It is an optional parameter and has no default. It can be used as follows:

```
worksheet.insert_chart('B5', chart,
                       {'description': 'Chart showing sales for the current year'})
```


The optional `decorative` parameter is also used to help accessibility. It is used to mark the chart as decorative, and thus uninformative, for automated screen readers. As in Excel, if this parameter is in use the `description` field isn't written. It is used as follows:

```
worksheet.insert_chart('B5', chart, {'decorative': True})
```

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

```
worksheet.insert_chart('B5', chart, {'object_position': 2})
```

Where `object_position` has the following allowable values:

1. Move and size with cells (the default).

2. Move but don't size with cells.

3. Don't move or size with cells.

See *Working with Object Positioning* for more detailed information about the positioning and scaling of charts within a worksheet.

7.22 worksheet.insert_textbox()

insert_textbox(*row*, *col*, *textbox*[, *options*])
> Write a string to a worksheet cell.
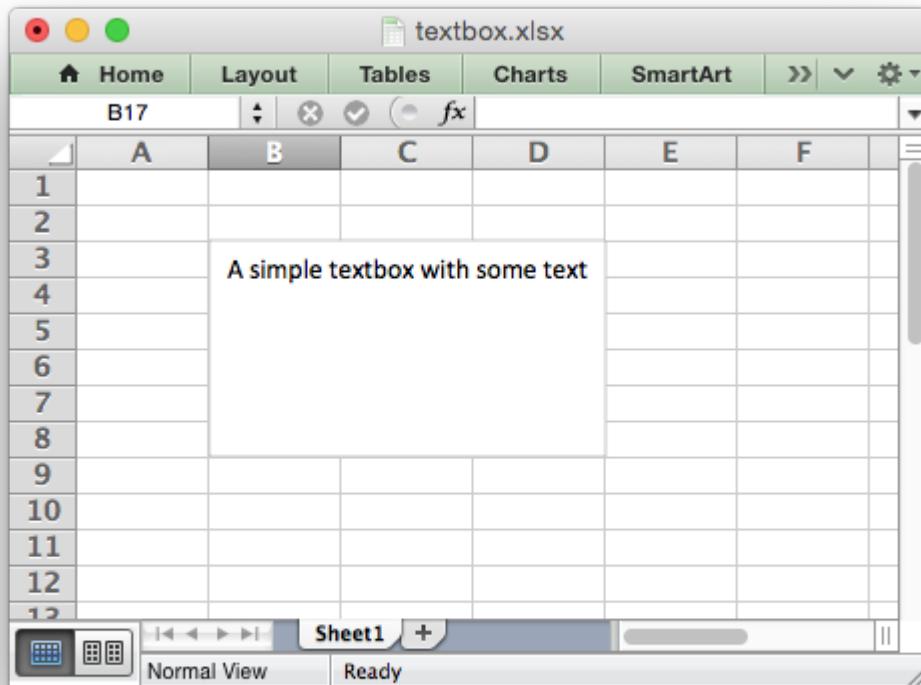
> **Parameters**

> > • **row** (*int*) – The cell row (zero indexed).

> > • **col** (*int*) – The cell column (zero indexed).

> > • **text** (*string*) – The text in the textbox.

> > • **options** (*dict*) – Optional parameters to position and scale the textbox.

> **Returns** 0: Success.

> **Returns** -1: Row or column is out of worksheet bounds.

This method can be used to insert a textbox into a worksheet:

```
worksheet.insert_textbox('B2', 'A simple textbox with some text')
```


Both row-column and A1 style notation are supported. The following are equivalent:

```
worksheet.insert_textbox(1, 1, 'Some text')
worksheet.insert_textbox('B2', 'Some text')
```

See *Working with Cell Notation* for more details.

The size and formatting of the textbox can be controlled via the `options` dict:

```
# Size and position
width
height
x_scale
y_scale
x_offset
y_offset
object_position

# Formatting
line
border
fill
gradient
font
align
text_rotation

# Links
textlink
url
tip

# Accessibility
description
decorative
```

These options are explained in more detail in the *Working with Textboxes* section.

See also *Example: Insert Textboxes into a Worksheet*.

See *Working with Object Positioning* for more detailed information about the positioning and scaling of images within a worksheet.

7.23 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.

Returns 0: Success.
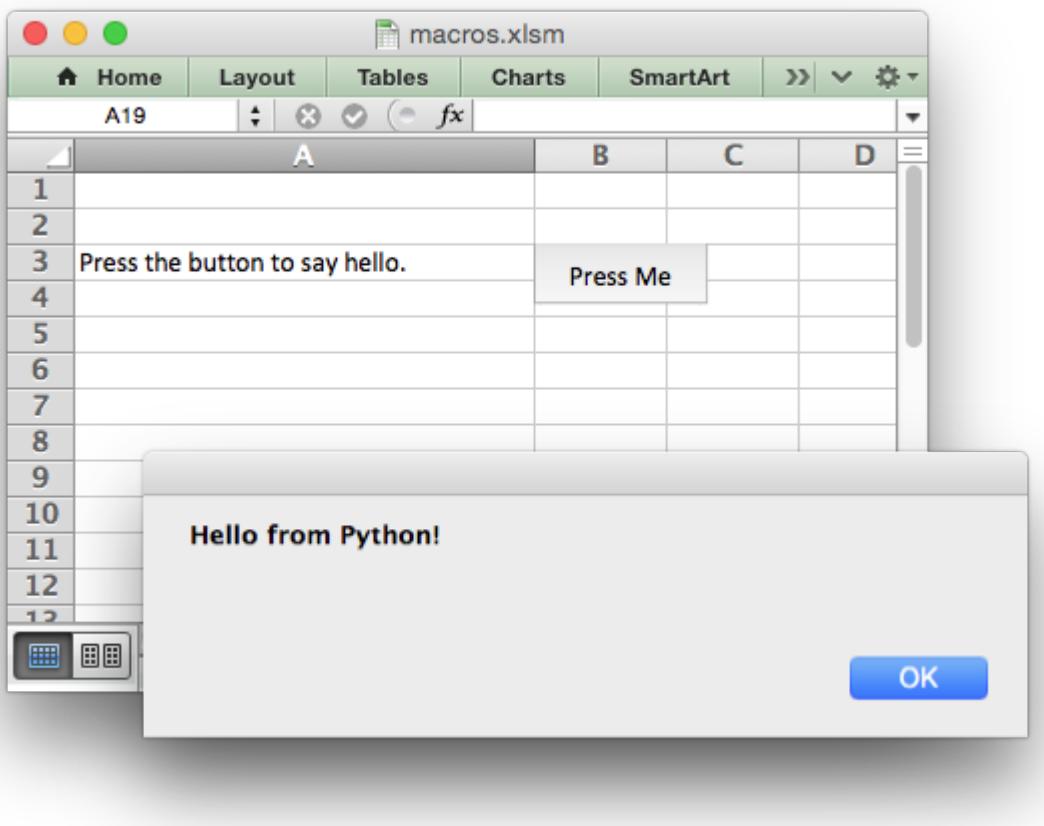
Returns -1: Row or column is out of worksheet bounds.

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:

```python
# 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.

Both row-column and A1 style notation are supported. The following are equivalent:

```python
worksheet.insert_button(2, 1, {'macro':   'say_hello',
                               'caption': 'Press Me'})
```

```
    worksheet.insert_button('B3', {'macro':   'say_hello',
                                   'caption': 'Press Me'})
```

See *Working with Cell Notation* 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,
    'description': None,
}
```

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, scale and description options are the same as for insert_chart(), see above.

7.24 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.

        Returns 0: Success.
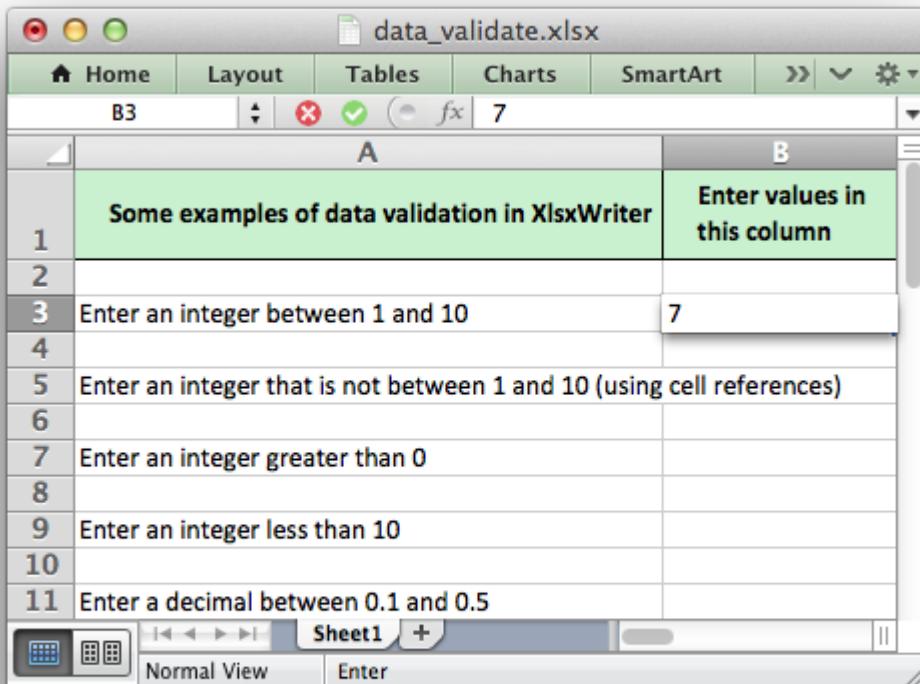
        Returns -1: Row or column is out of worksheet bounds.

        Returns -2: Incorrect parameter or option.

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*:

```
worksheet.data_validation(1, 1, {'validate': 'list',
                                 'source': ['open', 'high', 'close']})

worksheet.data_validation('B2', {'validate': 'list',
                                 'source': ['open', 'high', 'close']})
```

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',        {...})
```

7.24. worksheet.data_validation()

```
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*.

7.25 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.
>
> **Returns** 0: Success.
>
> **Returns** -1: Row or column is out of worksheet bounds.
>
> **Returns** -2: Incorrect parameter or option.
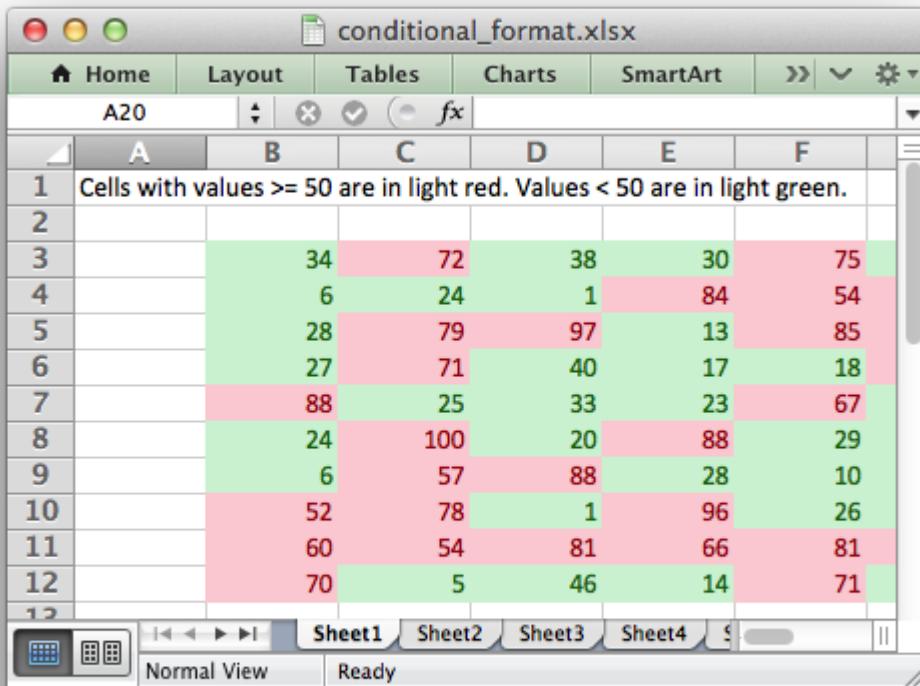
The `conditional_format()` method is used to add formatting to a cell or range of cells based on user defined criteria:

```
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*:

```
worksheet.conditional_format(0, 0, 2, 1, {'type':     'cell',
                                          'criteria': '>=',
                                          'value':    50,
                                          'format':   format1})

# This is equivalent to the following:
worksheet.conditional_format('A1:B3', {'type':     'cell',
                                       'criteria': '>=',
                                       'value':    50,
                                       'format':   format1})
```

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*.

7.26 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)

   Raises OverlappingRange if the range overlaps a previous merge or table range.

   Returns 0: Success.

   Returns -1: Row or column is out of worksheet bounds.

   Returns -2: Incorrect parameter or option.

   Returns -3: Not supported in `constant_memory` mode.

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*.

Both row-column and A1 style notation are supported. The following are equivalent:

```
worksheet.add_table(2, 1, 6, 5, { ... })
worksheet.add_table('B3:F7',    { ... })
```

See *Working with Cell Notation* for more details.

See also the examples in *Example: Worksheet Tables*.

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

7.27 worksheet.add_sparkline()

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

    Add sparklines to a worksheet.

   Parameters
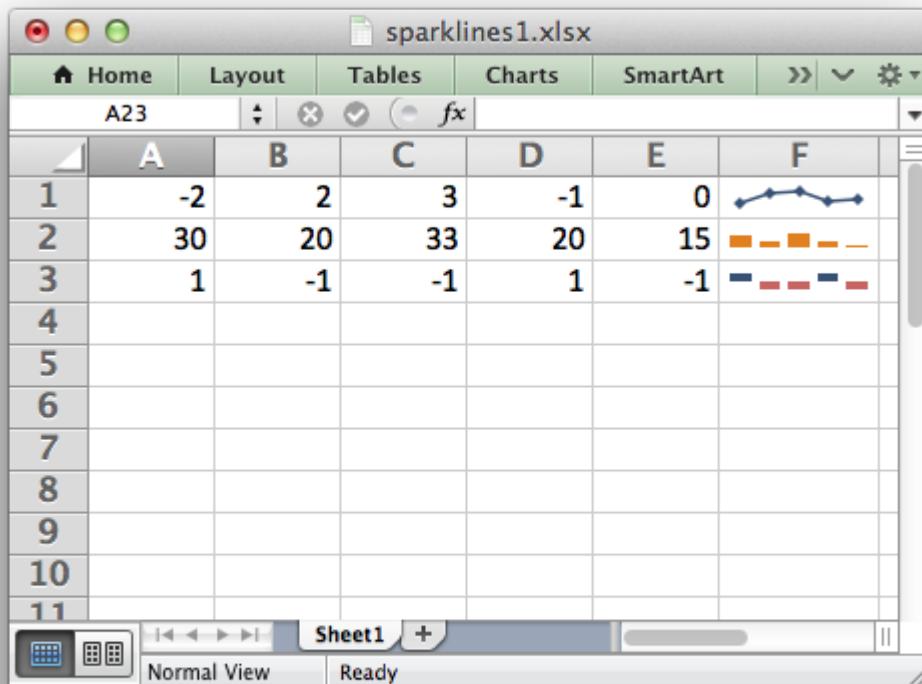
- **row** (*int*) – The cell row (zero indexed).

- **col** (*int*) – The cell column (zero indexed).

- **options** (*dict*) – Sparkline formatting options.

Returns 0: Success.

Returns -1: Row or column is out of worksheet bounds.

Returns -2: Incorrect parameter or option.

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'})
```

Both row-column and A1 style notation are supported. The following are equivalent:

```
worksheet.add_sparkline(0, 5, {'range': 'A1:E1'})
worksheet.add_sparkline('F1', {'range': 'A1:E1'})
```

See *Working with Cell Notation* for more details.

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.

7.28 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.
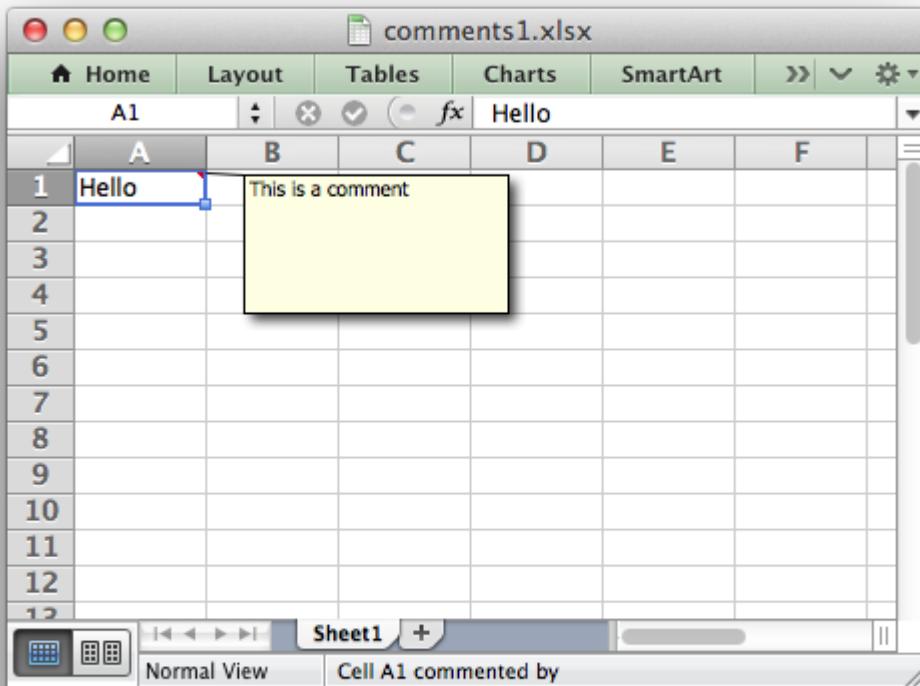
>   **Returns** 0: Success.

>   **Returns** -1: Row or column is out of worksheet bounds.

>   **Returns** -2: String longer than 32k characters.

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')
```

Both row-column and A1 style notation are supported. The following are equivalent:

```
worksheet.write_comment(0, 0, 'This is a comment')
worksheet.write_comment('A1', 'This is a comment')
```

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 behavior 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
font_name
font_size
```

```
start_cell
start_row
start_col
x_offset
y_offset
```

For more details see *Working with Cell Comments* and *Example: Adding Cell Comments to Work-sheets (Advanced)* .

7.29 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 Work-sheets (Advanced)* .

7.30 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 Work-sheets (Advanced)* .

7.31 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.

7.32 worksheet.activate()
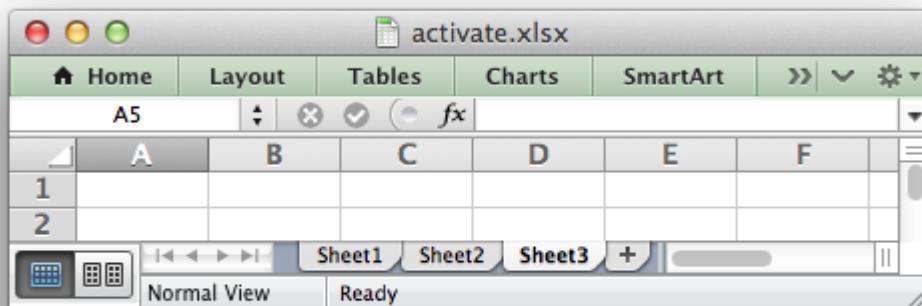
activate()

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.

7.33 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.

7.34 worksheet.hide()
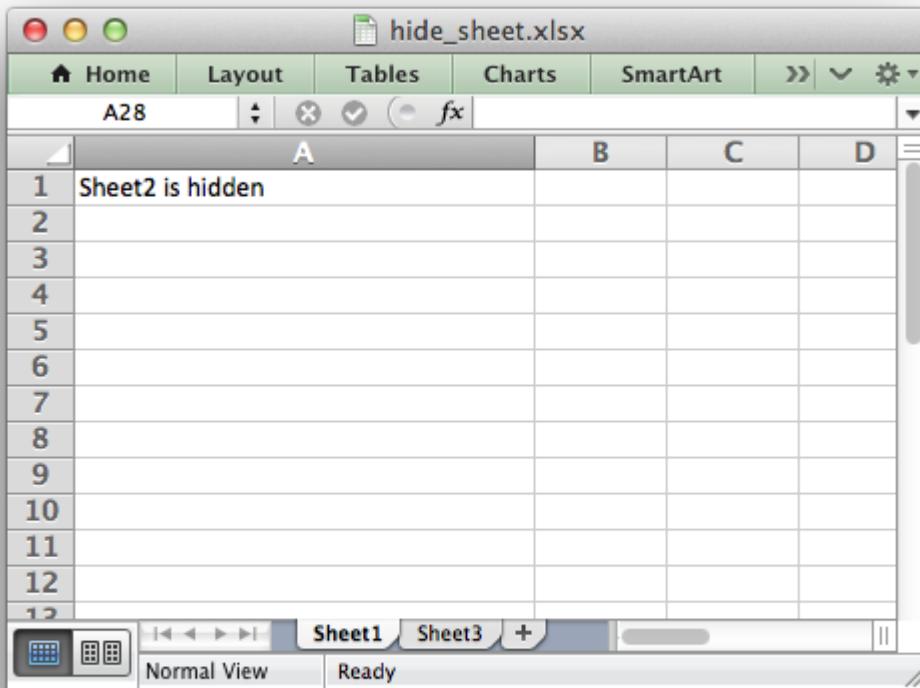
hide()
> Hide the current worksheet.

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.

7.35 worksheet.very_hidden()

very_hidden()
    Hide the current worksheet. Can only be unhidden by VBA.

The `very_hidden()` method can be used to hide a worksheet similar to the `hide()` method. The difference is that the worksheet cannot be unhidden in the the Excel user interface. The Excel worksheet "xlSheetVeryHidden" option can only be unset programmatically by VBA.

7.36 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.

7.37 worksheet.merge_range()

merge_range(*first_row*, *first_col*, *last_row*, *last_col*, *data*[, *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.
> >
> > **Raises OverlappingRange** if the range overlaps a previous merge or table range.
> >
> > **Returns** 0: Success.
> >
> > **Returns** -1: Row or column is out of worksheet bounds.
> >
> > **Returns** Other: Error return value of the called `write()` method.

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 behavior with XlsxWriter you need to apply a *Format*:

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

worksheet.merge_range('B3:D4', 'Merged Cells', merge_format)
```
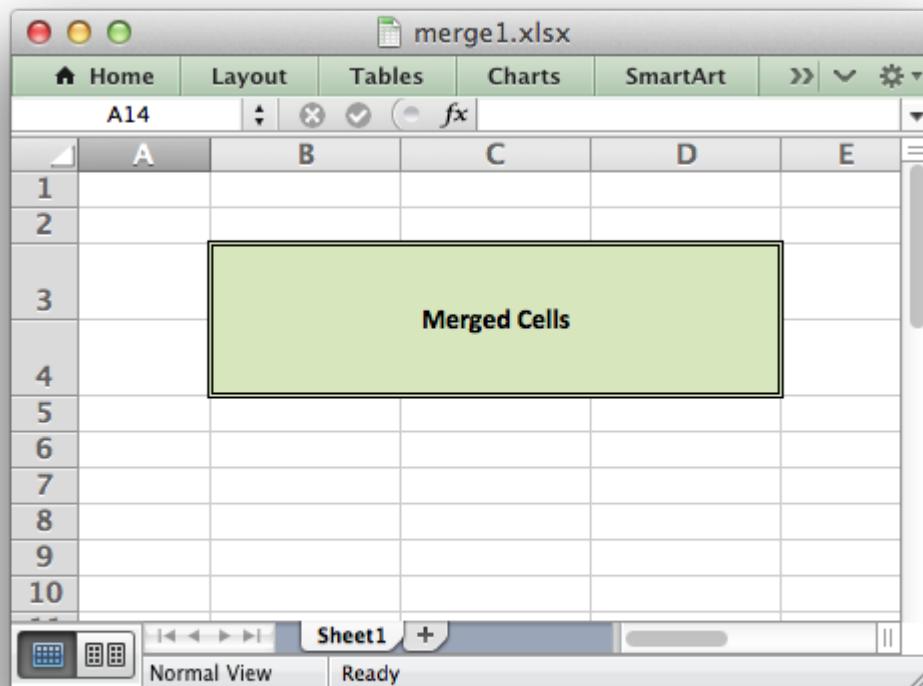
Both row-column and A1 style notation are supported. The following are equivalent:

```
worksheet.merge_range(2, 1, 3, 3, 'Merged Cells', merge_format)
worksheet.merge_range('B3:D4',    'Merged Cells', merge_format)
```

See *Working with Cell Notation* for more details.
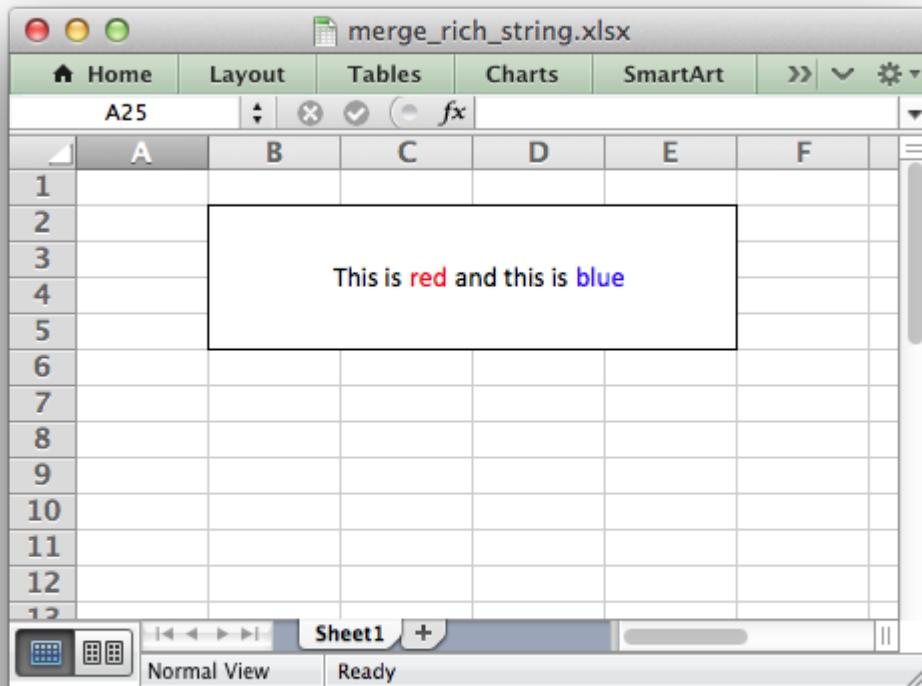
It is possible to apply other formatting to the merged cells as well:

```
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*.

Note: Merged ranges generally don't work in XlsxWriter when `Workbook()` 'constant_memory' mode is enabled.

7.38 worksheet.autofilter()
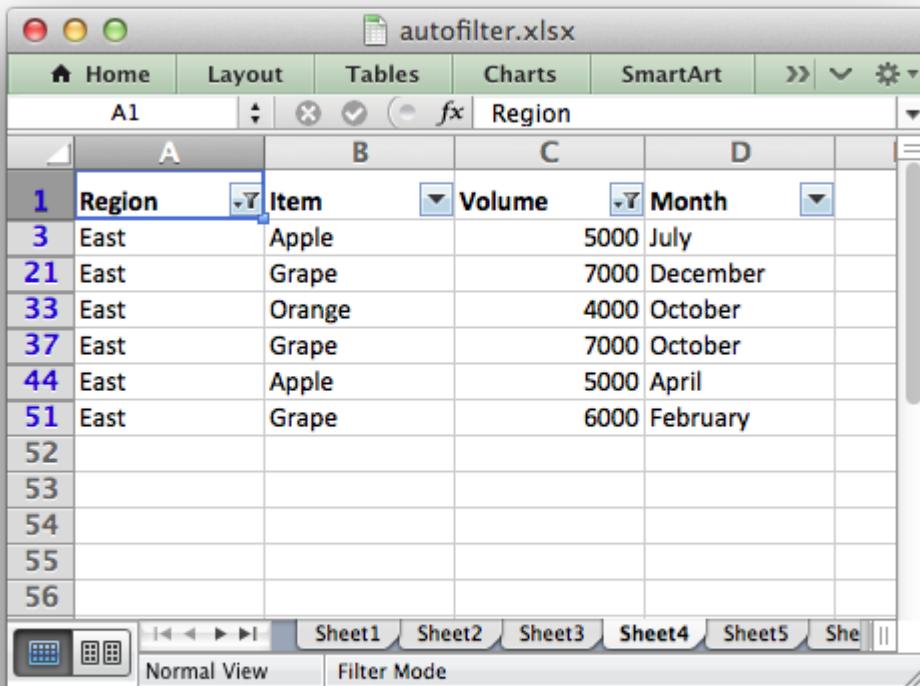
autofilter(*first_row*, *first_col*, *last_row*, *last_col*)
    Set the autofilter area in the worksheet.

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')
```

Both row-column and A1 style notation are supported. The following are equivalent:

```
worksheet.autofilter(0, 0, 10, 3)
worksheet.autofilter('A1:D11')
```

See *Working with Cell Notation* for more details.

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

See *Working with Autofilters* for more details.

7.39 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:

```
worksheet.filter_column(2,   'x > 2000')
worksheet.filter_column('C', 'x > 2000')
```

See *Working with Cell Notation* for more details.

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.

7.40 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:

```
worksheet.filter_column_list(2,   ['March', 'April', 'May'])
worksheet.filter_column_list('C', ['March', 'April', 'May'])
```

See *Working with Cell Notation* for more details.

One or more criteria can be selected:

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

To filter blanks as part of the list use *Blanks* as a list item:

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

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.

7.41 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'.

7.42 worksheet.set_top_left_cell()

set_top_left_cell(*row*, *col*)
    Set the first visible cell at the top left of a worksheet.
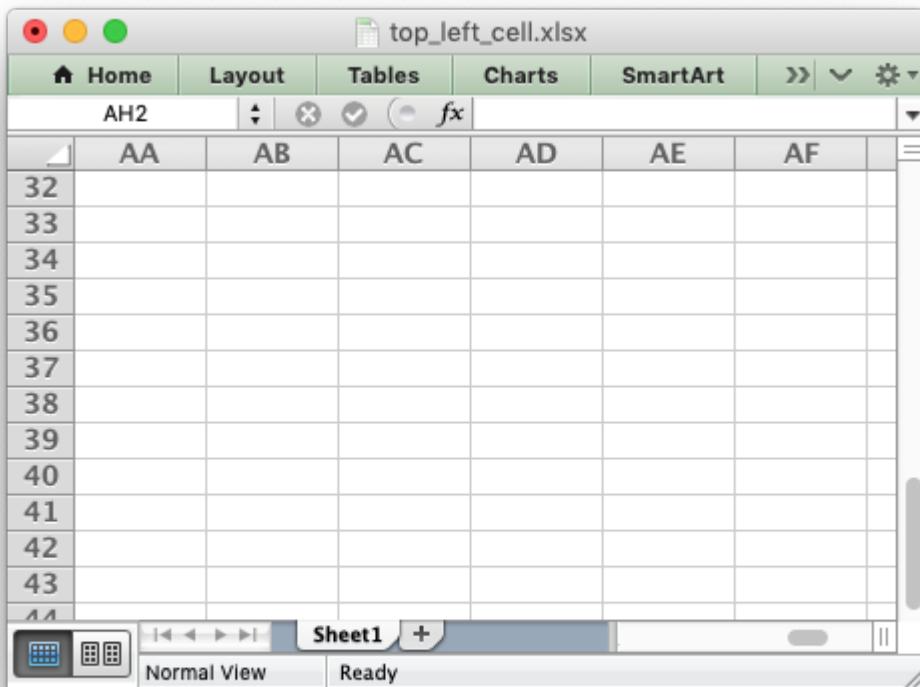
        Parameters

                • **row** (*int*) – The cell row (zero indexed).

                • **col** (*int*) – The cell column (zero indexed).

This set_top_left_cell method can be used to set the top leftmost visible cell in the worksheet:

```
worksheet.set_top_left_cell(31, 26)

# Same as:
worksheet.set_top_left_cell('AA32')
```

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

7.43 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.

The `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.
```

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

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.

7.44 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.

7.45 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()`.

7.46 worksheet.right_to_left()
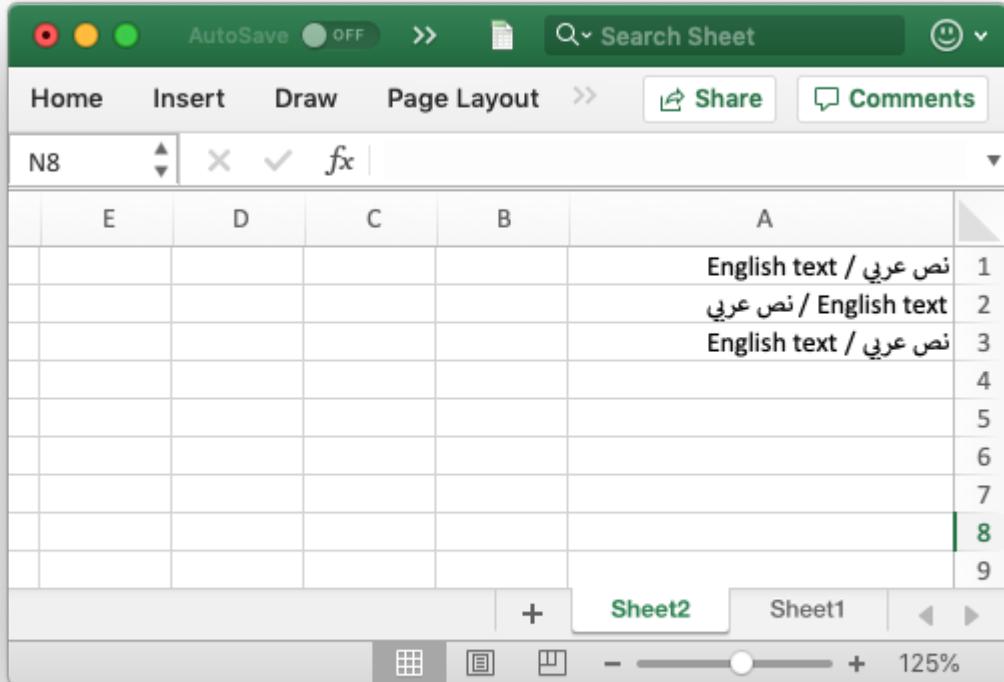
right_to_left()
    Display the worksheet cells from right to left for some versions of Excel.

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.

See also the Format `set_reading_order()` property to set the direction of the text within cells and the *Example: Left to Right worksheets and text* example program.

7.47 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()

7.48 worksheet.set_background()

set_background(*filename*[, *is_byte_stream*])
    Set the background image for a worksheet.
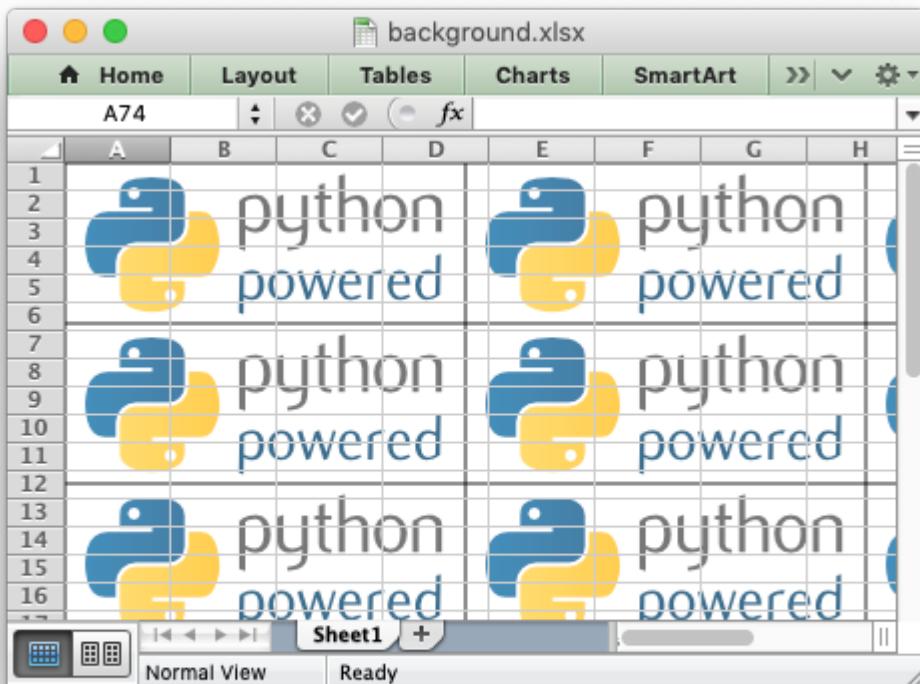
        Parameters

            • **filename** (*str*) – The image file (or byte stream).

            • **is_byte_stream** (*bool*) – The file is a stream of bytes.

The `set_background()` method can be used to set the background image for the worksheet:

```
worksheet.set_background('logo.png')
```


The `set_background()` method supports all the image formats supported by `insert_image()`.

Some people use this method to add a watermark background to their document. However, Microsoft recommends using a header image to set a watermark. The choice of method depends on whether you want the watermark to be visible in normal viewing mode or just when the file is printed. In XlsxWriter you can get the header watermark effect using `set_header()`:

```
worksheet.set_header('&C&G', {'image_center': 'watermark.png'})
```

It is also possible to pass an in-memory byte stream to `set_background()` if the `is_byte_stream` parameter is set to True. The stream should be `io.BytesIO`:

```
worksheet.set_background(io_bytes, is_byte_stream=True)
```

See *Example: Setting the Worksheet Background* for an example.

7.49 worksheet.set_tab_color()

set_tab_color()
>   Set the color of the worksheet tab.

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

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

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

The color can be a Html style #RRGGBB string or a limited number named colors, see *Working with Colors*.

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

7.50 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')
```

The password should be an ASCII string. Passing the empty string `''` is the same as turning on protection without a password. See the note below on the "password" strength.

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('abc123', {'insert_rows': True})
```

For chartsheets the allowable options and default values are:

```
    options = {
        'objects':             True,
        'content':             True,
    }
```

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. However, it is possible to encrypt an XlsxWriter file using a third party open source tool called msoffice-crypt. This works for macOS, Linux and Windows:

```
    msoffice-crypt.exe -e -p password clear.xlsx encrypted.xlsx
```

7.51 worksheet.unprotect_range()

unprotect_range(*cell_range*, *range_name*)
    Unprotect ranges within a protected worksheet.

        Parameters

                • **cell_range** (*string*) – The cell or cell range to unprotect.

                • **range_name** (*string*) – An name for the range.

The unprotect_range() method is used to unprotect ranges in a protected worksheet. It can be used to set a single range or multiple ranges:

```
    worksheet.unprotect_range('A1')
    worksheet.unprotect_range('C1')
    worksheet.unprotect_range('E1:E3')
    worksheet.unprotect_range('G1:K100')
```

As in Excel the ranges are given sequential names like Range1 and Range2 but a user defined name can also be specified:

```
worksheet.unprotect_range('G4:I6', 'MyRange')
```

7.52 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 optimization 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.

7.53 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.

7.54 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.

7.55 worksheet.ignore_errors()

ignore_errors(*options*)
    Ignore various Excel errors/warnings in a worksheet for user defined ranges.
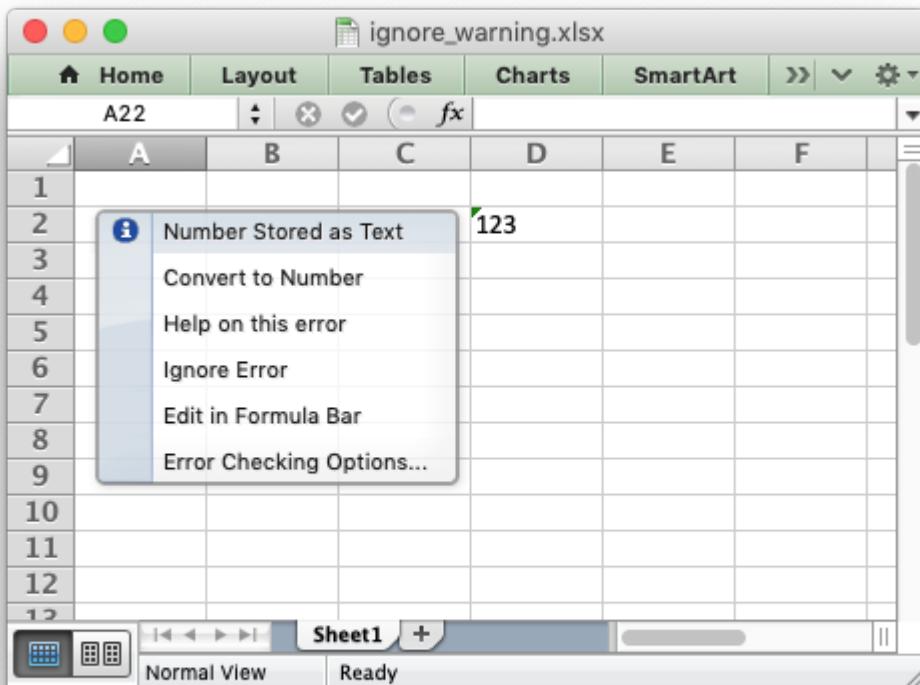
        Returns 0: Success.

        Returns -1: Incorrect parameter or option.

The ignore_errors() method can be used to ignore various worksheet cell errors/warnings. For example the following code writes a string that looks like a number:

```
        worksheet.write_string('D2', '123')
```

This causes Excel to display a small green triangle in the top left hand corner of the cell to indicate an error/warning:

Sometimes these warnings are useful indicators that there is an issue in the spreadsheet but sometimes it is preferable to turn them off. Warnings can be turned off at the Excel level for all workbooks and worksheets by using the using "Excel options -> Formulas -> Error checking rules". Alternatively you can turn them off for individual cells in a worksheet, or ranges of cells, using the `ignore_errors()` method with a dict of options and ranges like this:

```python
worksheet.ignore_errors({'number_stored_as_text': 'A1:H50'})

# Or for more than one option:
worksheet.ignore_errors({'number_stored_as_text': 'A1:H50',
                         'eval_error':            'A1:H50'})
```

The range can be a single cell, a range of cells, or multiple cells and ranges separated by spaces:

```python
# Single cell.
worksheet.ignore_errors({'eval_error': 'C6'})

# Or a single range:
worksheet.ignore_errors({'eval_error': 'C6:G8'})

# Or multiple cells and ranges:
worksheet.ignore_errors({'eval_error': 'C6 E6 G1:G20 J2:J6'})
```

Note: calling `ignore_errors()` multiple times will overwrite the previous settings.

You can turn off warnings for an entire column by specifying the range from the first cell in the column to the last cell in the column:

```
worksheet.ignore_errors({'number_stored_as_text': 'A1:A1048576'})
```

Or for the entire worksheet by specifying the range from the first cell in the worksheet to the last cell in the worksheet:

```
worksheet.ignore_errors({'number_stored_as_text': 'A1:XFD1048576'})
```

The worksheet errors/warnings that can be ignored are:

- `number_stored_as_text`: Turn off errors/warnings for numbers stores as text.
- `eval_error`: Turn off errors/warnings for formula errors (such as divide by zero).
- `formula_differs`: Turn off errors/warnings for formulas that differ from surrounding formulas.
- `formula_range`: Turn off errors/warnings for formulas that omit cells in a range.
- `formula_unlocked`: Turn off errors/warnings for unlocked cells that contain formulas.
- `empty_cell_reference`: Turn off errors/warnings for formulas that refer to empty cells.
- `list_data_validation`: Turn off errors/warnings for cells in a table that do not comply with applicable data validation rules.
- `calculated_column`: Turn off errors/warnings for cell formulas that differ from the column formula.
- `two_digit_text_year`: Turn off errors/warnings for formulas that contain a two digit text representation of a year.

See also *Example: Ignoring Worksheet errors and warnings*.

THE WORKSHEET CLASS (PAGE SETUP)

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

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(*view=1*)
> Set the page view mode.

> > **Parameters view** (*int*) – 0: Normal, 1: Page Layout, 2: Page Break.

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

```
worksheet.set_page_view()
```

It can also be used to set the other view modes:

- 0: Normal view mode.

- 1: Page view mode (the default).

- 2: Page break view mode. Same as `set_pagebreak_view()`.

8.4 worksheet.set_pagebreak_view()

`set_pagebreak_view`()
    Set the page break view mode.

This method is used to display the worksheet in "Page Break Preview" mode:

```
worksheet.set_pagebreak_view()
```

8.5 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
Continued on next page		

Table 8.1 – continued from previous page

Index	Paper format	Paper size
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
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:

```python
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.

8.6 worksheet.center_horizontally()

center_horizontally()

> Center the printed page horizontally.

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

```python
worksheet.center_horizontally()
```

8.7 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()
```

8.8 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.

8.9 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]
&&	Misc.	Literal ampersand "&"

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

For example:

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

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

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

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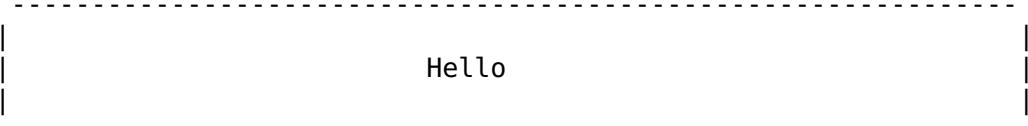
```
$worksheet->set_header('&RHello');
```

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

For simple text, if you do not specify any justification the text will be centered. 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')
```
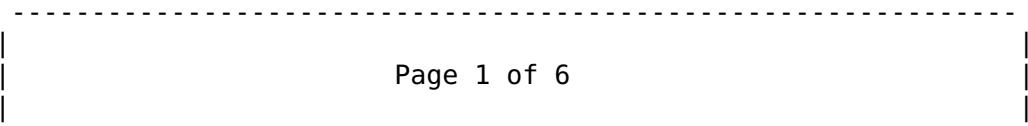
```
-----------------------------------------------------------------
|                                                               |
| Ciao                        Bello                       Cielo |
|                                                               |
```
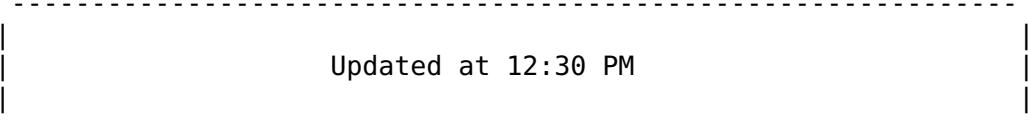
The information control characters act as variables that Excel will update as the workbook or worksheet changes. Times and dates are in the users default format:

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

```
-----------------------------------------------------------------
|                                                               |
|                          Page 1 of 6                          |
|                                                               |
```
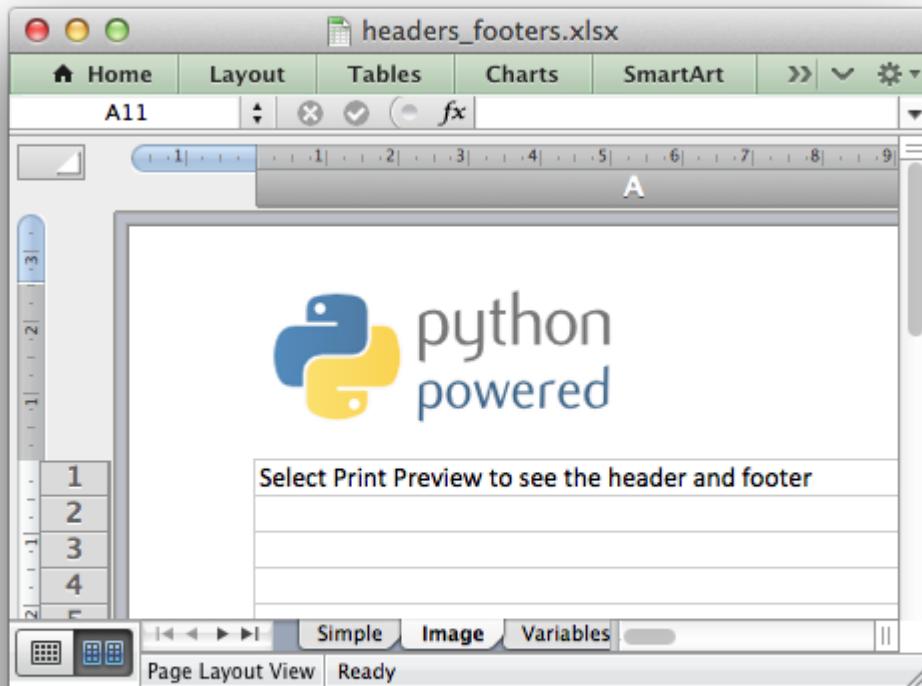
```
worksheet.set_header('&CUpdated at &T')
```

```
-----------------------------------------------------------------
|                                                               |
|                       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" | sed '

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

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 options 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 a warning will be issued.

See also *Example: Adding Headers and Footers to Worksheets*.

8.10 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()`.

8.11 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.
```

8.12 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.
```

8.13 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.

8.14 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()
```
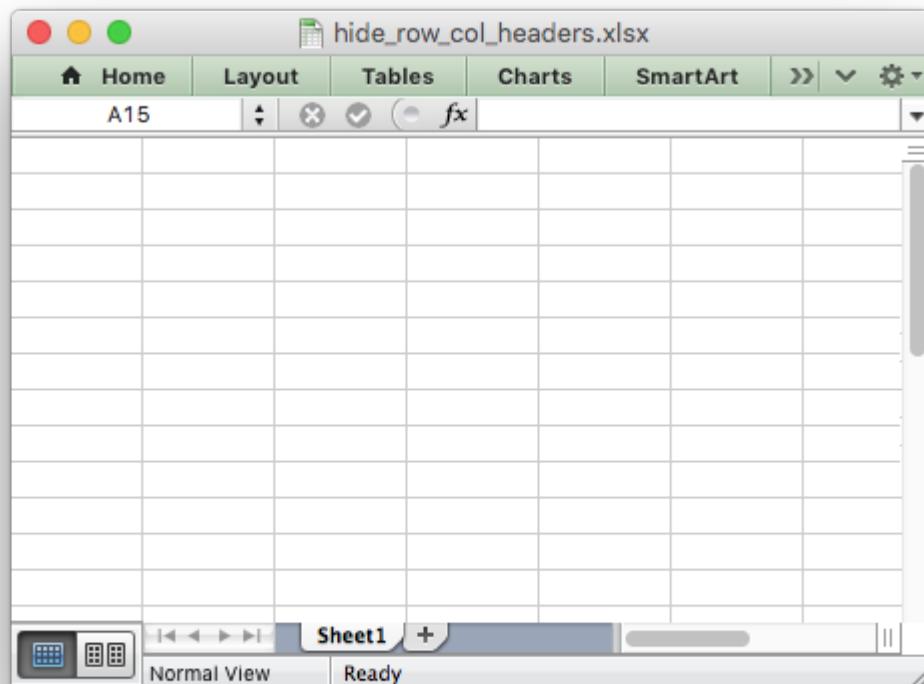
8.15 worksheet.hide_row_col_headers()

hide_row_col_headers()
Set the option to hide the row and column headers in a worksheet.

This method is similar to the `print_row_col_headers()` except that it hides the row and column headers on the worksheet:

```
worksheet.hide_row_col_headers()
```


8.16 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.
>
> **Returns** 0: Success.
>
> **Returns** -1: Row or column is out of worksheet bounds.

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.
```

8.17 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]
```

8.18 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 allows 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".

8.19 worksheet.set_start_page()

set_start_page()
> Set the start/first page number when printing.
>
> > **Parameters start_page** (*int*) – Starting page number.

The set_start_page() method is used to set the page number of the starting page when the worksheet is printed out. It is the same as the "First Page Number" option in Excel:

```
# Start print from page 2.
worksheet.set_start_page(2)
```

8.20 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.

8.21 worksheet.print_black_and_white()

print_black_and_white()
> Set the worksheet to print in black and white.

Set the option to print the worksheet in black and white:

```
worksheet.print_black_and_white()
```

8.22 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.

8.23 worksheet.set_v_pagebreaks()

set_v_pagebreaks(*breaks*)
> Set the vertical page breaks on a worksheet.
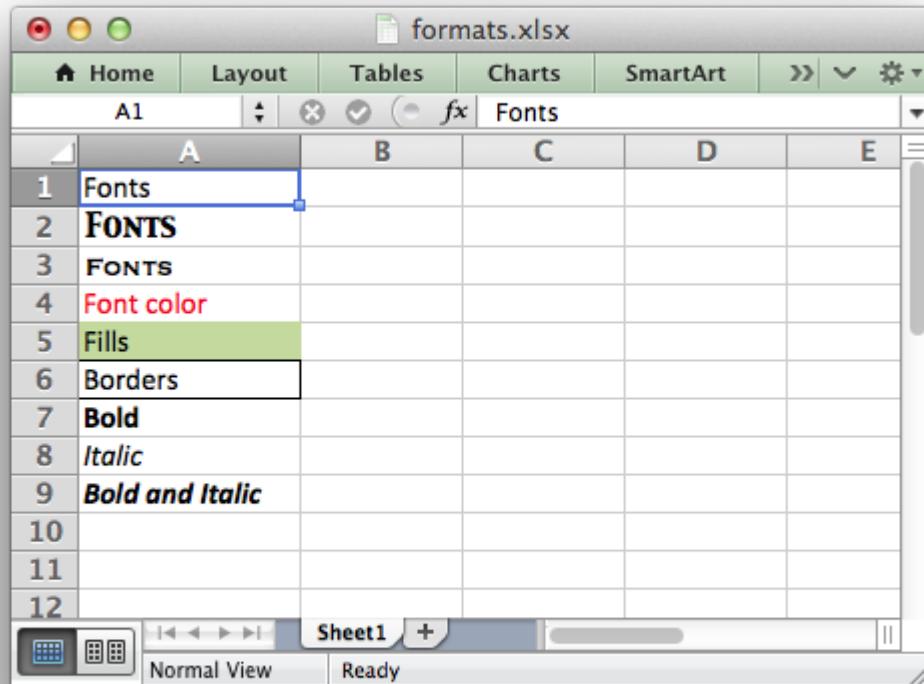
> > **Parameters breaks** (*list*) – List of page break columns.

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

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, colors, patterns, borders, alignment and number formatting.

9.1 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:

```
cell_format1 = workbook.add_format()        # Set properties later.
cell_format2 = workbook.add_format(props)   # Set properties at creation.
```

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:

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

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

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

In general the key/value interface is more flexible and clearer than the object method and is the recommended method for setting format properties. However, both methods produce the same result.

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', cell_format)
worksheet.write_string(1, 0, 'Bar', cell_format)
worksheet.write_number(2, 0, 3,     cell_format)
worksheet.write_blank (3, 0, '',    cell_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, cell_format)
worksheet.set_column('A:D', 20, cell_format)
```

9.2 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:

```
cell_format = workbook.add_format()

cell_format.set_bold()       # Turns bold on.
cell_format.set_bold(True)   # 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:

```
cell_format.set_bold(False)  # Turns bold off.
```

9.3 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:

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

# Later...
cell_format.set_font_color('green')
worksheet.write('B1', 'Cell B1', cell_format)
```
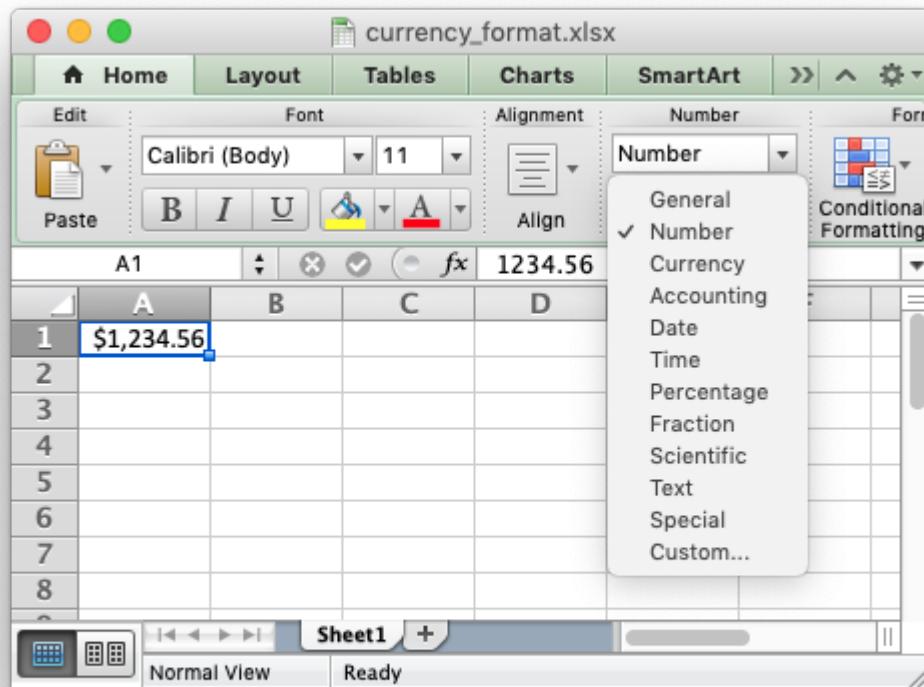
Cell A1 is assigned a format which initially has the font set to the color red. However, the color is subsequently set to green. When Excel displays Cell A1 it will display the final state of the Format which in this case will be the color green.

9.4 Number Format Categories
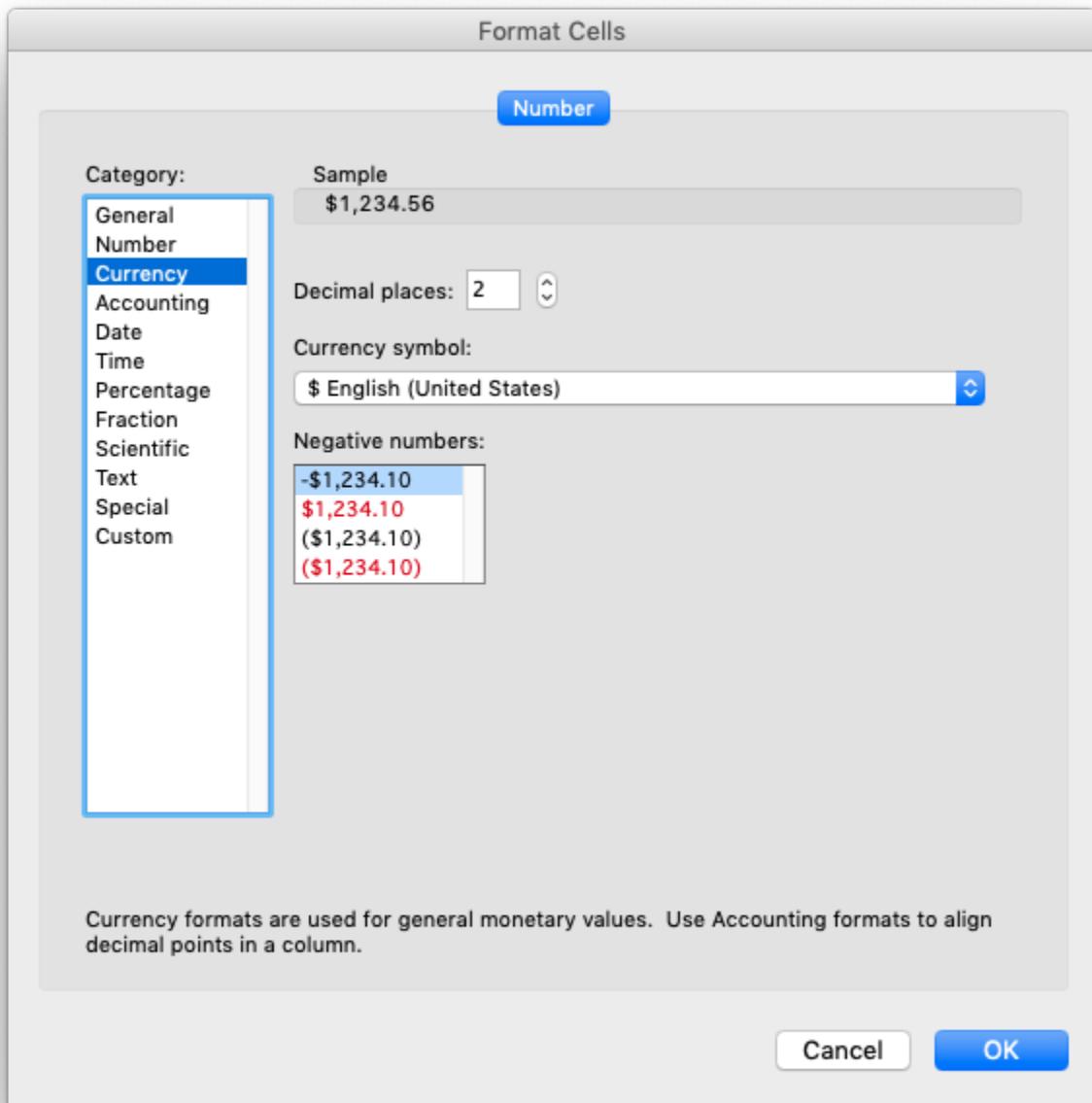
The `set_num_format()` method, shown below, is used to set the number format for numbers:

```python
import xlsxwriter

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

currency_format = workbook.add_format({'num_format': '$#,##0.00'})
worksheet.write('A1', 1234.56, currency_format)

workbook.close()
```
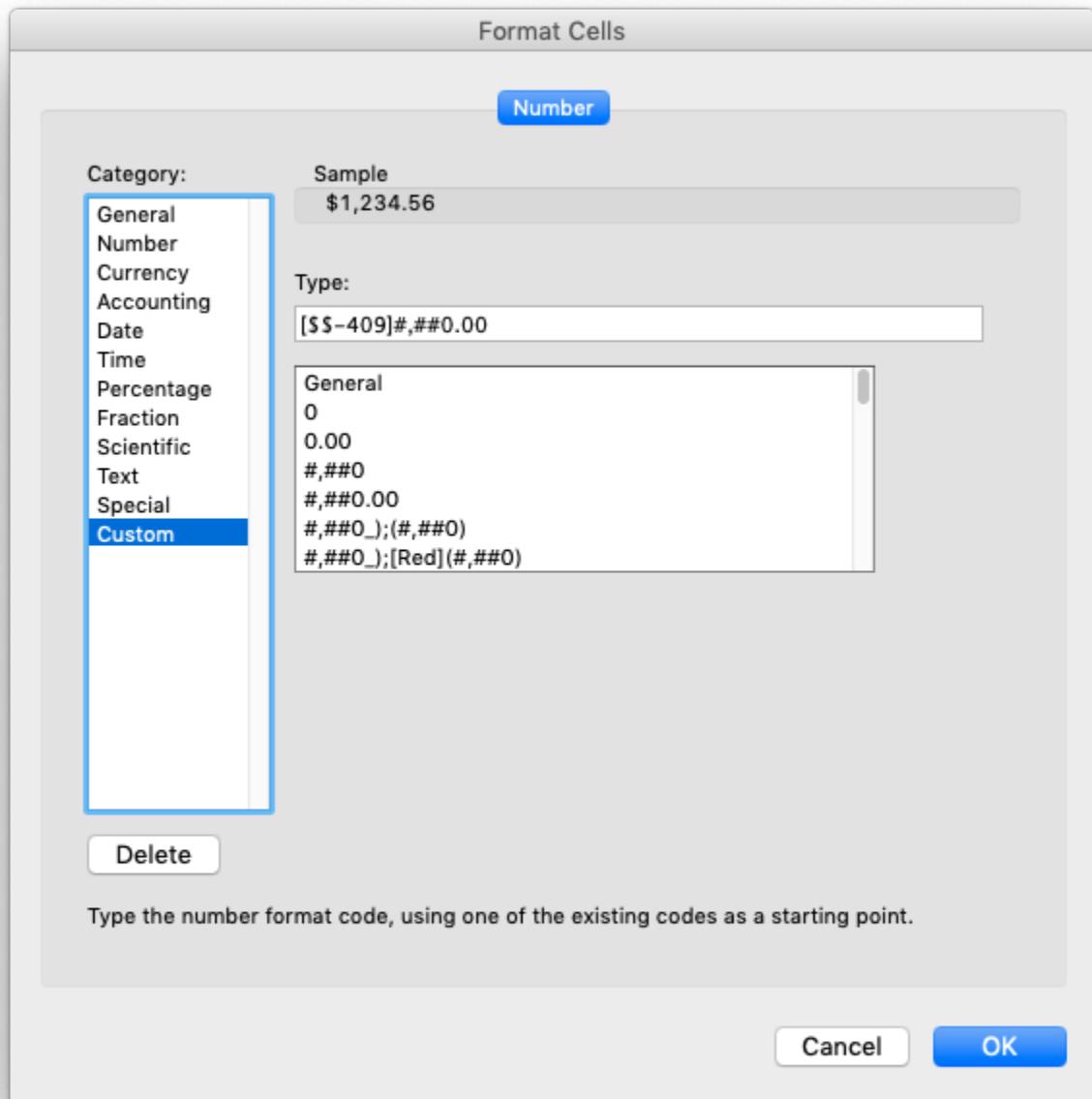
If the number format you use is the same as one of Excel's built in number formats then it will have a number category such as General, Number, Currency, Accounting, Date, Time, Percentage, Fraction, Scientific, Text, Special or Custom. In the case of the example above the formatted output shows up as a Number category:

If we wanted it to have a different category, such as Currency, then we would have to match the number format string with the number format used by Excel. The easiest way to do this is to open the Number Formatting dialog in Excel and set the format that you want:

Then, while still in the dialog, change to Custom. The format displayed is the format used by Excel.

If we put the format that we found (`'[$$-409]#,##0.00'`) into our previous example and rerun it we will get a number format in the Currency category:

```python
import xlsxwriter

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

currency_format = workbook.add_format({'num_format': '[$$-409]#,##0.00'})
worksheet.write('A1', 1234.56, currency_format)
```
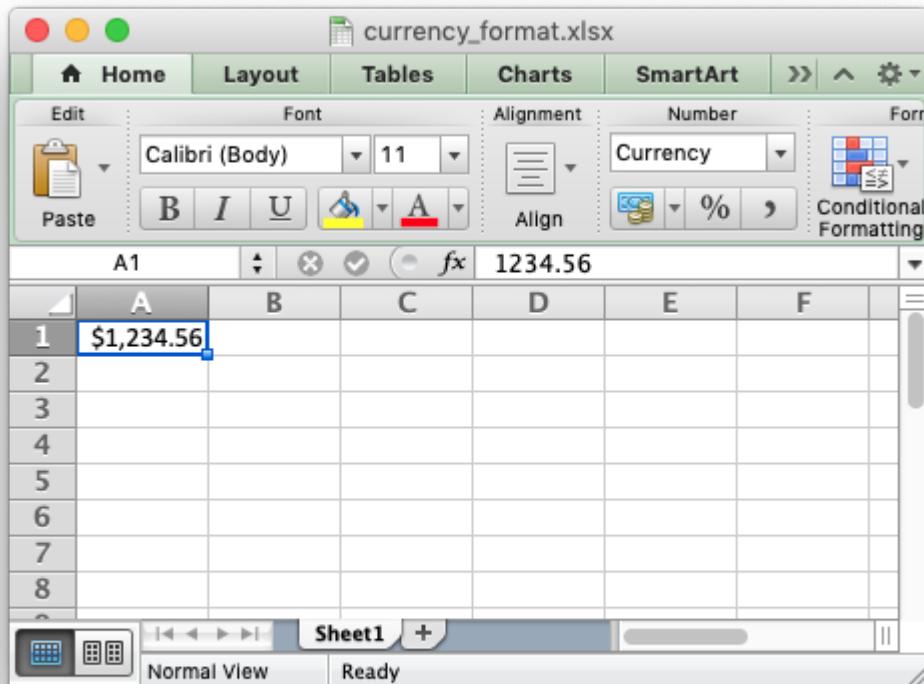
```
    workbook.close()
```

Here is the output:

The same process can be used to find format strings for Date or Accountancy formats. However, you also need to be aware of the OS settings Excel uses for number separators such as the "grouping/thousands" separator and the "decimal" point. See the next section for details.

9.5 Number Formats in different locales

As shown in the previous section the `set_num_format()` method is used to set the number format for Xlsxwriter formats. A common use case is to set a number format with a "grouping/thousands" separator and a "decimal" point:
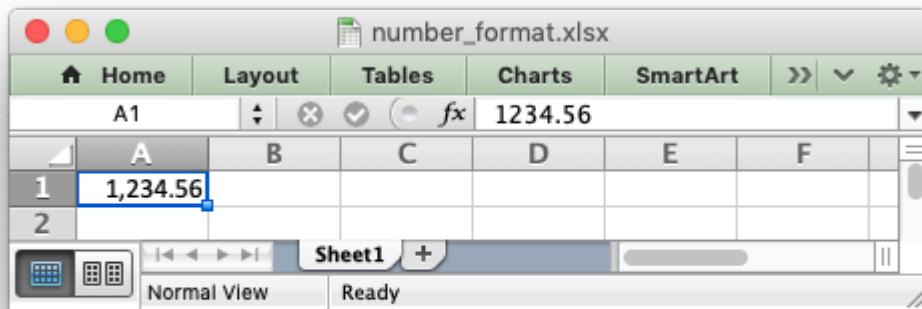
```python
import xlsxwriter

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

number_format = workbook.add_format({'num_format': '#,##0.00'})
worksheet.write('A1', 1234.56, number_format)

workbook.close()
```
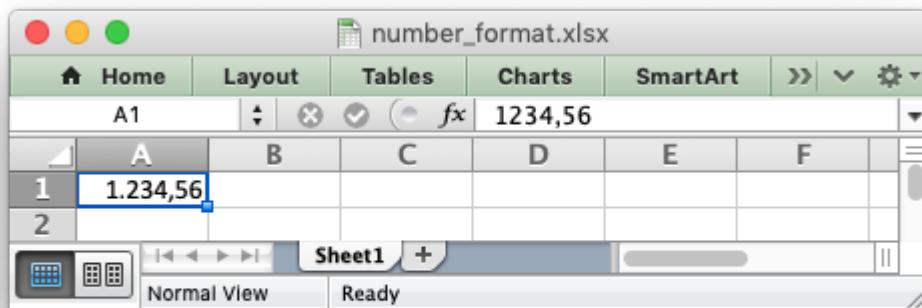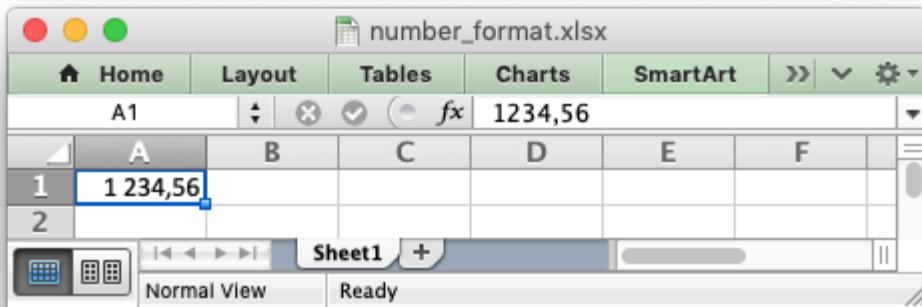
In the US locale (and some others) where the number "grouping/thousands" separator is "," and the "decimal" point is "." this would be shown in Excel as:

In other locales these values may be reversed or different. They are generally set in the "Region" settings of Windows or Mac OS. Excel handles this by storing the number format in the file format in the US locale, in this case `#,##0.00`, but renders it according to the regional settings of the host OS. For example, here is the same, unmodified, output file shown above in a German locale:

And here is the same file in a Russian locale. Note the use of a space as the "grouping/thousands" separator:

In order to replicate Excel's behavior all XlsxWriter programs should use US locale formatting which will then be rendered in the settings of your host OS.

9.6 Format methods and Format properties

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

Category	Description	Property	Method Name
Font	Font type	'font_name'	set_font_name()
	Font size	'font_size'	set_font_size()
	Font color	'font_color'	set_font_color()
	Bold	'bold'	set_bold()
	Italic	'italic'	set_italic()
	Underline	'underline'	set_underline()
	Strikeout	'font_strikeout'	set_font_strikeout()
	Super/Subscript	'font_script'	set_font_script()
Number	Numeric format	'num_format'	set_num_format()
Protection	Lock cells	'locked'	set_locked()
	Hide formulas	'hidden'	set_hidden()
Alignment	Horizontal align	'align'	set_align()
	Vertical align	'valign'	set_align()
	Rotation	'rotation'	set_rotation()
	Text wrap	'text_wrap'	set_text_wrap()
	Reading order	'reading_order'	set_reading_order()
	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()
			Continued on next page

Table 9.1 – continued from previous page

Category	Description	Property	Method Name
	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()

The format properties and methods are explained in the following sections.

9.7 format.set_font_name()
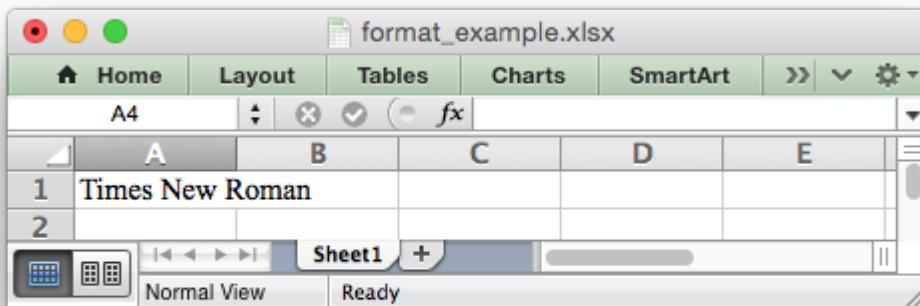
set_font_name(*fontname*)
    Set the font used in the cell.

        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'.

9.8 format.set_font_size()
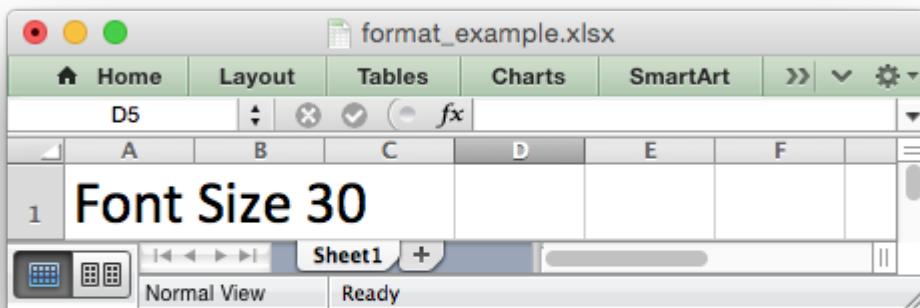
set_font_size(*size*)
>   Set the size of the font used in the cell.

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

Set the font size of the cell format:

```
cell_format = workbook.add_format()
cell_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.

9.9 format.set_font_color()
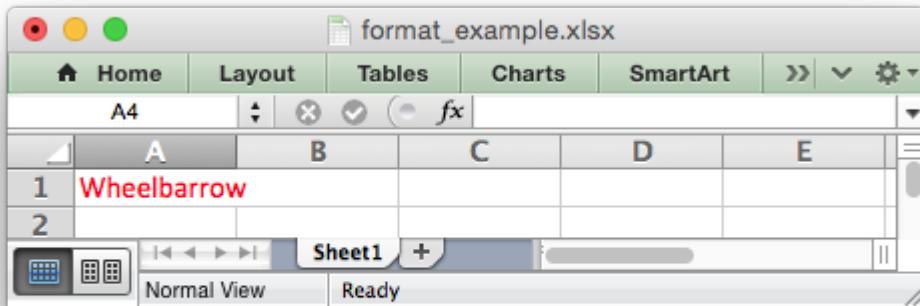
set_font_color(*color*)
>   Set the color of the font used in the cell.

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

Set the font color:

```
cell_format = workbook.add_format()

cell_format.set_font_color('red')

worksheet.write(0, 0, 'Wheelbarrow', cell_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 color of the font in a cell. To set the color of a cell use the set_bg_color() and set_pattern() methods.
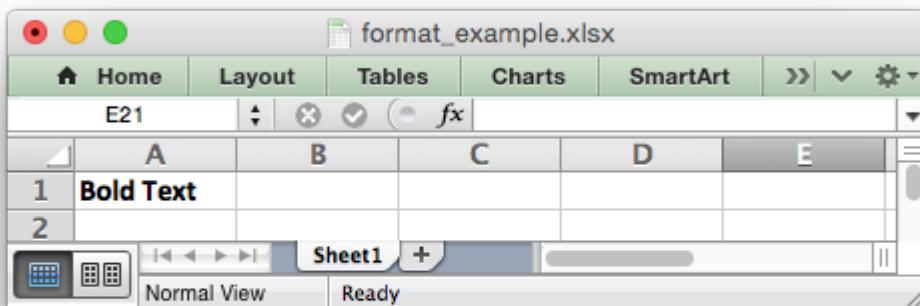
9.10 format.set_bold()

set_bold()
    Turn on bold for the format font.

Set the bold property of the font:

```
cell_format.set_bold()
```
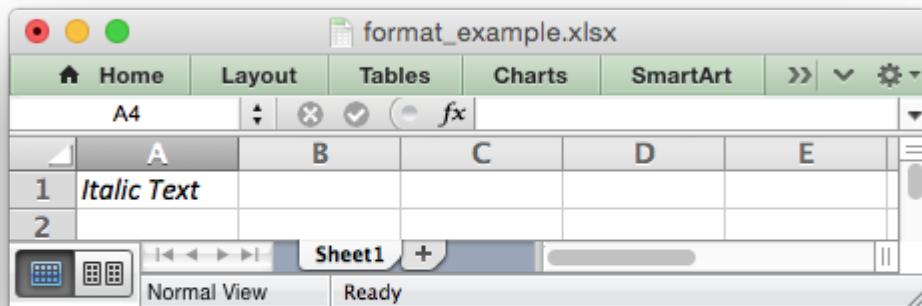
9.11 format.set_italic()

set_italic()
> Turn on italic for the format font.

Set the italic property of the font:

    cell_format.set_italic()

9.12 format.set_underline()
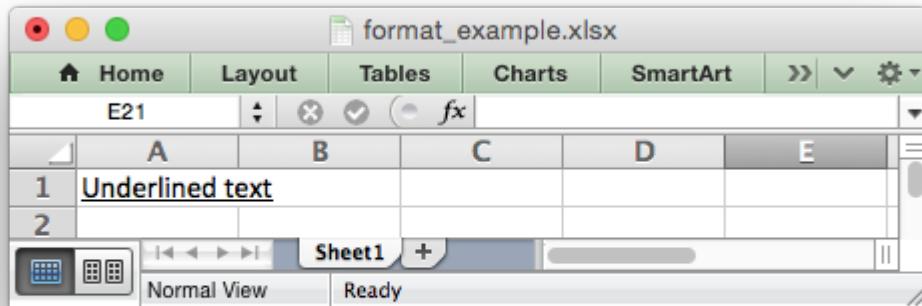
set_underline()
> Turn on underline for the format.

> > **Parameters style** (*int*) – Underline style.

Set the underline property of the format:

    cell_format.set_underline()
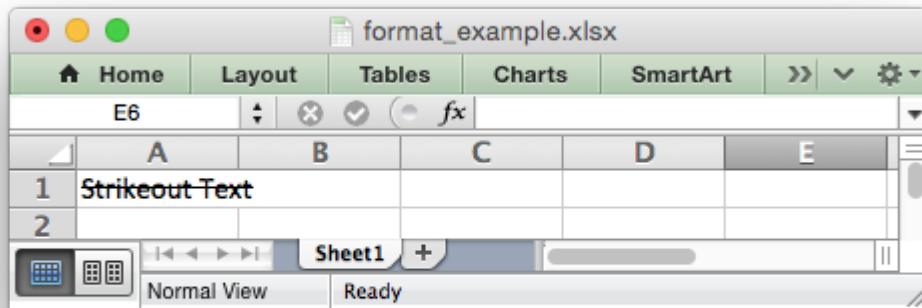
The available underline styles are:

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

9.13 format.set_font_strikeout()

`set_font_strikeout()`
     Set the strikeout property of the font.
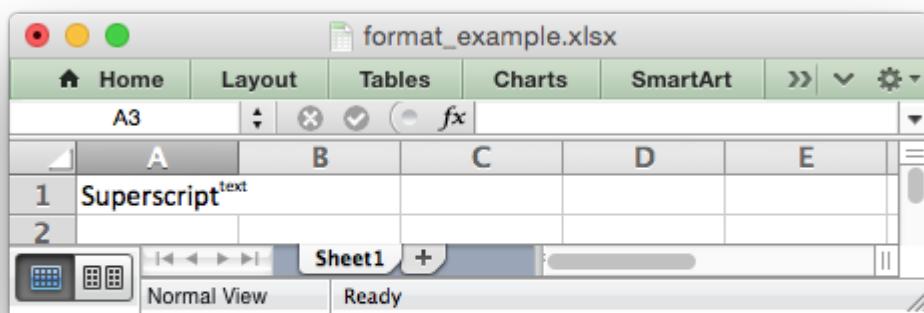
9.14 format.set_font_script()

set_font_script()
> Set the superscript/subscript property of the font.

The available options are:

- 1 = Superscript
- 2 = Subscript

This property is generally only useful when used in conjunction with `write_rich_string()`.

9.15 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:

```
cell_format1 = workbook.add_format()
cell_format2 = workbook.add_format()

cell_format1.set_num_format('d mmm yyyy')  # Format string.
cell_format2.set_num_format(0x0F)          # Format index.
```

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

```
    cell_format01.set_num_format('0.000')
    worksheet.write(1, 0, 3.1415926, cell_format01)        # -> 3.142

    cell_format02.set_num_format('#,##0')
    worksheet.write(2, 0, 1234.56, cell_format02)          # -> 1,235

    cell_format03.set_num_format('#,##0.00')
    worksheet.write(3, 0, 1234.56, cell_format03)          # -> 1,234.56

    cell_format04.set_num_format('0.00')
    worksheet.write(4, 0, 49.99, cell_format04)            # -> 49.99

    cell_format05.set_num_format('mm/dd/yy')
    worksheet.write(5, 0, 36892.521, cell_format05)        # -> 01/01/01

    cell_format06.set_num_format('mmm d yyyy')
    worksheet.write(6, 0, 36892.521, cell_format06)        # -> Jan 1 2001

    cell_format07.set_num_format('d mmmm yyyy')
    worksheet.write(7, 0, 36892.521, cell_format07)        # -> 1 January 2001

    cell_format08.set_num_format('dd/mm/yyyy hh:mm AM/PM')
    worksheet.write(8, 0, 36892.521, cell_format08)        # -> 01/01/2001 12:30 AM

    cell_format09.set_num_format('0 "dollar and" .00 "cents"')
    worksheet.write(9, 0, 1.87, cell_format09)             # -> 1 dollar and .87 cents

    # Conditional numerical formatting.
    cell_format10.set_num_format('[Green]General;[Red]-General;General')
    worksheet.write(10, 0, 123, cell_format10)  # > 0 Green
    worksheet.write(11, 0, -45, cell_format10)  # < 0 Red
    worksheet.write(12, 0,   0, cell_format10)  # = 0 Default color

    # Zip code.
    cell_format11.set_num_format('00000')
    worksheet.write(13, 0, 1209, cell_format11)
```
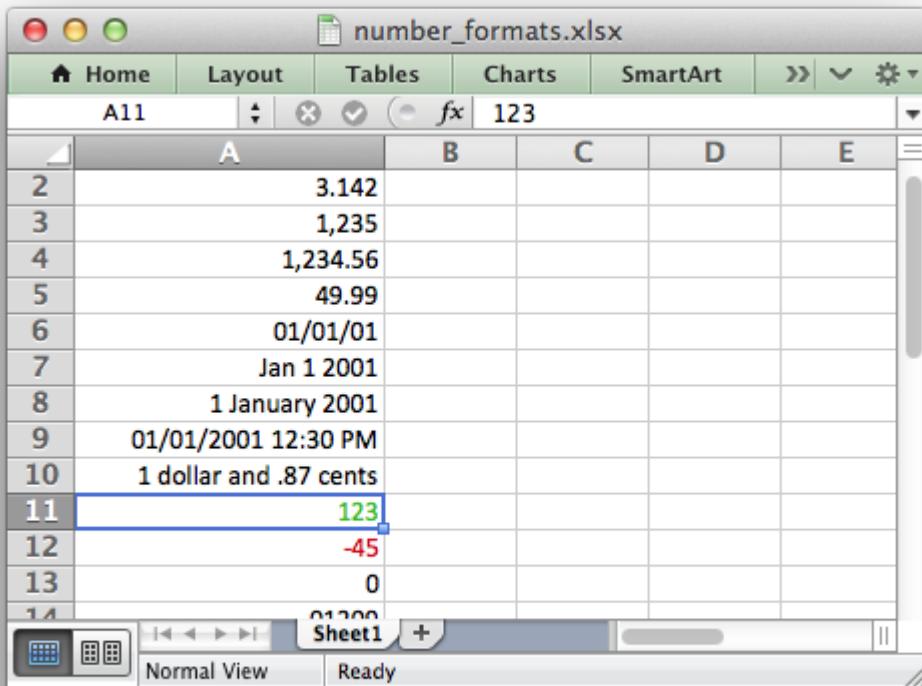
The number system used for dates is described in *Working with Dates and Time*.

The color 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.

For information on how to get a number format to show up as one of the number format categories such as Currency, Accounting, Date, Time, Percentage, Fraction, Scientific or Text, see *Number Format Categories*, above.

For backwards compatibility XlsxWriter also supports Excel's built-in formats which are set via an index number, rather than a string:

```
cell_format.set_num_format(3)   # Same as #,##0
```

The format indexes and the equivalent strings are shown in the following table:

Index	Format String
0	General
1	0
2	0.00
	Continued on next page

Table 9.2 – continued from previous page

Index	Format String
3	`#,##0`
4	`#,##0.00`
5	`($#,##0_);($#,##0)`
6	`($#,##0_);[Red]($#,##0)`
7	`($#,##0.00_);($#,##0.00)`
8	`($#,##0.00_);[Red]($#,##0.00)`
9	`0%`
10	`0.00%`
11	`0.00E+00`
12	`# ?/?`
13	`# ??/??`
14	`m/d/yy`
15	`d-mmm-yy`
16	`d-mmm`
17	`mmm-yy`
18	`h:mm AM/PM`
19	`h:mm:ss AM/PM`
20	`h:mm`
21	`h:mm:ss`
22	`m/d/yy h:mm`
...	...
37	`(#,##0_);(#,##0)`
38	`(#,##0_);[Red](#,##0)`
39	`(#,##0.00_);(#,##0.00)`
40	`(#,##0.00_);[Red](#,##0.00)`
41	`_(* #,##0_);_(* (#,##0);_(* "-"_);_(@_)`
42	`_($* #,##0_);_($* (#,##0);_($* "-"_);_(@_)`
43	`_(* #,##0.00_);_(* (#,##0.00);_(* "-"??_);_(@_)`
44	`_($* #,##0.00_);_($* (#,##0.00);_($* "-"??_);_(@_)`
45	`mm:ss`
46	`[h]:mm:ss`
47	`mm:ss.0`
48	`##0.0E+0`
49	`@`

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.

The dollar sign in the above format usually appears as the defined local currency symbol. To get more locale specific formatting see see *Number Format Categories*, above.

9.16 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 cell's 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:

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

unlocked = workbook.add_format()
unlocked.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)
```

9.17 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:

```python
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)
```

9.18 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:
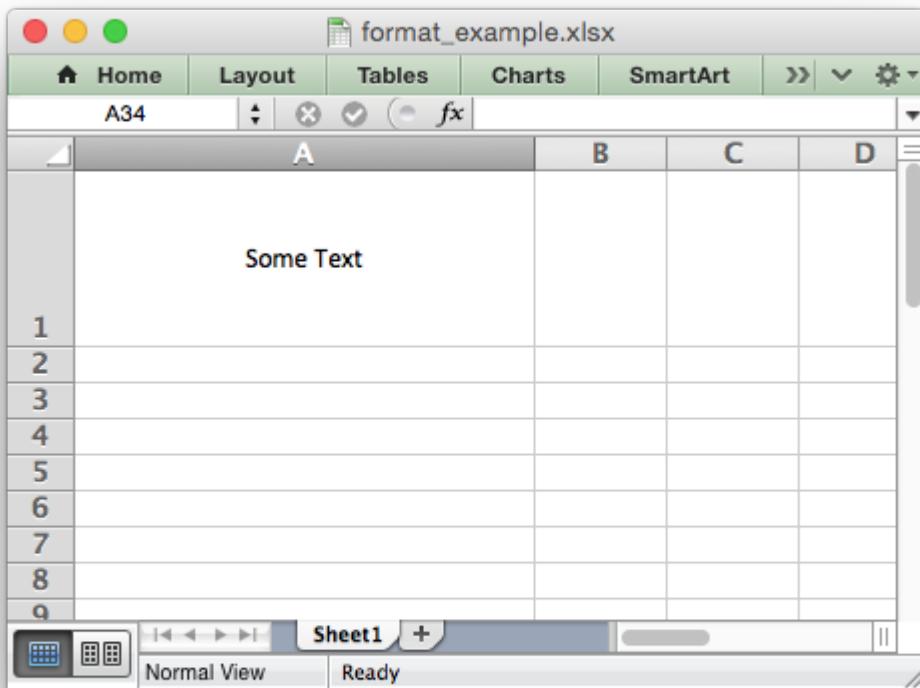
Horizontal alignment
left
center
right
fill
justify
center_across
distributed

The following are the available vertical alignments:

Vertical alignment
top
vcenter
bottom
vjustify
vdistributed

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

```python
cell_format = workbook.add_format()

cell_format.set_align('center')
cell_format.set_align('vcenter')

worksheet.set_row(0, 70)
worksheet.set_column('A:A', 30)

worksheet.write(0, 0, 'Some Text', cell_format)
```

2a9056b87bc08a94

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.

9.19 format.set_center_across()

set_center_across()
    Center 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 the leftmost cell should contain the text. The other cells in the range should be blank but should include the formatting:

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

worksheet.write(1, 1, 'Center across selection', cell_format)
```

```
worksheet.write_blank(1, 2, '', cell_format)
```

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

9.20 format.set_text_wrap()
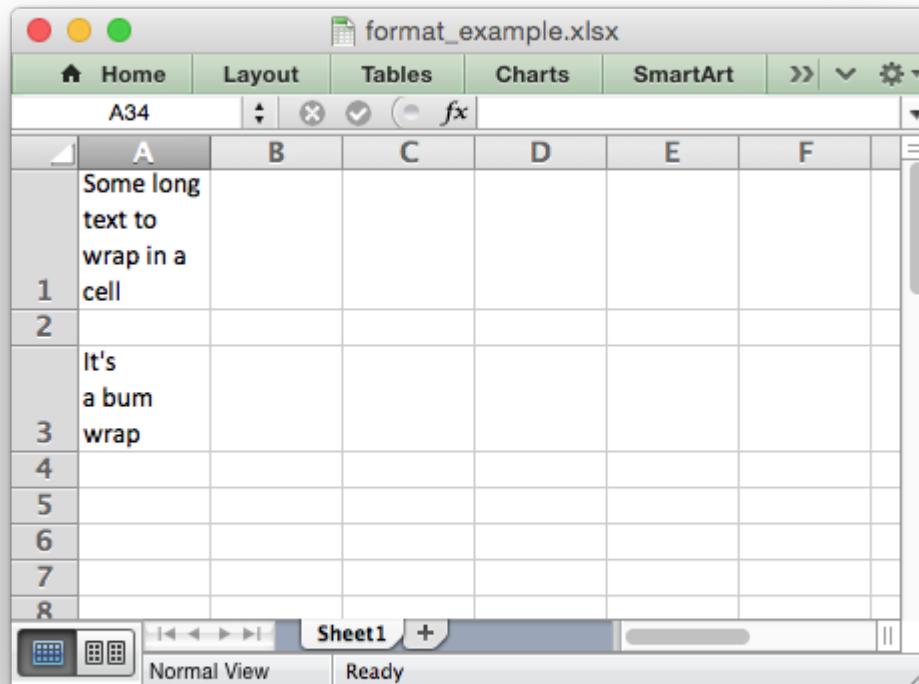
set_text_wrap()
    Wrap text in a cell.

Turn text wrapping on for text in a cell:

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

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

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

```
worksheet.write(2, 0, "It's\na bum\nwrap", cell_format)
```

Excel will adjust the height of the row to accommodate the wrapped text, as shown in the image above. This can be useful but it can also have unwanted side-effects:

- Objects such as images or charts that cross the automatically adjusted cells will not be scaled correctly. See *Object scaling due to automatic row height adjustment*.

- You may not want the row height to change. In that case you should set the row height to a non-default value such as 15.001.

9.21 format.set_rotation()
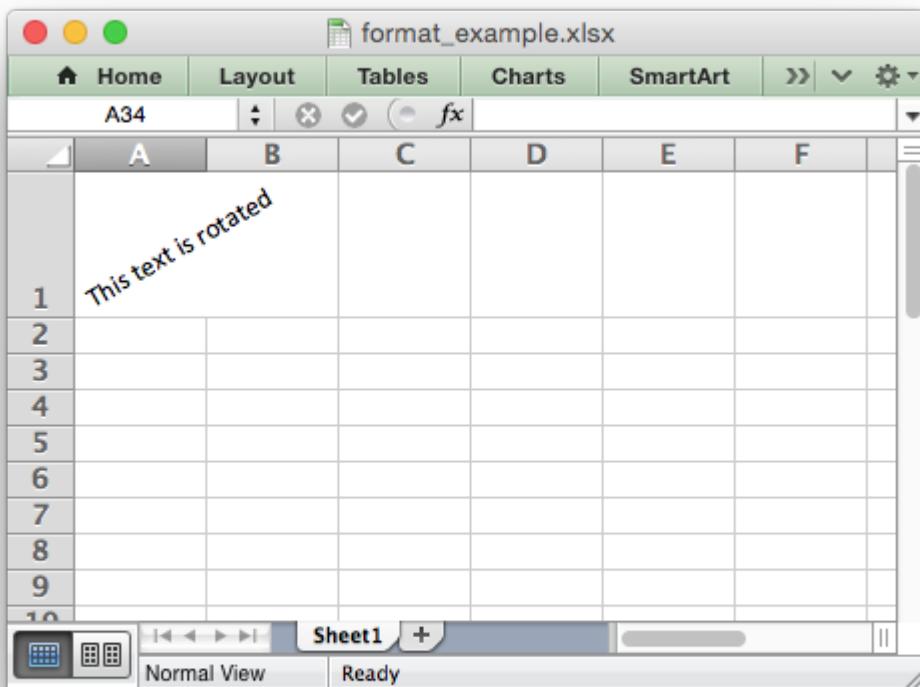
set_rotation(*angle*)
    Set the rotation of the text in a cell.

        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:

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

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


The angle 270 is also supported. This indicates text where the letters run from top to bottom.
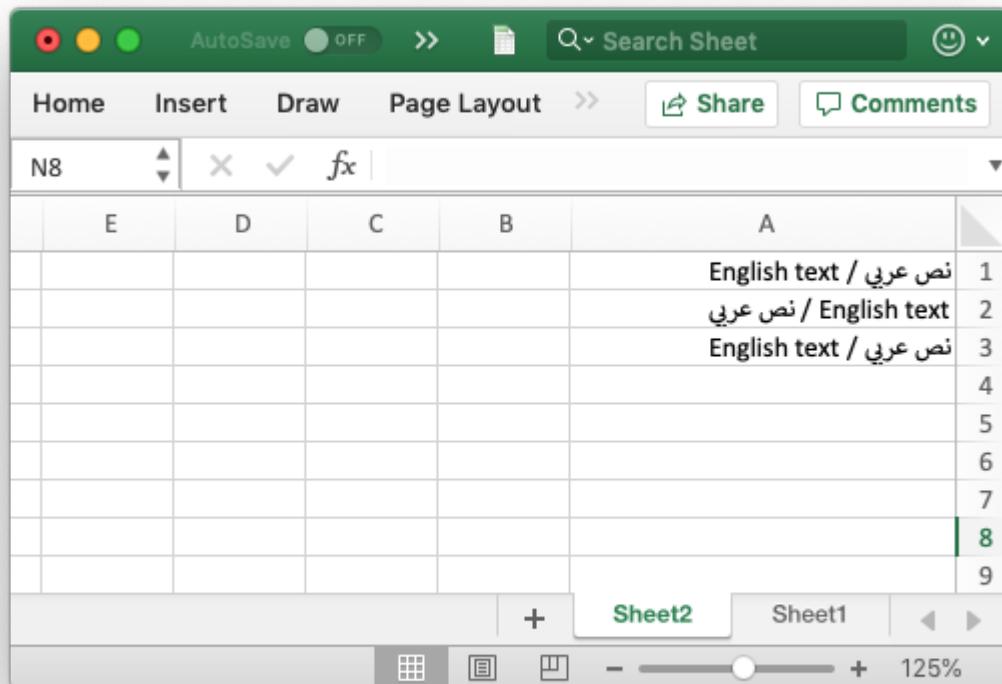
9.22 format.set_reading_order()

set_reading_order(*direction*)
> Set the reading order for the text in a cell.

> > **Parameters direction** (*int*) – Reading order direction.

Set the text reading direction. This is useful when creating Arabic, Hebrew or other near or far eastern worksheets. It can be used in conjunction with the Worksheet `right_to_left()` method to also change the direction of the worksheet.

9.23 format.set_indent()

set_indent(*level*)
> Set the cell text indentation level.
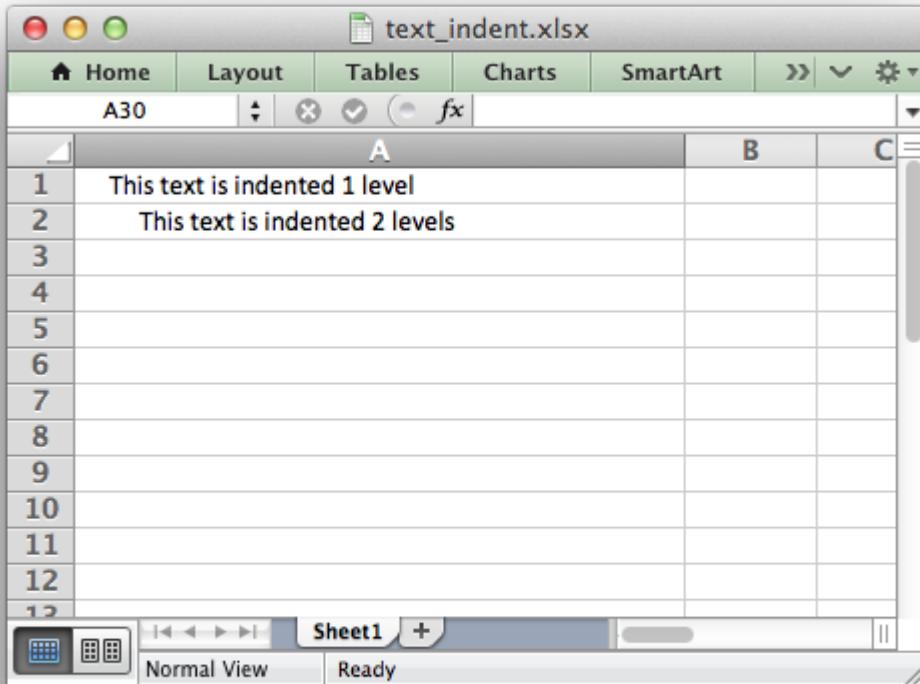
> > **Parameters level** (*int*) – Indentation level.

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:

```
cell_format1 = workbook.add_format()
cell_format2 = workbook.add_format()
```

```
cell_format1.set_indent(1)
cell_format2.set_indent(2)

worksheet.write('A1', 'This text is indented 1 level',  cell_format1)
worksheet.write('A2', 'This text is indented 2 levels', cell_format2)
```


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

9.24 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:

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

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

9.25 format.set_text_justlast()

set_text_justlast()
> Turn on the justify last text property.

Only applies to Far Eastern versions of Excel.

9.26 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.

9.27 format.set_bg_color()
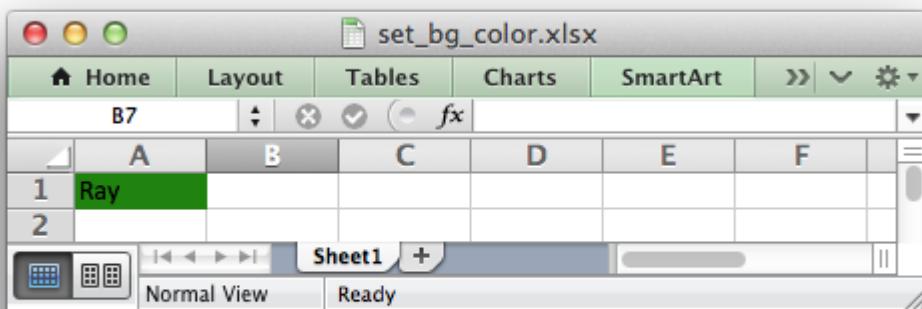
set_bg_color(*color*)
> Set the color of the background pattern in a cell.

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

The set_bg_color() method can be used to set the background color 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:

```
cell_format = workbook.add_format()

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

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

The color can be a Html style #RRGGBB string or a limited number of named colors, see *Working with Colors*.

9.28 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 color of a pattern.

The color can be a Html style #RRGGBB string or a limited number of named colors, see *Working with Colors*.

9.29 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	============

9.30 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.

9.31 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.

9.32 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.

9.33 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.

9.34 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 color 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 color 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*.

9.35 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.

9.36 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.

9.37 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.

9.38 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.

9.39 format.set_diag_border()
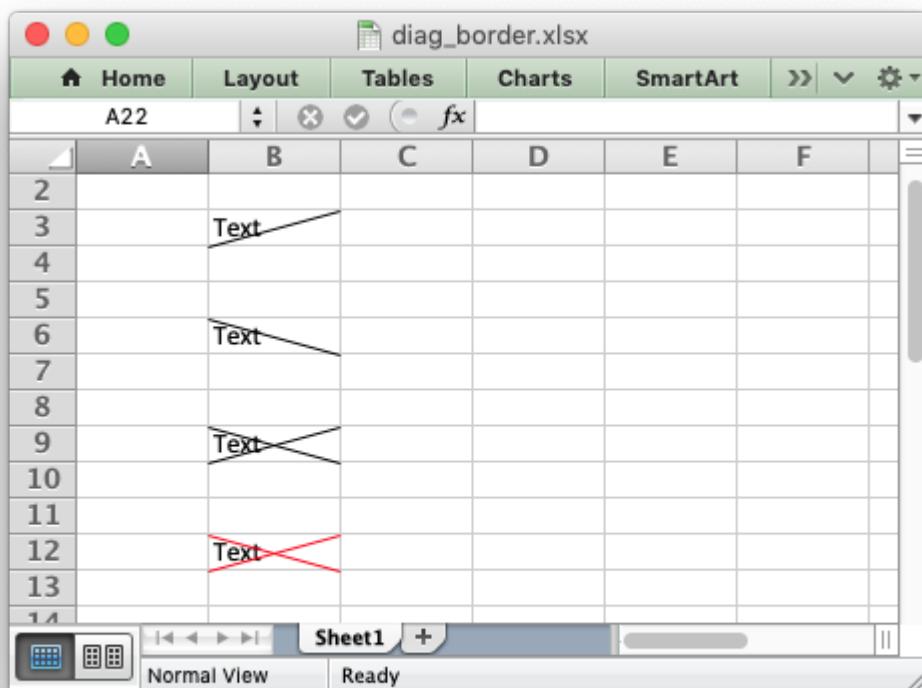
set_diag_border(*style*)
    Set the diagonal cell border style.

        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*.

9.40 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.

9.41 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.

9.42 format.set_quote_prefix()

set_quote_prefix()
>   Turn on quote prefix for the format.

Set the quote prefix property of a format to ensure a string is treated as a string after editing. This is the same as prefixing the string with a single quote in Excel. You don't need to add the quote to the string but you do need to add the format.

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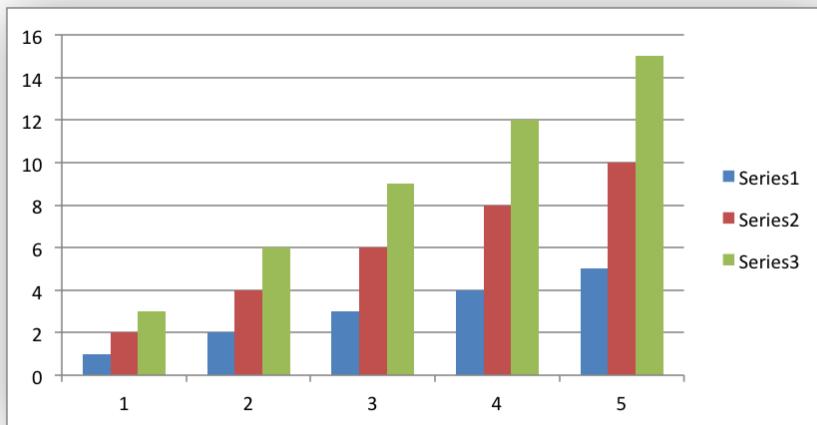
Or it can be set in a chartsheet using the set_chart() Chartsheet method:

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

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

```python
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

line
    stacked
    percent_stacked

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 defines 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:

```python
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 formula bar. For non-Pie/Doughnut charts it is also displayed in the legend. 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 color and width. See *Chart formatting: Line*.

- `border`: Set the border properties of the series such as color and style. See *Chart formatting: Border*.

- `fill`: Set the solid fill properties of the series such as color. See *Chart formatting: Solid Fill*.

- `pattern`: Set the pattern fill properties of the series. See *Chart formatting: Pattern Fill*.

- `gradient`: Set the gradient fill properties of the series. See *Chart formatting: Gradient Fill*.

- `marker`: Set the properties of the series marker such as style and color. 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 color 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()
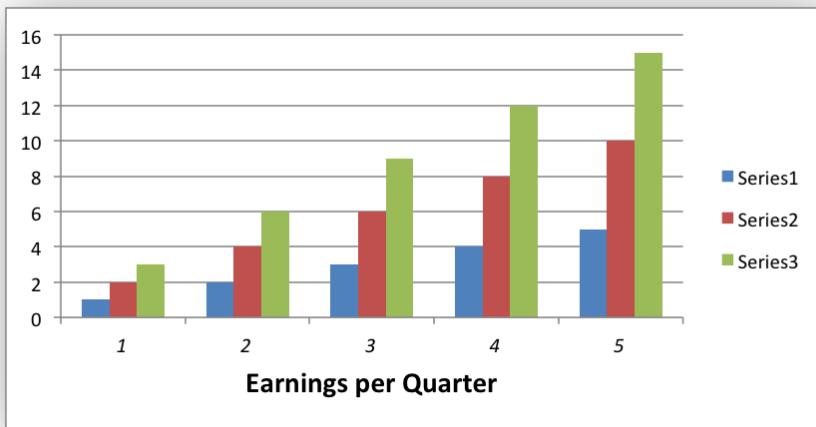
set_x_axis(*options*)
    Set the chart X axis options.

        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
pattern
gradient
min
max
minor_unit
major_unit
interval_unit
interval_tick
crossing
position_axis
reverse
log_base
label_position
label_align
major_gridlines
minor_gridlines
visible
date_axis
text_axis
minor_unit_type
major_unit_type
minor_tick_mark
major_tick_mark
display_units
display_units_visible
```

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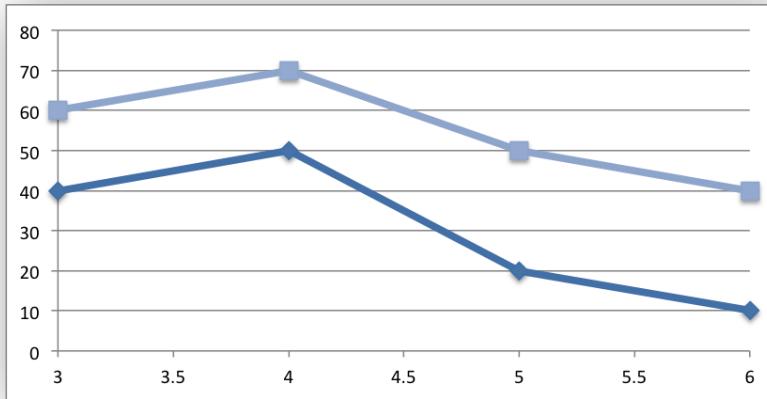
The number format is similar to the Worksheet Cell Format num_format apart from the fact that a format index cannot be used. An explicit format string must be used as shown above. See set_num_format() for more information.

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

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

- `fill`: Set the solid fill properties of the axis such as color. See *Chart formatting: Solid 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.

- `pattern`: Set the pattern fill properties of the axis. See *Chart formatting: Pattern Fill*.

- `gradient`: Set the gradient fill properties of the axis. See *Chart formatting: Gradient 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': 5})
    ```

- `interval_tick`: Set the tick interval for a category axis. Should be an integer value. (Applicable to category axes only.):

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

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

    The `crossing` value can be a numeric value or the strings `'max'` or `'min'` to set the crossing at the maximum/minimum axis:

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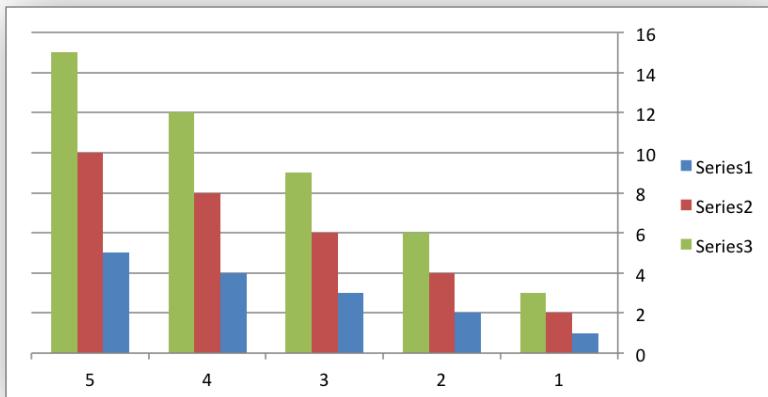
• `position_axis`: Position the axis on or between the axis tick marks. (Applicable to category axes only.)

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_x_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'})
```

- `label_align`: Align the "Axis labels" the axis. (Applicable to category axes only.)

  The following Excel alignments are available:

  ```
  center (the default)
  right
  left
  ```

  For example:
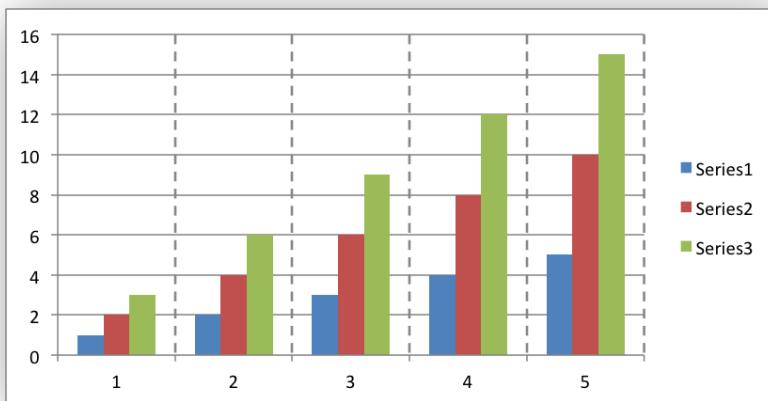
  ```
  chart.set_x_axis({'label_align': 'left'})
  ```

- `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 color 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.

- `minor_tick_mark`: Set the axis minor tick mark type/position to one of the following values:

```
none
inside
outside
cross    (inside and outside)
```

For example:

```
chart.set_x_axis({'major_tick_mark': 'none',
                  'minor_tick_mark': 'inside'})
```
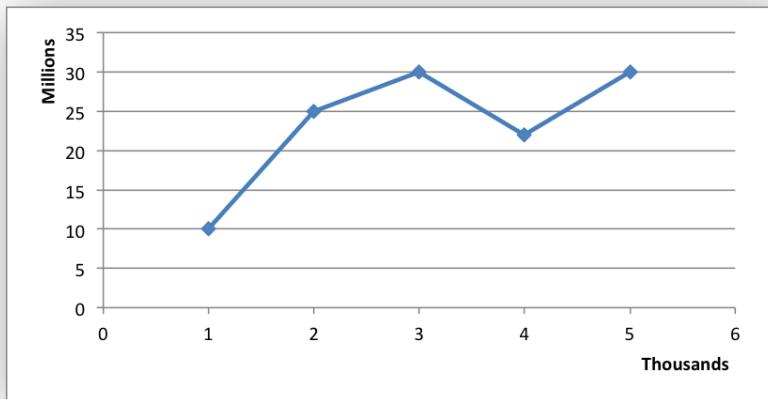
- `major_tick_mark`: Same as `minor_tick_mark`, see above, but for major axes ticks.

- `display_units`: Set the display units for the axis. This can be useful if the axis numbers are very large but you don't want to represent them in scientific notation. The available display units are:

```
hundreds
thousands
ten_thousands
hundred_thousands
millions
```

```
ten_millions
hundred_millions
billions
trillions
```

Applicable to value axes only.:

```
chart.set_x_axis({'display_units': 'thousands'})
chart.set_y_axis({'display_units': 'millions'})
```


- `display_units_visible`: Control the visibility of the display units turned on by the previous option. This option is on by default. (Applicable to value axes only.):

```
chart.set_x_axis({'display_units': 'hundreds',
                  'display_units_visible': False})
```

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.combine()

combine(*chart*)
    Combine two charts of different types.
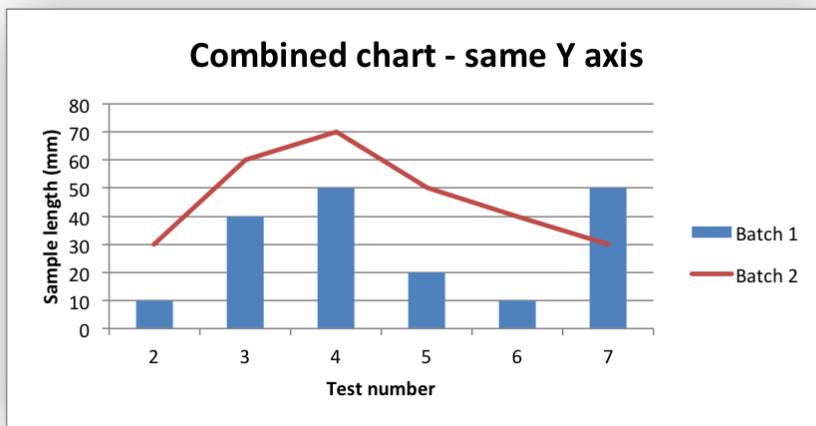
> **Parameters chart** – A chart object created with `add_chart()`.

The chart `combine()` method is used to combine two charts of different types, for example a column and line chart:

```
# Create a primary chart.
column_chart = workbook.add_chart({'type': 'column'})
column_chart.add_series({...})

# Create a secondary chart.
line_chart = workbook.add_chart({'type': 'line'})
line_chart.add_series({...})

# Combine the charts.
column_chart.combine(line_chart)
```

See the *Combined Charts* section for more details.

10.7 chart.set_size()

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})
```

10.8 chart.set_title()
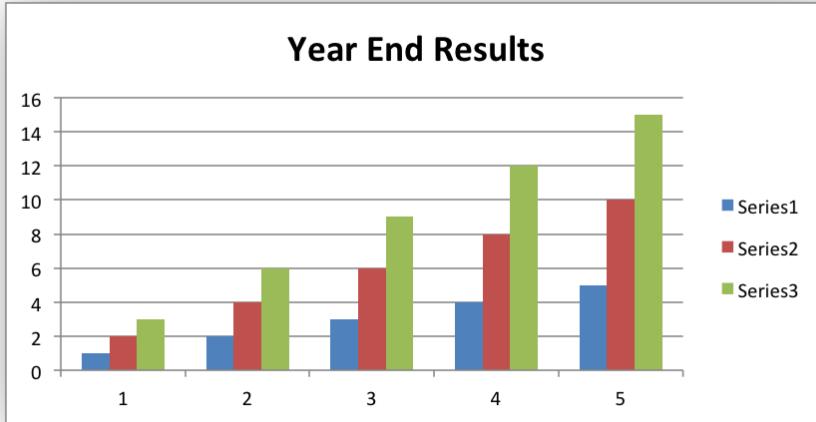
set_title(*options*)
    Set the chart title options.

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})
```

10.9 chart.set_legend()
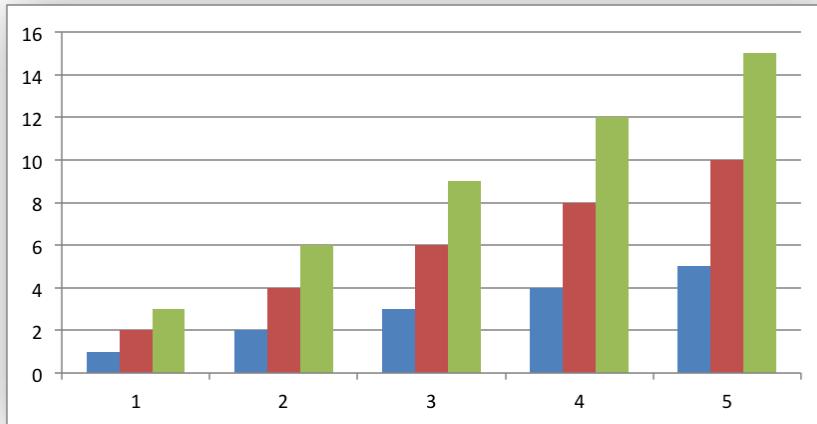
set_legend(*options*)
    Set the chart legend options.

>   **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
font
border
fill
pattern
gradient
delete_series
layout
```

- none: In Excel chart legends are on by default. The none option turns off the chart legend:
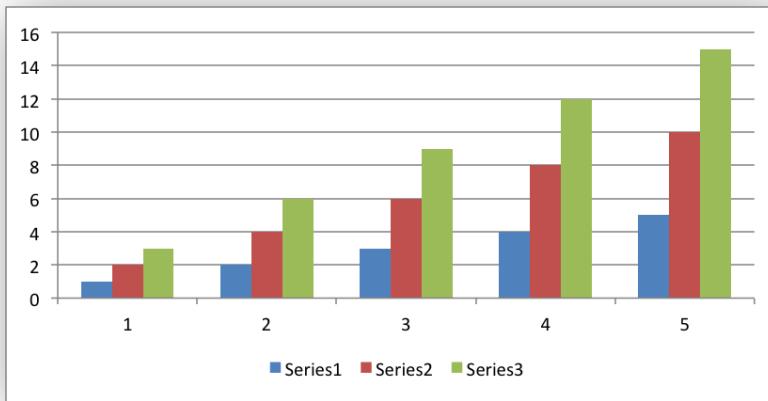
```
chart.set_legend({'none': True})
```

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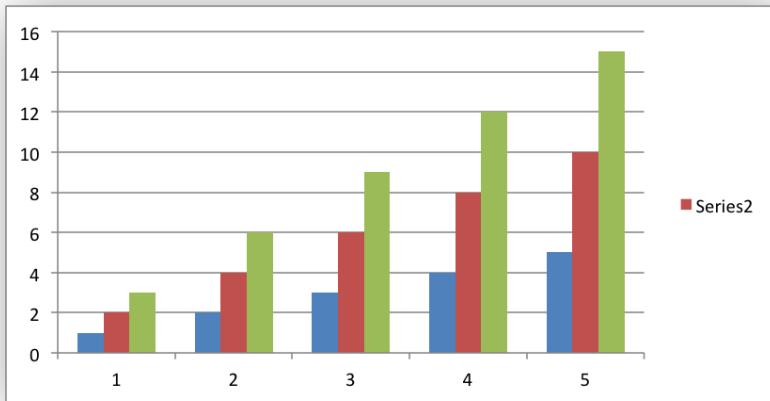
- `font`: Set the font properties of the chart legend:

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

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

- `border`: Set the border properties of the legend such as color and style. See *Chart formatting: Border*.

- `fill`: Set the solid fill properties of the legend such as color. See *Chart formatting: Solid Fill*.

- `pattern`: Set the pattern fill properties of the legend. See *Chart formatting: Pattern Fill*.

- `gradient`: Set the gradient fill properties of the legend. See *Chart formatting: Gradient Fill*.

- `delete_series`: This allows you to remove one or more series from 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]})
```

- `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.

10.10 chart.set_chartarea()
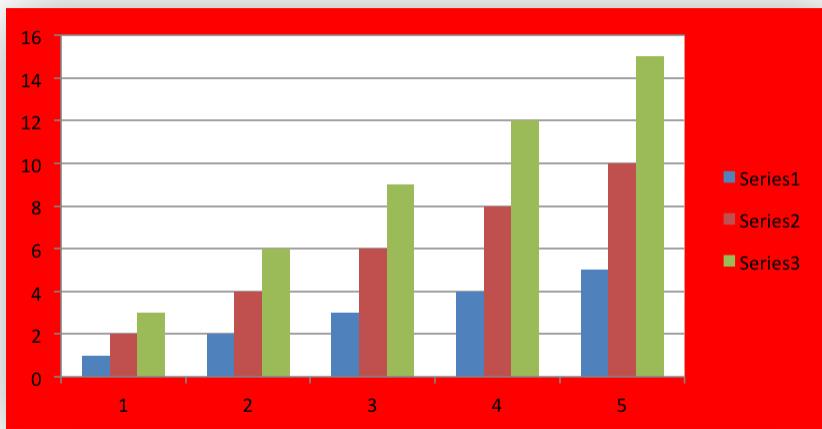
set_chartarea(*options*)
    Set the chart area options.

> **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 color and style. See *Chart formatting: Border*.

- `fill`: Set the solid fill properties of the chartarea such as color. See *Chart formatting: Solid Fill*.

- `pattern`: Set the pattern fill properties of the chartarea. See *Chart formatting: Pattern Fill*.

- `gradient`: Set the gradient fill properties of the chartarea. See *Chart formatting: Gradient Fill*.

10.11 chart.set_plotarea()
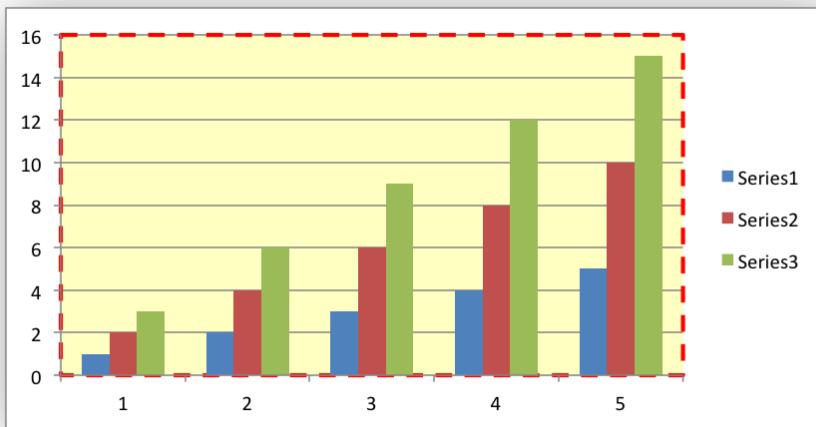
set_plotarea(*options*)
    Set the plot area options.

> **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 color and style. See *Chart formatting: Border*.

- `fill`: Set the solid fill properties of the plotarea such as color. See *Chart formatting: Solid Fill*.

- `pattern`: Set the pattern fill properties of the plotarea. See *Chart formatting: Pattern Fill*.

- `gradient`: Set the gradient fill properties of the plotarea. See *Chart formatting: Gradient Fill*.

- `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.

10.12 chart.set_style()
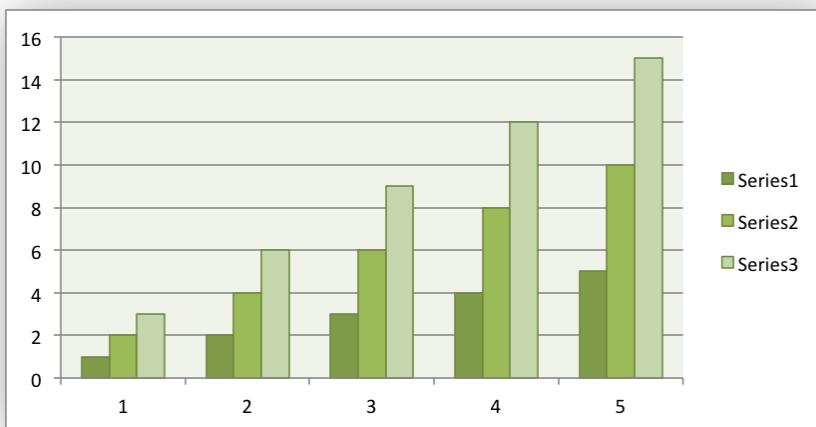
set_style(*style_id*)
    Set the chart style type.

        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.

10.13 chart.set_table()
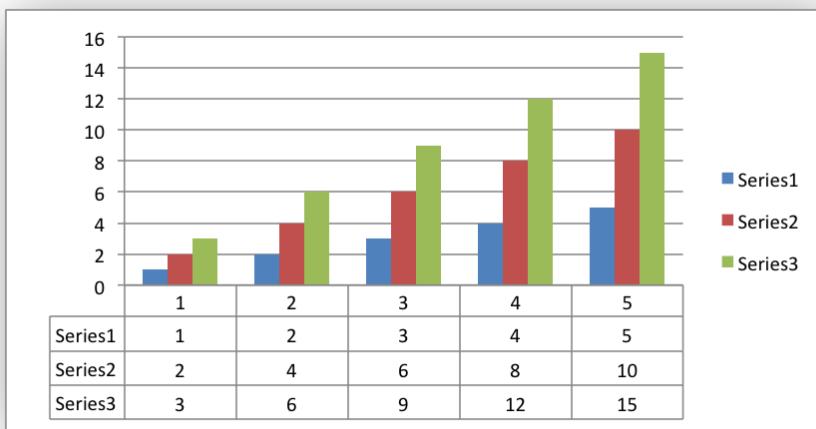
set_table(*options*)
> Set properties for an axis data table.
>
>> **Parameters options** (*dict*) – A dictionary of axis table options.

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.
'font':       {}     # Standard chart font properties.
```

For example:

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

The data table can only be shown with Bar, Column, Line, Area and stock charts. See the *Chart Fonts* section for more details on font properties.

10.14 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.
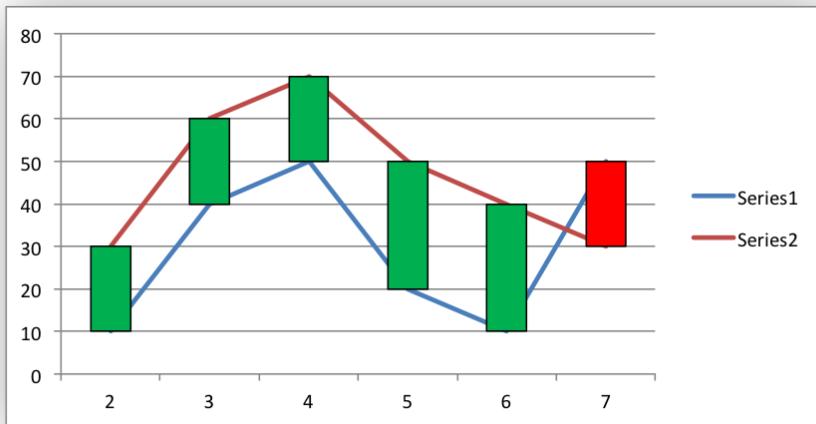
The `set_up_down_bars()` method adds Up-Down bars to Line charts to indicate the difference between the first and last data series:

```
chart.set_up_down_bars()
```

It is possible to format the up and down bars to add `fill`, `pattern` or `gradient` 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).

10.15 chart.set_drop_lines()
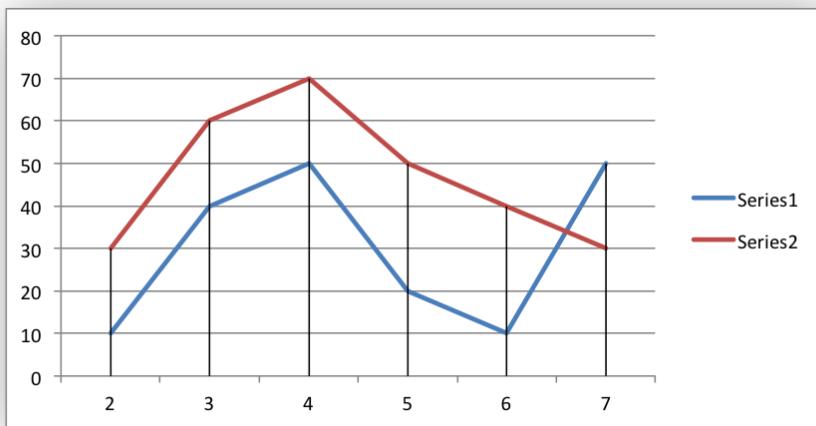
set_drop_lines(*options*)
> Set properties for the chart drop lines.

> > **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.

10.16 chart.set_high_low_lines()
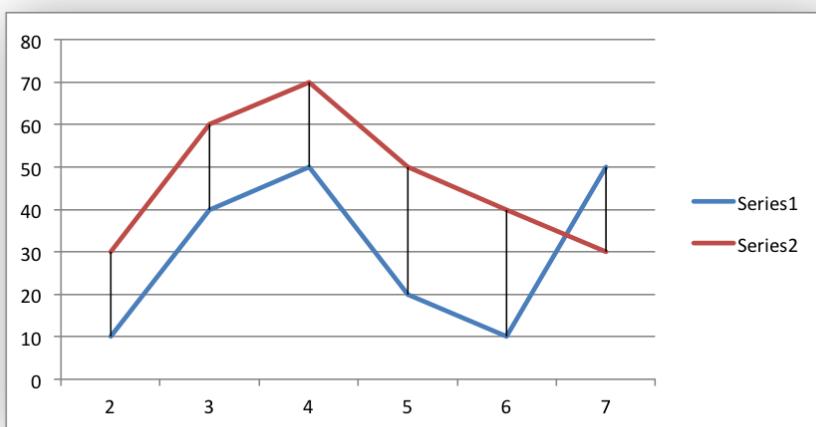
set_high_low_lines(*options*)
    Set properties for the chart high-low lines.

        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.

10.17 chart.show_blanks_as()

show_blanks_as(*option*)
 Set the option for displaying blank/empty data cells in a chart.

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

The `show_blanks_as()` method controls how blank/empty 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.
```

10.18 chart.show_na_as_empty_cell()

show_na_as_empty_cell()
 Display #N/A on charts as blank/empty cells.

Display #N/A values on a chart as blank/empty cells.:

```
chart.show_na_as_empty_cell()
```

10.19 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()
```

10.20 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 in the range `0 <= rotation <= 360`.

This option is only available for Pie/Doughnut charts.

10.21 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 value of the hole size must be in the range `10 <= size <= 90`.

This option is only available for Doughnut charts.

See also *Working with Charts* and *Chart Examples*.
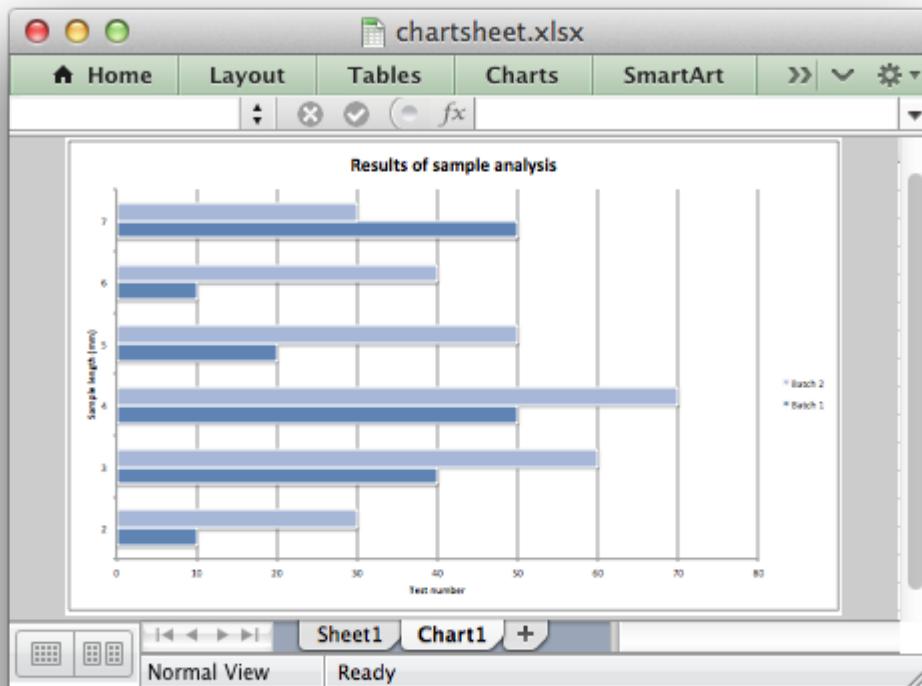
THE CHARTSHEET CLASS

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*.

THE EXCEPTIONS CLASS

The Exception class contains the various exceptions that can be raised by XlsxWriter. In general XlsxWriter only raised exceptions for un-recoverable errors or for errors that would lead to file corruption such as creating two worksheets with the same name.

The hierarchy of exceptions in XlsxWriter is:

- `XlsxWriterException(Exception)`
    - `XlsxFileError(XlsxWriterException)`
        - `FileCreateError(XlsxFileError)`
        - `UndefinedImageSize(XlsxFileError)`
        - `UndefinedImageSize(XlsxFileError)`
        - `FileSizeError(XlsxFileError)`
    - `XlsxInputError(XlsxWriterException)`
        - `DuplicateTableName(XlsxInputError)`
        - `InvalidWorksheetName(XlsxInputError)`
        - `DuplicateWorksheetName(XlsxInputError)`
        - `OverlappingRange(XlsxInputError)`

12.1 Exception: XlsxWriterException

exception **XlsxWriterException**

Base exception for XlsxWriter.

12.2 Exception: XlsxFileError

exception **XlsxFileError**

Base exception for all file related errors.

12.3 Exception: XlsxInputError

exception XlsxInputError

Base exception for all input data related errors.

12.4 Exception: FileCreateError

exception FileCreateError

This exception is raised if there is a file permission, or IO error, when writing the xlsx file to disk. This can be caused by an non-existent directory or (in Windows) if the file is already open in Excel:

```python
import xlsxwriter

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

worksheet = workbook.add_worksheet()

# The file exception.xlsx is already open in Excel.
workbook.close()
```

Raises:

```
xlsxwriter.exceptions.FileCreateError:
    [Errno 13] Permission denied: 'exception.xlsx'
```

This exception can be caught in a `try` block where you can instruct the user to close the open file before overwriting it:

```python
while True:
    try:
        workbook.close()
    except xlsxwriter.exceptions.FileCreateError as e:
        decision = input("Exception caught in workbook.close(): %s\n"
                         "Please close the file if it is open in Excel.\n"
                         "Try to write file again? [Y/n]: " % e)
        if decision != 'n':
            continue

    break
```

See also *Example: Catch exception on closing*.

12.5 Exception: UndefinedImageSize

exception UndefinedImageSize

This exception is raised if an image added via `insert_image()` doesn't contain height or width information. The exception is raised during Workbook `close()`:

```
import xlsxwriter

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

worksheet = workbook.add_worksheet()

worksheet.insert_image('A1', 'logo.png')

workbook.close()
```

Raises:

```
xlsxwriter.exceptions.UndefinedImageSize:
    logo.png: no size data found in image file.
```

Note: This is a relatively rare error that is most commonly caused by XlsxWriter failing to parse the dimensions of the image rather than the image not containing the information. In these cases you should raise a GitHub issue with the image attached, or provided via a link.

12.6 Exception: UnsupportedImageFormat

exception `UnsupportedImageFormat`

This exception is raised if if an image added via `insert_image()` isn't one of the supported file formats: PNG, JPEG, GIF, BMP, WMF or EMF. The exception is raised during Workbook `close()`:

```
import xlsxwriter

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

worksheet = workbook.add_worksheet()

worksheet.insert_image('A1', 'logo.xyz')

workbook.close()
```

Raises:

```
xlsxwriter.exceptions.UnsupportedImageFormat:
    logo.xyz: Unknown or unsupported image file format.
```

Note: If the image type is one of the supported types, and you are sure that the file format is correct, then the exception may be caused by XlsxWriter failing to parse the type of the image correctly. In these cases you should raise a GitHub issue with the image attached, or provided via a link.

12.7 Exception: FileSizeError

exception `FileSizeError`

This exception is raised if one of the XML files that is part of the xlsx file, or the xlsx file itself, exceeds 4GB in size:

```python
import xlsxwriter

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

worksheet = workbook.add_worksheet()

# Write lots of data to create a very big file.

workbook.close()
```

Raises:

```
xlsxwriter.exceptions.FileSizeError:
    Filesize would require ZIP64 extensions. Use workbook.use_zip64().
```

As noted in the exception message, files larger than 4GB can be created by turning on the zipfile.py ZIP64 extensions using the use_zip64() method.

12.8 Exception: EmptyChartSeries

exception EmptyChartSeries

This exception is raised if a chart is added to a worksheet without a data series. The exception is raised during Workbook close():

```python
import xlsxwriter

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

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

worksheet.insert_chart('A7', chart)

workbook.close()
```

Raises:

```
xlsxwriter.exceptions.EmptyChartSeries:
    Chart1 must contain at least one data series. See chart.add_series().
```

12.9 Exception: DuplicateTableName

exception DuplicateTableName

This exception is raised if a duplicate worksheet table name in used via add_table(). The exception is raised during Workbook close():

```
import xlsxwriter

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

worksheet.add_table('B1:F3', {'name': 'SalesData'})
worksheet.add_table('B4:F7', {'name': 'SalesData'})

workbook.close()
```

Raises:

```
xlsxwriter.exceptions.DuplicateTableName:
    Duplicate name 'SalesData' used in worksheet.add_table().
```

12.10 Exception: InvalidWorksheetName

exception InvalidWorksheetName

This exception is raised during Workbook add_worksheet() if a worksheet name is too long or contains restricted characters.

For example with a 32 character worksheet name:

```
import xlsxwriter

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

name = 'name_that_is_longer_than_thirty_one_characters'
worksheet = workbook.add_worksheet(name)

workbook.close()
```

Raises:

```
xlsxwriter.exceptions.InvalidWorksheetName:
    Excel worksheet name 'name_that_is_longer_than_thirty_one_characters'
    must be <= 31 chars.
```

Or for a worksheet name containing one of the Excel restricted characters, i.e. [ ] : * ? / \:

```
import xlsxwriter

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

worksheet = workbook.add_worksheet('Data[Jan]')

workbook.close()
```

Raises:

```
xlsxwriter.exceptions.InvalidWorksheetName:
    Invalid Excel character '[]:*?/\' in sheetname 'Data[Jan]'.
```

Or for a worksheet name start or ends with an apostrophe:

```python
import xlsxwriter

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

worksheet = workbook.add_worksheet("'Sheet1'")

workbook.close()
```

Raises:

```
xlsxwriter.exceptions.InvalidWorksheetName:
    Sheet name cannot start or end with an apostrophe "'Sheet1'".
```

12.11 Exception: DuplicateWorksheetName

exception DuplicateWorksheetName

This exception is raised during Workbook add_worksheet() if a worksheet name has already been used. As with Excel the check is case insensitive:

```python
import xlsxwriter

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

worksheet1 = workbook.add_worksheet('Sheet1')
worksheet2 = workbook.add_worksheet('sheet1')

workbook.close()
```

Raises:

```
xlsxwriter.exceptions.DuplicateWorksheetName:
    Sheetname 'sheet1', with case ignored, is already in use.
```

12.12 Exception: OverlappingRange

exception OverlappingRange

This exception is raised during Worksheet add_table() or merge_range() if the range overlaps an existing worksheet table or merge range. This is a file corruption error in Excel:

```python
import xlsxwriter

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

worksheet = workbook.add_worksheet()
```

```
worksheet.merge_range('A1:G10', 'Range 1')
worksheet.merge_range('G10:K20', 'Range 2')

workbook.close()
```

Raises:

```
xlsxwriter.exceptions.OverlappingRange:
    Merge range 'G10:K20' overlaps previous merge range 'A1:G10'.
```

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. This also applies to methods that take a range of cells:

```
worksheet.merge_range(2, 1, 3, 3, 'Merged Cells', merge_format)
worksheet.merge_range('B3:D4',   'Merged Cells', merge_format)
```

XlsxWriter supports Excel's worksheet limits of 1,048,576 rows by 16,384 columns.

Note:

- Ranges in A1 notation must be in uppercase, like in Excel.

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

13.1 Row and Column Ranges

In Excel you can specify row or column ranges such as `1:1` for all of the first row or `A:A` for all of the first column. In XlsxWriter these can be set by specifying the full cell range for the row or column:

```
worksheet.print_area('A1:XFD1')       # Same as 1:1
worksheet.print_area('A1:A1048576')   # Same as A:A
```

This is actually how Excel stores ranges such as `1:1` and `A:A` internally.

These ranges can also be specified using row-column notation, as explained above:

```
worksheet.print_area(0, 0,          0, 16383)  # Same as 1:1
worksheet.print_area(0, 0, 1048575,      0)  # Same as A:A
```

To select the entire worksheet range you can specify `A1:XFD1048576`.

13.2 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()` may require absolute references, depending on the range being specified.

13.3 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*.

13.4 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
```

13.4.1 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
```

13.4.2 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)
```

13.4.3 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
```

13.4.4 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
cell_range = xl_range(0, 0, 0, 0)   # A1
```

13.4.5 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
cell_range = xl_range_abs(0, 0, 0, 0)  # $A$1
```

WORKING WITH AND WRITING DATA

The following sections explain how to write various types of data to an Excel worksheet using XlsxWriter.

14.1 Writing data to a worksheet cell
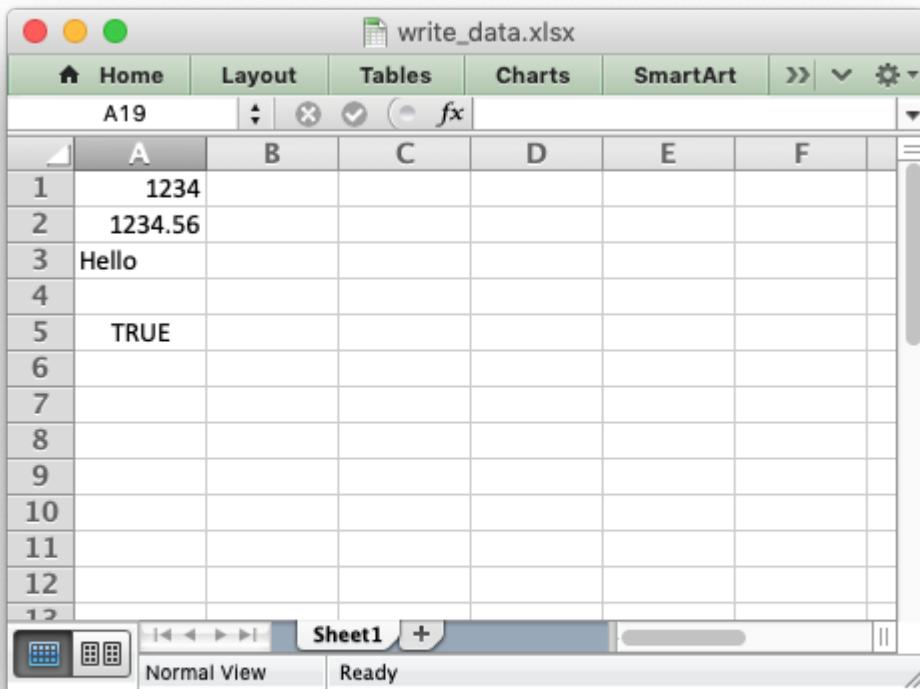
The worksheet `write()` method is the most common means of writing Python data to cells based on its type:

```python
import xlsxwriter

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

worksheet.write(0, 0, 1234)      # Writes an int
worksheet.write(1, 0, 1234.56)   # Writes a float
worksheet.write(2, 0, 'Hello')   # Writes a string
worksheet.write(3, 0, None)      # Writes None
worksheet.write(4, 0, True)      # Writes a bool

workbook.close()
```

The `write()` method uses the `type()` of the data to determine which specific method to use for writing the data. These methods then map some basic Python types to corresponding Excel types. The mapping is as follows:

Python type	Excel type	Worksheet methods

`int`
`long`
`float`
`Decimal`
`Fraction`
	Number	`write()`, `write_number()`

`basestring`
`str`
`unicode`
	String	`write()`, `write_string()`

`None`	String (blank)	`write()`, `write_blank()`

`datetime.date`
`datetime.datetime`
`datetime.time`
`datetime.timedelta`
	Number	`write()`, `write_datetime()`

`bool`	Boolean	`write()`, `write_boolean()`

The `write()` method also handles a few other Excel types that are encoded as Python strings in XlsxWriter:
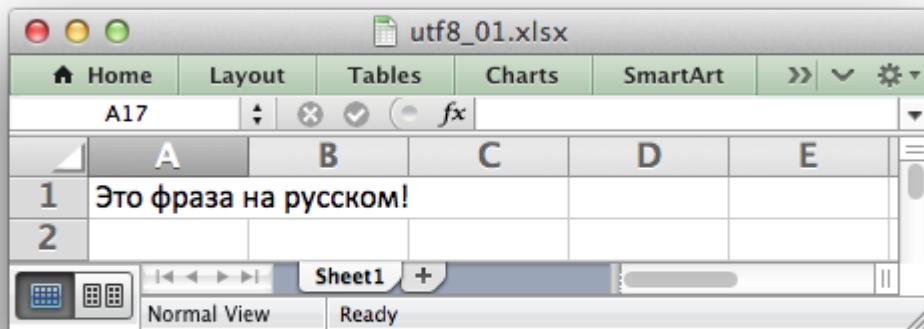
Pseudo-type	Excel type	Worksheet methods
formula string	Formula	`write()`, `write_formula()`
url string	URL	`write()`, `write_url()`

It should be noted that Excel has a very limited set of types to map to. The Python types that the `write()` method can handle can be extended as explained in the *Writing user defined types* section below.

14.2 Writing unicode data

Unicode data in Excel is encoded as UTF-8. XlsxWriter also supports writing UTF-8 data. This generally requires that your source file is UTF-8 encoded:

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


See *Example: Simple Unicode with Python 3* for a more complete example.

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. See *Example: Unicode - Polish in UTF-8* and *Example: Unicode - Shift JIS*.

14.3 Writing lists of data

Writing compound data types such as lists with XlsxWriter is done the same way it would be in any other Python program: with a loop. The Python `enumerate()` function is also very useful in this context:
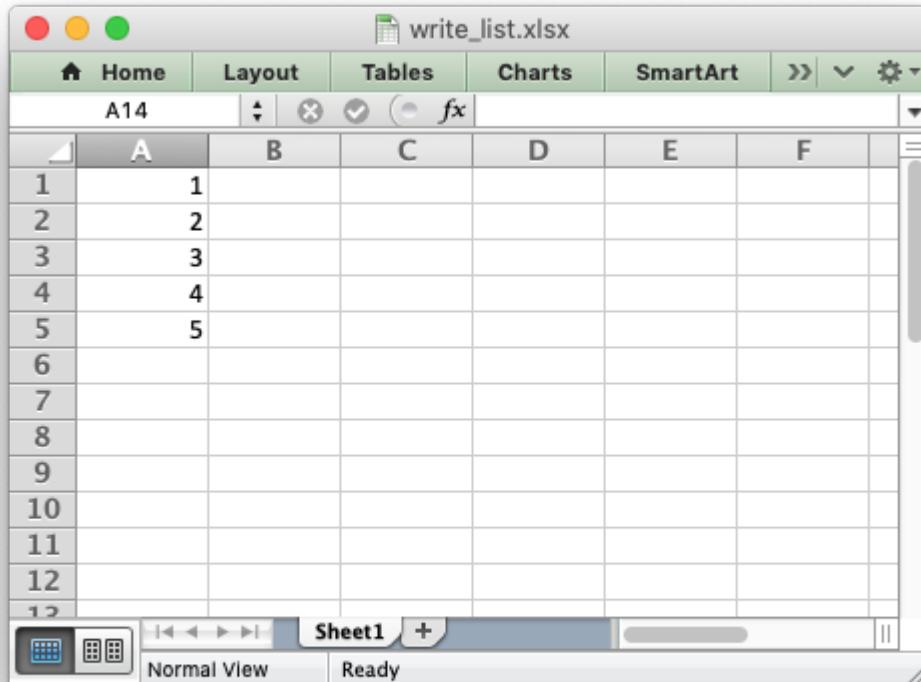
```python
import xlsxwriter

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

my_list = [1, 2, 3, 4, 5]
```

```python
for row_num, data in enumerate(my_list):
    worksheet.write(row_num, 0, data)

workbook.close()
```


Or if you wanted to write this horizontally as a row:
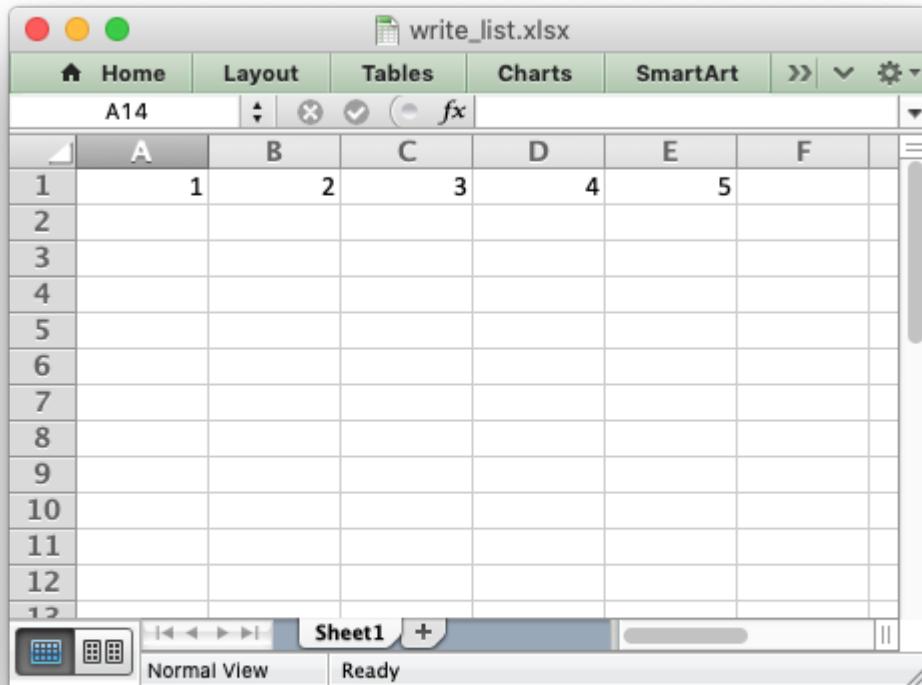
```python
import xlsxwriter

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

my_list = [1, 2, 3, 4, 5]

for col_num, data in enumerate(my_list):
    worksheet.write(0, col_num, data)

workbook.close()
```

Chapter 14. Working with and Writing Data

For a list of lists structure you would use two loop levels:
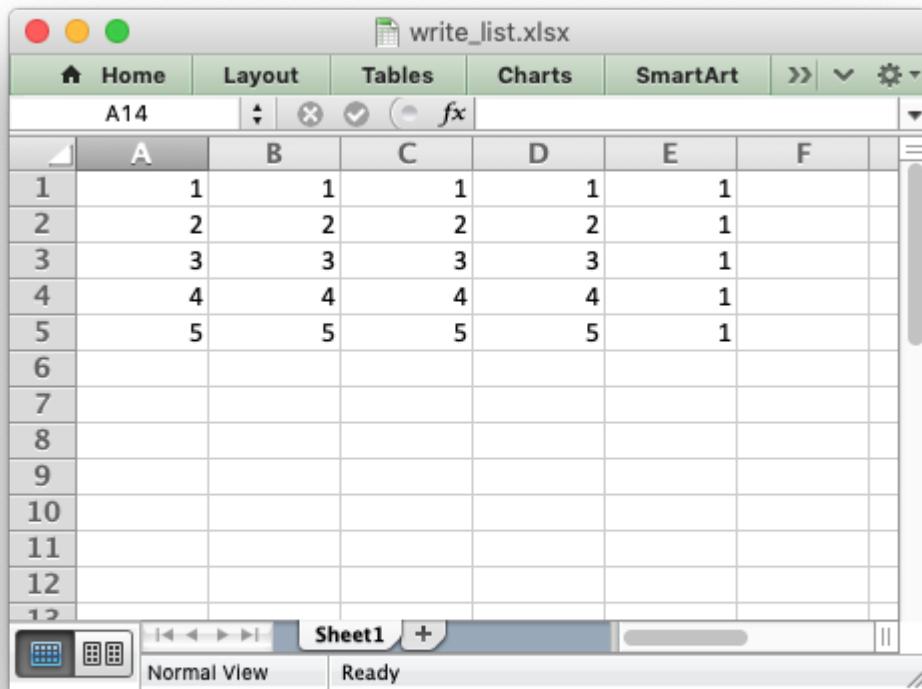
```
import xlsxwriter

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

my_list = [[1, 1, 1, 1, 1],
           [2, 2, 2, 2, 1],
           [3, 3, 3, 3, 1],
           [4, 4, 4, 4, 1],
           [5, 5, 5, 5, 1]]

for row_num, row_data in enumerate(my_list):
    for col_num, col_data in enumerate(row_data):
        worksheet.write(row_num, col_num, col_data)

workbook.close()
```

The *worksheet* class has two utility functions called `write_row()` and `write_column()` which are basically a loop around the `write()` method:
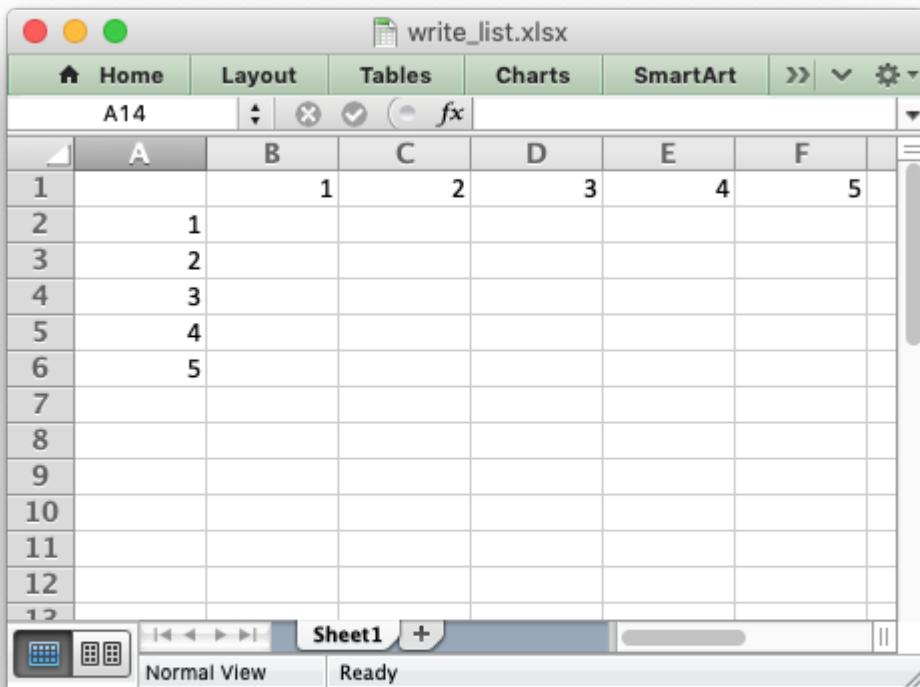
```python
import xlsxwriter

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

my_list = [1, 2, 3, 4, 5]

worksheet.write_row(0, 1, my_list)
worksheet.write_column(1, 0, my_list)

workbook.close()
```

14.4 Writing dicts of data

Unlike lists there is no single simple way to write a Python dictionary to an Excel worksheet using Xlsxwriter. The method will depend of the structure of the data in the dictionary. Here is a simple example for a simple data structure:
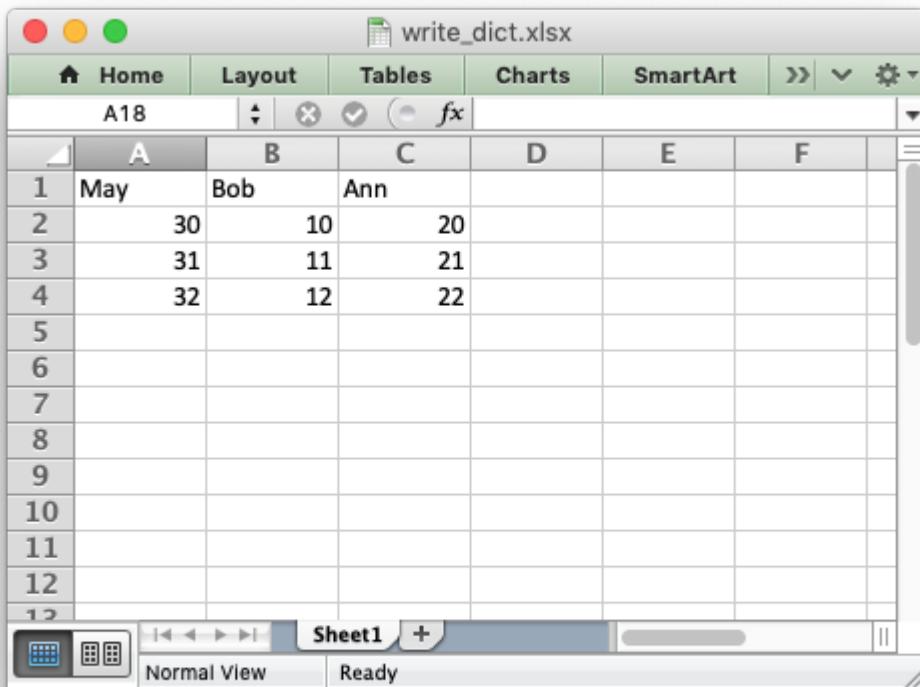
```python
import xlsxwriter

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

my_dict = {'Bob': [10, 11, 12],
           'Ann': [20, 21, 22],
           'May': [30, 31, 32]}

col_num = 0
for key, value in my_dict.items():
    worksheet.write(0, col_num, key)
    worksheet.write_column(1, col_num, value)
    col_num += 1

workbook.close()
```

14.5 Writing dataframes

The best way to deal with dataframes or complex data structure is to use Python Pandas. Pandas is a Python data analysis library. It can read, filter and re-arrange small and large data sets and output them in a range of formats including Excel.

To use XlsxWriter with Pandas you specify it as the Excel writer *engine*:
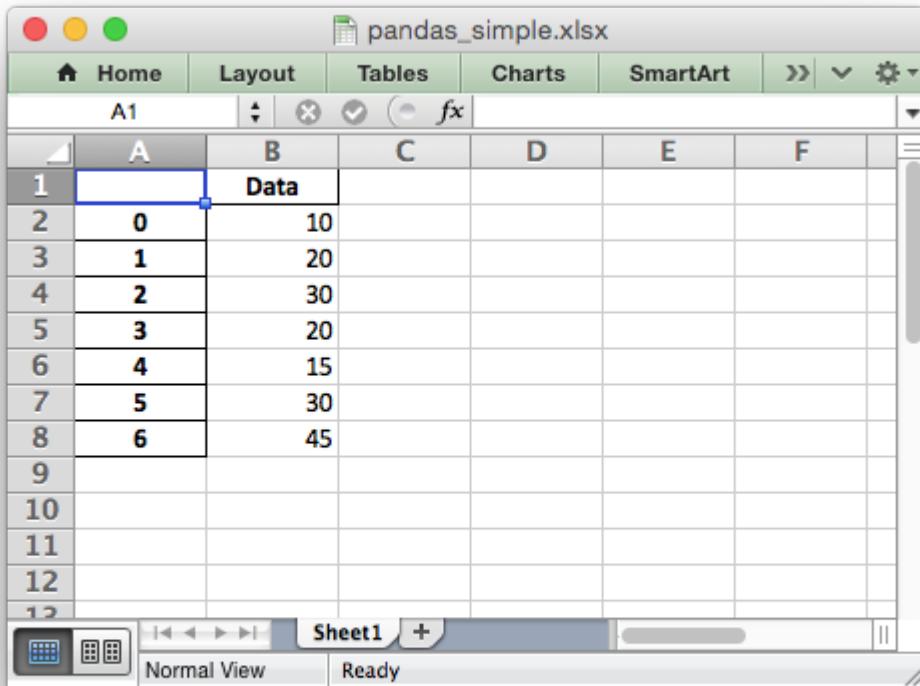
```python
import pandas as pd

# Create a Pandas dataframe from the data.
df = pd.DataFrame({'Data': [10, 20, 30, 20, 15, 30, 45]})

# Create a Pandas Excel writer using XlsxWriter as the engine.
writer = pd.ExcelWriter('pandas_simple.xlsx', engine='xlsxwriter')

# Convert the dataframe to an XlsxWriter Excel object.
df.to_excel(writer, sheet_name='Sheet1')

# Close the Pandas Excel writer and output the Excel file.
writer.close()
```

The output from this would look like the following:

For more information on using Pandas with XlsxWriter see *Working with Pandas and XlsxWriter*.

14.6 Writing user defined types

As shown in the first section above, the worksheet `write()` method maps the main Python data types to Excel's data types. If you want to write an unsupported type then you can either avoid `write()` and map the user type in your code to one of the more specific write methods or you can extend it using the `add_write_handler()` method. This can be, occasionally, more convenient then adding a lot of if/else logic to your code.

As an example, say you wanted to modify `write()` to automatically write `uuid` types as strings. You would start by creating a function that takes the uuid, converts it to a string and then writes it using `write_string()`:

```python
def write_uuid(worksheet, row, col, uuid, format=None):
    return worksheet.write_string(row, col, str(uuid), format)
```

You could then add a handler that matches the `uuid` type and calls your user defined function:

```python
#                           match,      action()
worksheet.add_write_handler(uuid.UUID, write_uuid)
```

Then you can use `write()` without further modification:

```python
my_uuid = uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')

# Write the UUID. This would raise a TypeError without the handler.
worksheet.write('A1', my_uuid)
```

Multiple callback functions can be added using `add_write_handler()` but only one callback action is allowed per type. However, it is valid to use the same callback function for different types:

```python
worksheet.add_write_handler(int,   test_number_range)
worksheet.add_write_handler(float, test_number_range)
```

14.6.1 How the write handler feature works

The `write()` method is mainly a large `if()` statement that checks the `type()` of the input value and calls the appropriate worksheet method to write the data. The `add_write_handler()` method works by injecting additional type checks and associated actions into this `if()` statement.

Here is a simplified version of the `write()` method:

```python
def write(self, row, col, *args):

    # The first arg should be the token for all write calls.
    token = args[0]

    # Get the token type.
    token_type = type(token)

    # Check for any user defined type handlers with callback functions.
    if token_type in self.write_handlers:
        write_handler = self.write_handlers[token_type]
        function_return = write_handler(self, row, col, *args)

        # If the return value is None then the callback has returned
        # control to this function and we should continue as
```

```
        # normal. Otherwise we return the value to the caller and exit.
        if function_return is None:
            pass
        else:
            return function_return

    # Check for standard Python types, if we haven't returned already.
    if token_type is bool:
        return self.write_boolean(row, col, *args)

    # Etc. ...
```

14.6.2 The syntax of write handler functions

Functions used in the add_write_handler() method should have the following method signature/parameters:

```
def my_function(worksheet, row, col, token, format=None):
    return worksheet.write_string(row, col, token, format)
```
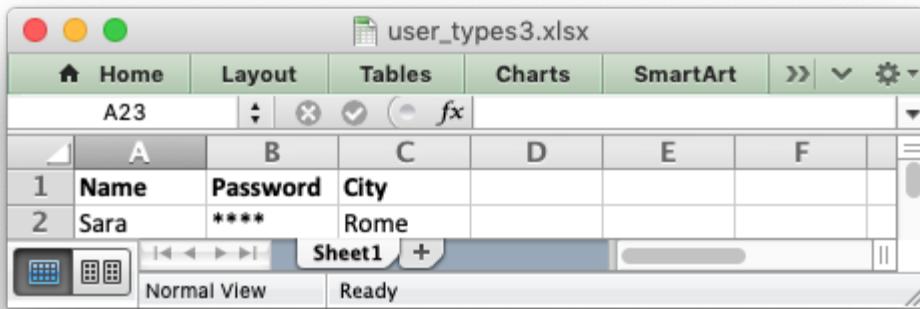
The function will be passed a *worksheet* instance, an integer row and col value, a token that matches the type added to add_write_handler() and some additional parameters. Usually the additional parameter(s) will only be a cell *format* instance. However, if you need to handle other additional parameters, such as those passed to write_url() then you can have more generic handling like this:

```
def my_function(worksheet, row, col, token, *args):
    return worksheet.write_string(row, col, token, *args)
```

Note, you don't have to explicitly handle A1 style cell ranges. These will be converted to row and column values prior to your function being called.

You can also make use of the row and col parameters to control the logic of the function. Say for example you wanted to hide/replace user passwords with '****' when writing string data. If your data was structured so that the password data was in the second column, apart from the header row, you could write a handler function like this:

```
def hide_password(worksheet, row, col, string, format=None):
    if col == 1 and row > 0:
        return worksheet.write_string(row, col, '****', format)
    else:
        return worksheet.write_string(row, col, string, format)
```

14.6.3 The return value of write handler functions

Functions used in the `add_write_handler()` method should return one of the following values:

- None: to indicate that control is return to the parent `write()` method to continue as normal. This is used if your handler function logic decides that you don't need to handle the matched token.

- The return value of the called `write_xxx()` function. This is generally 0 for no error and a negative number for errors. This causes an immediate return from the calling `write()` method with the return value that was passed back.

For example, say you wanted to ignore NaN values in your data since Excel doesn't support them. You could create a handler function like the following that matched against floats and which wrote a blank cell if it was a NaN or else just returned to `write()` to continue as normal:

```python
def ignore_nan(worksheet, row, col, number, format=None):
    if math.isnan(number):
        return worksheet.write_blank(row, col, None, format)
    else:
        # Return control to the calling write() method.
        return None
```

If you wanted to just drop the NaN values completely and not add any formatting to the cell you could just return 0, for no error:

```python
def ignore_nan(worksheet, row, col, number, format=None):
    if math.isnan(number):
        return 0
    else:
        # Return control to the calling write() method.
        return None
```

14.6.4 Write handler examples

See the following, more complete, examples of handling user data types:
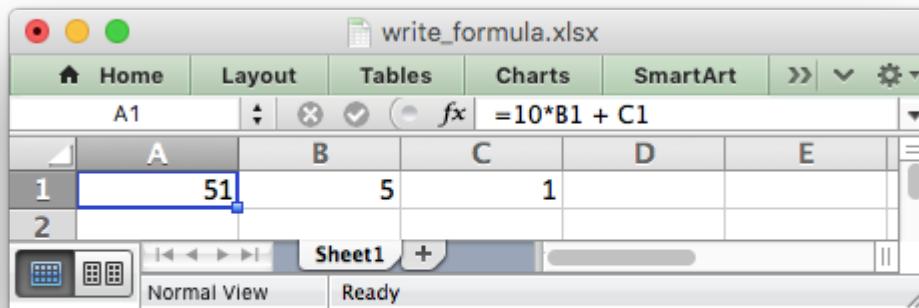
- *Example: Writing User Defined Types (1)*
- *Example: Writing User Defined Types (2)*
- *Example: Writing User Defined types (3)*

WORKING WITH FORMULAS

In general a formula in Excel can be used directly in the `write_formula()` method:

```
worksheet.write_formula('A1', '=10*B1 + C1')
```


However, there are a few potential issues and differences that the user should be aware of. These are explained in the following sections.

15.1 Non US Excel functions and syntax

Excel stores formulas in the format of the US English version, regardless of the language or locale of the end-user's version of Excel. Therefore all formula function names written using XlsxWriter must be in English:

```
worksheet.write_formula('A1', '=SUM(1, 2, 3)')    # OK
worksheet.write_formula('A2', '=SOMME(1, 2, 3)')  # French. Error on load.
```

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)')    # Semi-colon. Error on load.
```

If you have a non-English version of Excel you can use the following multi-lingual formula translator to help you convert the formula. It can also replace semi-colons with commas.

15.2 Formula Results

XlsxWriter doesn't calculate the result 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 will only display the 0 results. Examples of such applications are Excel Viewer, PDF Converters, and some mobile device applications.

If required, it is also possible to specify the calculated result of the formula using the optional `value` parameter for `write_formula()`:

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

The `value` parameter can be a number, a string, a bool or one of the following Excel error codes:

```
#DIV/0!
#N/A
#NAME?
#NULL!
#NUM!
#REF!
#VALUE!
```

It is also possible to specify the calculated result of an array formula created with `write_array_formula()`:

```
# Specify the result for a single cell range.
worksheet.write_array_formula('A1:A1', '{=SUM(B1:C1*B2:C2)}', cell_format, 2005)
```

However, using this parameter only writes a single value to the upper left cell in the result array. For a multi-cell array formula where the results are required, the other result values can be specified by using `write_number()` to write to the appropriate cell:

```
# Specify the results for a multi cell range.
worksheet.write_array_formula('A1:A3', '{=TREND(C1:C3,B1:B3)}', cell_format, 15)
worksheet.write_number('A2', 12, cell_format)
worksheet.write_number('A3', 14, cell_format)
```

15.3 Dynamic Array support

Excel introduced the concept of "Dynamic Arrays" and new functions that use them in Office 365. The new functions are:

- BYCOL()
- BYROW()

- CHOOSECOLS()
- CHOOSEROWS()
- DROP()
- EXPAND()
- FILTER()
- HSTACK()
- MAKEARRAY()
- MAP()
- RANDARRAY()
- REDUCE()
- SCAN()
- SEQUENCE()
- SORT()
- SORTBY()
- SWITCH()
- TAKE()
- TEXTSPLIT()
- TOCOL()
- TOROW()
- UNIQUE()
- VSTACK()
- WRAPCOLS()
- WRAPROWS()
- XLOOKUP()

The following special case functions were also added with Dynamic Arrays:

- SINGLE() - Explained below in *Dynamic Arrays - The Implicit Intersection Operator "@"*.
- ANCHORARRAY() - Explained below in *Dynamic Arrays - The Spilled Range Operator "#"*.
- LAMBDA() and LET() - Explained below in *The Excel 365 LAMBDA() function*.

Dynamic arrays are ranges of return values that can change in size based on the results. For example, a function such as FILTER() returns an array of values that can vary in size depending on the filter results. This is shown in the snippet below from *Example: Dynamic array formulas*:

```
worksheet1.write('F2', '=FILTER(A1:D17,C1:C17=K2)')
```

15.3. Dynamic Array support 213

Which gives the results shown in the image below. The dynamic range here is "F2:I5" but it could be different based on the filter criteria.

It is also possible to get dynamic array behavior with older Excel functions. For example, the Excel function =LEN(A1) applies to a single cell and returns a single value but it is also possible to apply it to a range of cells and return a range of values using an array formula like {=LEN(A1:A3)}. This type of "static" array behavior is called a CSE (Ctrl+Shift+Enter) formula. With the introduction of dynamic arrays in Excel 365 you can now write this function as =LEN(A1:A3) and get a dynamic range of return values. In XlsxWriter you can use the write_array_formula() worksheet method to get a static/CSE range and write_dynamic_array_formula() to get a dynamic range. For example:

```
worksheet.write_dynamic_array_formula('B1:B3', '=LEN(A1:A3)')
```

Which gives the following result:
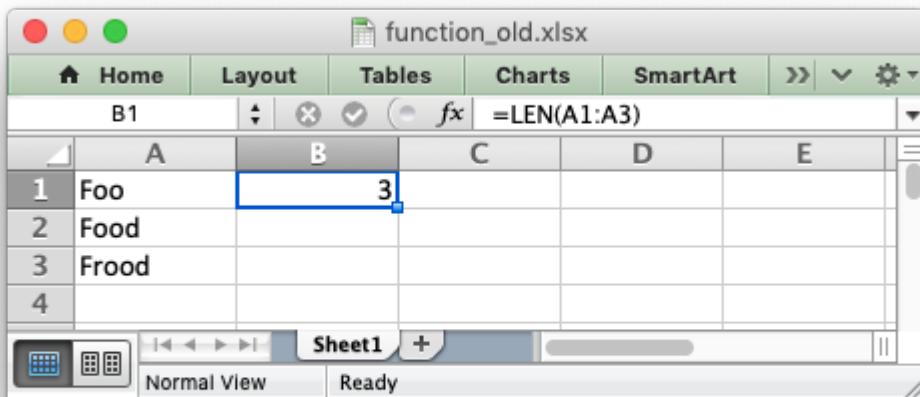
The difference between the two types of array functions is explained in the Microsoft documentation on Dynamic array formulas vs. legacy CSE array formulas. Note the use of the word "legacy" here. This, and the documentation itself, is a clear indication of the future importance of dynamic arrays in Excel.

For a wider and more general introduction to dynamic arrays see the following: Dynamic array formulas in Excel.
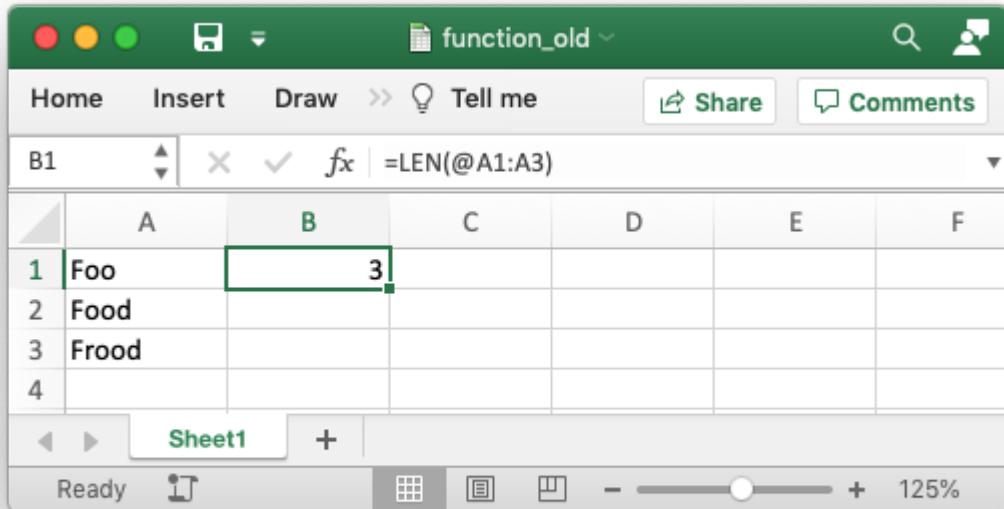
15.4 Dynamic Arrays - The Implicit Intersection Operator "@"

The Implicit Intersection Operator, "@", is used by Excel 365 to indicate a position in a formula that is implicitly returning a single value when a range or an array could be returned.

We can see how this operator works in practice by considering the formula we used in the last section: =LEN(A1:A3). In Excel versions without support for dynamic arrays, i.e. prior to Excel 365, this formula would operate on a single value from the input range and return a single value, like this:
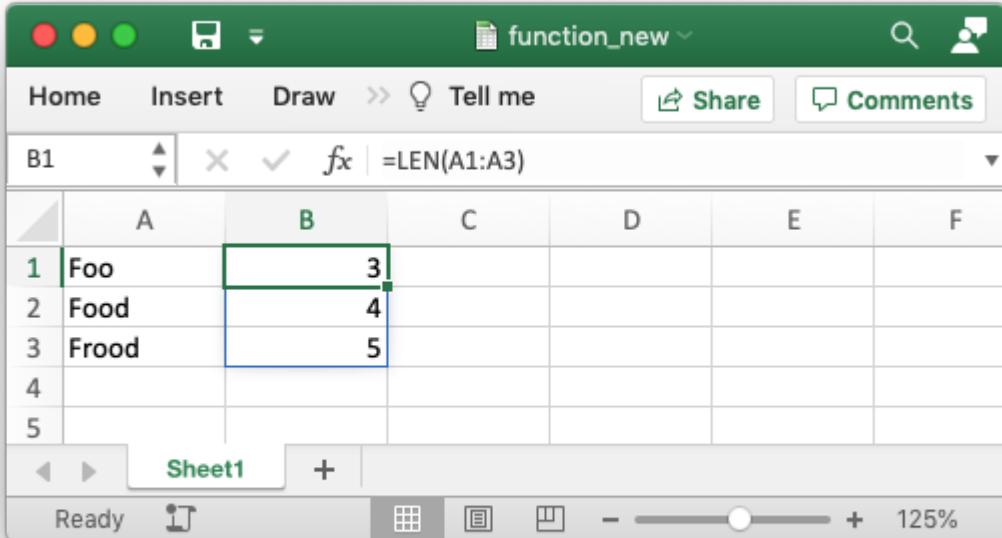
There is an implicit conversion here of the range of input values, "A1:A3", to a single value "A1". Since this was the default behavior of older versions of Excel this conversion isn't highlighted in any way. But if you open the same file in Excel 365 it will appear as follows:

The result of the formula is the same (this is important to note) and it still operates on, and returns, a single value. However the formula now contains a "@" operator to show that it is implicitly using a single value from the given range.

Finally, if you entered this formula in Excel 365, or with `write_dynamic_array_formula()` in XlsxWriter, it would operate on the entire range and return an array of values:

If you are encountering the Implicit Intersection Operator "@" for the first time then it is probably from a point of view of "why is Excel/XlsxWriter putting @s in my formulas". In practical terms if you encounter this operator, and you don't intend it to be there, then you should probably write the formula as a CSE or dynamic array function using `write_array_formula()` or `write_dynamic_array_formula()` (see the previous section on *Dynamic Array support*).
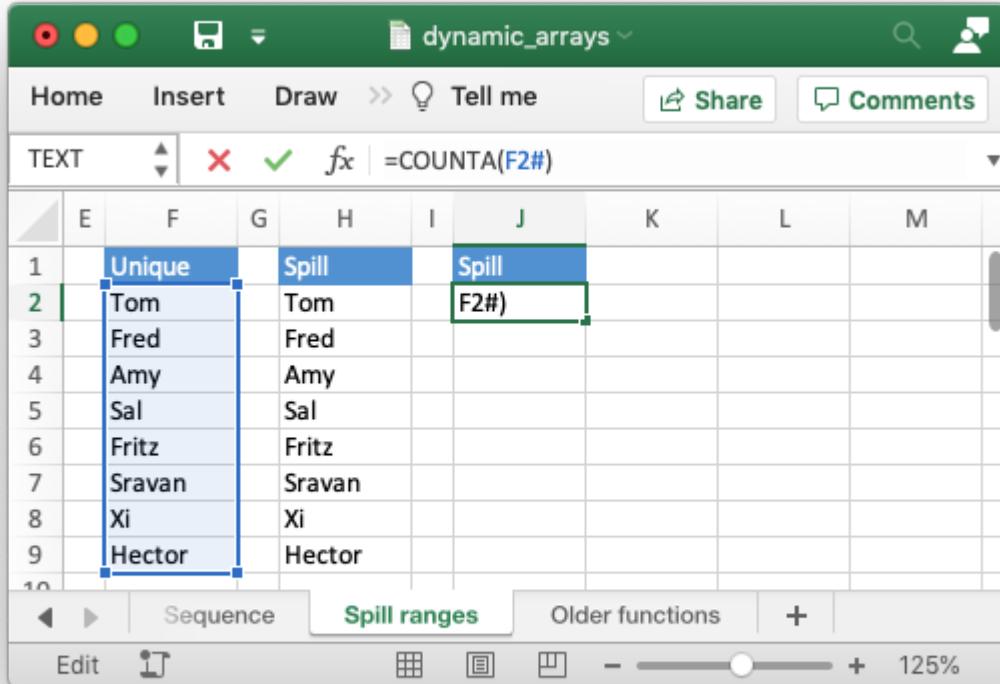
A full explanation of this operator is shown in the Microsoft documentation on the Implicit intersection operator: @.

One important thing to note is that the "@" operator isn't stored with the formula. It is just displayed by Excel 365 when reading "legacy" formulas. However, it is possible to write it to a formula, if necessary, using `SINGLE()` or `_xlfn.SINGLE()`. The unusual cases where this may be necessary are shown in the linked document in the previous paragraph.

15.5 Dynamic Arrays - The Spilled Range Operator "#"

In the section above on *Dynamic Array support* we saw that dynamic array formulas can return variable sized ranges of results. The Excel documentation refers to this as a "Spilled" range/array from the idea that the results spill into the required number of cells. This is explained in the Microsoft documentation on Dynamic array formulas and spilled array behavior.

Since a spilled range is variable in size a new operator is required to refer to the range. This operator is the Spilled range operator and it is represented by "#". For example, the range F2# in the image below is used to refer to a dynamic array returned by `UNIQUE()` in the cell F2. This example is taken from the XlsxWriter program *Example: Dynamic array formulas*.

Unfortunately, Excel doesn't store the formula like this and in XlsxWriter you need to use the explicit function ANCHORARRAY() to refer to a spilled range. The example in the image above was generated using the following:

```
worksheet9.write('J2', '=COUNTA(ANCHORARRAY(F2))')  # Same as '=COUNTA(F2#)' in Excel.
```

15.6 The Excel 365 LAMBDA() function

Recent versions of Excel 365 have introduced a powerful new function/feature called LAMBDA(). This is similar to the lambda function in Python (and other languages).

Consider the following Excel example which converts the variable `temp` from Fahrenheit to Celsius:

```
LAMBDA(temp, (5/9) * (temp-32))
```

This could be called in Excel with an argument:

```
=LAMBDA(temp, (5/9) * (temp-32))(212)
```

Or assigned to a defined name and called as a user defined function:

```
=ToCelsius(212)
```

This is similar to this example in Python:
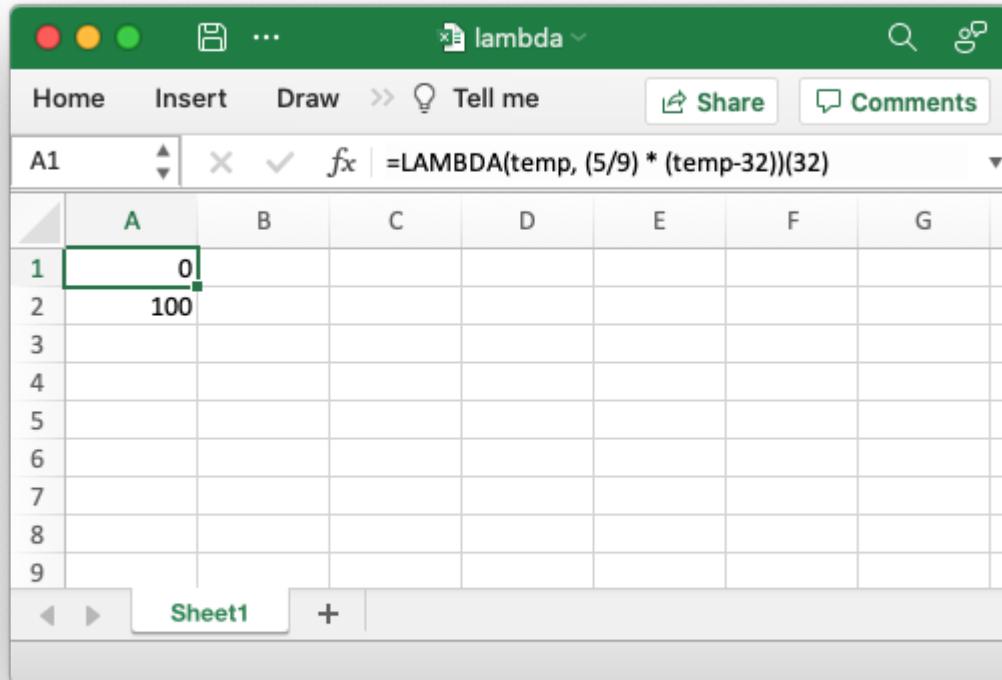
```
>>> to_celsius = lambda temp: (5.0/9.0) * (temp-32)
>>> to_celsius(212)
100.0
```

A XlsxWriter program that replicates the Excel is shown in *Example: Excel 365 LAMBDA() function*.

The formula is written as follows:

```
worksheet.write('A2', '=LAMBDA(_xlpm.temp, (5/9) * (_xlpm.temp-32))(32)')
```

Note, that the parameters in the LAMBDA() function must have a "_xlpm." prefix for compatibility with how the formulas are stored in Excel. These prefixes won't show up in the formula, as shown in the image.

The LET() function is often used in conjunction with LAMBDA() to assign names to calculation results.
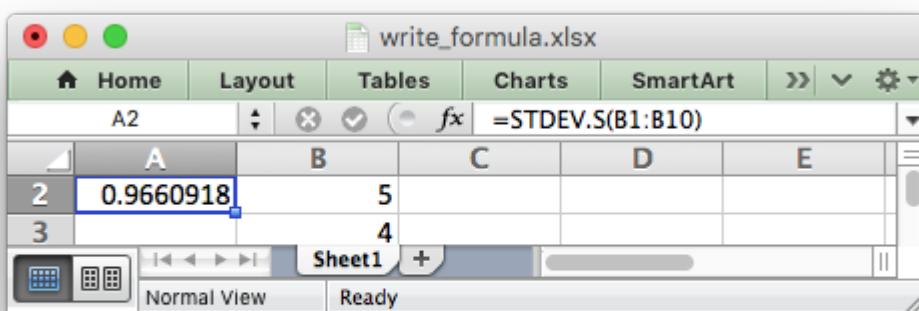
15.7 Formulas added in Excel 2010 and later

Excel 2010 and later added functions which weren't defined in the original file specification. These functions are referred to by Microsoft as *future* functions. Examples of these functions are ACOT, `CHISQ.DIST.RT`, `CONFIDENCE.NORM`, `STDEV.P`, `STDEV.S` and `WORKDAY.INTL`.

When written using `write_formula()` these functions need to be fully qualified with a `_xlfn.` (or other) prefix as they are shown the list below. For example:

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

These functions will appear without the prefix in Excel:

Alternatively, you can enable the `use_future_functions` option in the `Workbook()` constructor, which will add the prefix as required:

```
workbook = Workbook('write_formula.xlsx', {'use_future_functions': True})

# ...

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

If the formula already contains a `_xlfn.` prefix, on any function, then the formula will be ignored and won't be expanded any further.

Note: Enabling the *use_future_functions* option adds an overhead to all formula processing in XlsxWriter. If your application has a lot of formulas or is performance sensitive then it is best to use the explicit `_xlfn.` prefix instead.

The following list is taken from MS XLSX extensions documentation on future functions.

- `_xlfn.ACOTH`
- `_xlfn.ACOT`
- `_xlfn.AGGREGATE`

- _xlfn.ARABIC
- _xlfn.ARRAYTOTEXT
- _xlfn.BASE
- _xlfn.BETA.DIST
- _xlfn.BETA.INV
- _xlfn.BINOM.DIST.RANGE
- _xlfn.BINOM.DIST
- _xlfn.BINOM.INV
- _xlfn.BITAND
- _xlfn.BITLSHIFT
- _xlfn.BITOR
- _xlfn.BITRSHIFT
- _xlfn.BITXOR
- _xlfn.CEILING.MATH
- _xlfn.CEILING.PRECISE
- _xlfn.CHISQ.DIST.RT
- _xlfn.CHISQ.DIST
- _xlfn.CHISQ.INV.RT
- _xlfn.CHISQ.INV
- _xlfn.CHISQ.TEST
- _xlfn.COMBINA
- _xlfn.CONCAT
- _xlfn.CONFIDENCE.NORM
- _xlfn.CONFIDENCE.T
- _xlfn.COTH
- _xlfn.COT
- _xlfn.COVARIANCE.P
- _xlfn.COVARIANCE.S
- _xlfn.CSCH
- _xlfn.CSC
- _xlfn.DAYS
- _xlfn.DECIMAL

- `ECMA.CEILING`
- `_xlfn.ERF.PRECISE`
- `_xlfn.ERFC.PRECISE`
- `_xlfn.EXPON.DIST`
- `_xlfn.F.DIST.RT`
- `_xlfn.F.DIST`
- `_xlfn.F.INV.RT`
- `_xlfn.F.INV`
- `_xlfn.F.TEST`
- `_xlfn.FILTERXML`
- `_xlfn.FLOOR.MATH`
- `_xlfn.FLOOR.PRECISE`
- `_xlfn.FORECAST.ETS.CONFINT`
- `_xlfn.FORECAST.ETS.SEASONALITY`
- `_xlfn.FORECAST.ETS.STAT`
- `_xlfn.FORECAST.ETS`
- `_xlfn.FORECAST.LINEAR`
- `_xlfn.FORMULATEXT`
- `_xlfn.GAMMA.DIST`
- `_xlfn.GAMMA.INV`
- `_xlfn.GAMMALN.PRECISE`
- `_xlfn.GAMMA`
- `_xlfn.GAUSS`
- `_xlfn.HYPGEOM.DIST`
- `_xlfn.IFNA`
- `_xlfn.IFS`
- `_xlfn.IMAGE`
- `_xlfn.IMCOSH`
- `_xlfn.IMCOT`
- `_xlfn.IMCSCH`
- `_xlfn.IMCSC`
- `_xlfn.IMSECH`

- _xlfn.IMSEC
- _xlfn.IMSINH
- _xlfn.IMTAN
- _xlfn.ISFORMULA
- _xlfn.ISOMITTED
- _xlfn.ISOWEEKNUM
- _xlfn.LET
- _xlfn.LOGNORM.DIST
- _xlfn.LOGNORM.INV
- _xlfn.MAXIFS
- _xlfn.MINIFS
- _xlfn.MODE.MULT
- _xlfn.MODE.SNGL
- _xlfn.MUNIT
- _xlfn.NEGBINOM.DIST
- NETWORKDAYS.INTL
- _xlfn.NORM.DIST
- _xlfn.NORM.INV
- _xlfn.NORM.S.DIST
- _xlfn.NORM.S.INV
- _xlfn.NUMBERVALUE
- _xlfn.PDURATION
- _xlfn.PERCENTILE.EXC
- _xlfn.PERCENTILE.INC
- _xlfn.PERCENTRANK.EXC
- _xlfn.PERCENTRANK.INC
- _xlfn.PERMUTATIONA
- _xlfn.PHI
- _xlfn.POISSON.DIST
- _xlfn.QUARTILE.EXC
- _xlfn.QUARTILE.INC
- _xlfn.QUERYSTRING

- `_xlfn.RANK.AVG`
- `_xlfn.RANK.EQ`
- `_xlfn.RRI`
- `_xlfn.SECH`
- `_xlfn.SEC`
- `_xlfn.SHEETS`
- `_xlfn.SHEET`
- `_xlfn.SKEW.P`
- `_xlfn.STDEV.P`
- `_xlfn.STDEV.S`
- `_xlfn.T.DIST.2T`
- `_xlfn.T.DIST.RT`
- `_xlfn.T.DIST`
- `_xlfn.T.INV.2T`
- `_xlfn.T.INV`
- `_xlfn.T.TEST`
- `_xlfn.TEXTAFTER`
- `_xlfn.TEXTBEFORE`
- `_xlfn.TEXTJOIN`
- `_xlfn.UNICHAR`
- `_xlfn.UNICODE`
- `_xlfn.VALUETOTEXT`
- `_xlfn.VAR.P`
- `_xlfn.VAR.S`
- `_xlfn.WEBSERVICE`
- `_xlfn.WEIBULL.DIST`
- `WORKDAY.INTL`
- `_xlfn.XMATCH`
- `_xlfn.XOR`
- `_xlfn.Z.TEST`

The dynamic array functions shown in the *Dynamic Array support* section above are also future functions:

- `_xlfn.ANCHORARRAY`
- `_xlfn.BYCOL`
- `_xlfn.BYROW`
- `_xlfn.CHOOSECOLS`
- `_xlfn.CHOOSEROWS`
- `_xlfn.DROP`
- `_xlfn.EXPAND`
- `_xlfn._xlws.FILTER`
- `_xlfn.HSTACK`
- `_xlfn.LAMBDA`
- `_xlfn.MAKEARRAY`
- `_xlfn.MAP`
- `_xlfn.RANDARRAY`
- `_xlfn.REDUCE`
- `_xlfn.SCAN`
- `_xlfn.SINGLE`
- `_xlfn.SEQUENCE`
- `_xlfn._xlws.SORT`
- `_xlfn.SORTBY`
- `_xlfn.SWITCH`
- `_xlfn.TAKE`
- `_xlfn.TEXTSPLIT`
- `_xlfn.TOCOL`
- `_xlfn.TOROW`
- `_xlfn.UNIQUE`
- `_xlfn.VSTACK`
- `_xlfn.WRAPCOLS`
- `_xlfn.WRAPROWS`
- `_xlfn.XLOOKUP`

However, since these functions are part of a powerful new feature in Excel, and likely to be very important to end users, they are converted automatically from their shorter version to the explicit future function version by XlsxWriter, even without the `use_future_function` option. If you

---

need to override the automatic conversion you can use the explicit versions with the prefixes shown above.

15.8 Using Tables in Formulas

Worksheet tables can be added with XlsxWriter using the `add_table()` method:

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

By default tables are named `Table1`, `Table2`, etc., in the order that they are added. However it can also be set by the user using the `name` parameter:

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

When used in a formula a table name such as `TableX` should be referred to as `TableX[]` (like a Python list):

```
worksheet.write_formula('A5', '=VLOOKUP("Sales", Table1[], 2, FALSE')
```

15.9 Dealing with formula errors

If there is an error in the syntax of a formula it is usually displayed in Excel as #NAME?. Alternatively you may get a warning from Excel when the file is loaded. If you encounter an error like this you can debug it as follows:

1. Ensure the formula is valid in Excel by copying and pasting it into a cell. Note, this should be done in Excel and not other applications such as OpenOffice or LibreOffice since they may have slightly different syntax.

2. Ensure the formula is using comma separators instead of semi-colons, see *Non US Excel functions and syntax* above.

3. Ensure the formula is in English, see *Non US Excel functions and syntax* above.

4. Ensure that the formula doesn't contain an Excel 2010+ future function as listed above (*Formulas added in Excel 2010 and later*). If it does then ensure that the correct prefix is used.

5. If the function loads in Excel but appears with one or more @ symbols added then it is probably an array function and should be written using `write_array_formula()` or `write_dynamic_array_formula()` (see the sections above on *Dynamic Array support* and *Dynamic Arrays - The Implicit Intersection Operator "@"*).

Finally if you have completed all the previous steps and still get a #NAME? error you can examine a valid Excel file to see what the correct syntax should be. To do this you should create a valid formula in Excel and save the file. You can then examine the XML in the unzipped file.

The following shows how to do that using Linux `unzip` and libxml's xmllint to format the XML for clarity:

```
$ unzip myfile.xlsx -d myfile
$ xmllint --format myfile/xl/worksheets/sheet1.xml | grep '</f>'
```

```
<f>SUM(1, 2, 3)</f>
```

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:

```python
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
```
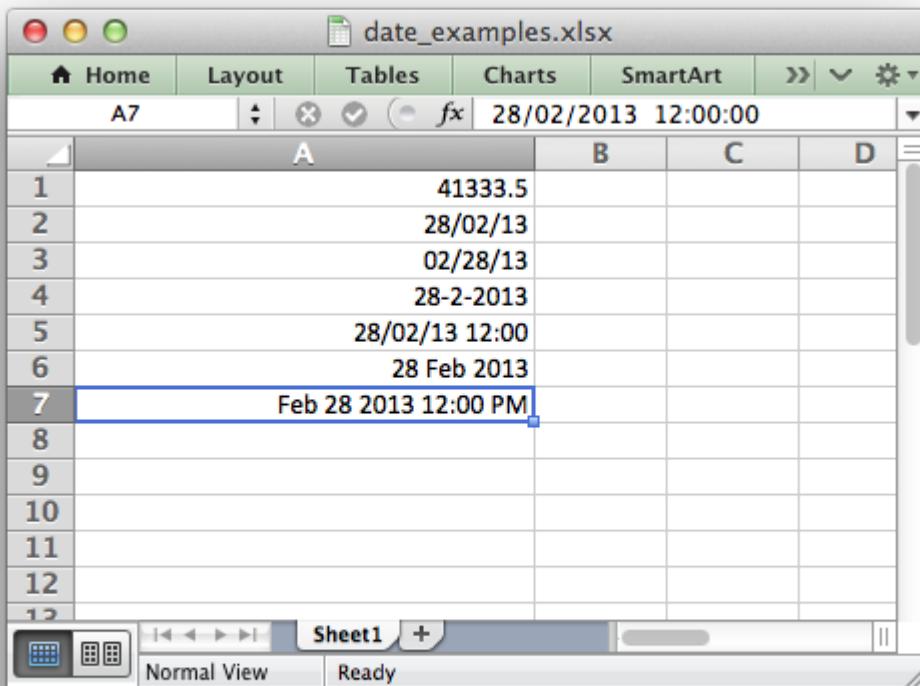
```
workbook.close()
```


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()
```


16.1 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()
```

16.2 Timezone Handling

Excel doesn't support timezones in datetimes/times so there isn't any fail-safe way that XlsxWriter can map a Python timezone aware datetime into an Excel datetime. As such the user should handle the timezones in some way that makes sense according to their requirements. Usually this will require some conversion to a timezone adjusted time and the removal of the `tzinfo` from the datetime object so that it can be passed to `write_datetime()`:

```python
utc_datetime = datetime(2016, 9, 23, 14, 13, 21, tzinfo=utc)
naive_datetime = utc_datetime.replace(tzinfo=None)

worksheet.write_datetime(row, 0, naive_datetime, date_format)
```

Alternatively the `Workbook()` constructor option `remove_timezone` can be used to strip the timezone from datetime values passed to `write_datetime()`. The default is `False`. To enable this option use:

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

When *Working with Pandas and XlsxWriter* you can pass the argument as follows:

```python
writer = pd.ExcelWriter('pandas_example.xlsx',
                        engine='xlsxwriter',
                        options={'remove_timezone': True})
```

SEVENTEEN

WORKING WITH COLORS

Throughout XlsxWriter colors are specified using a Html style #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

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:
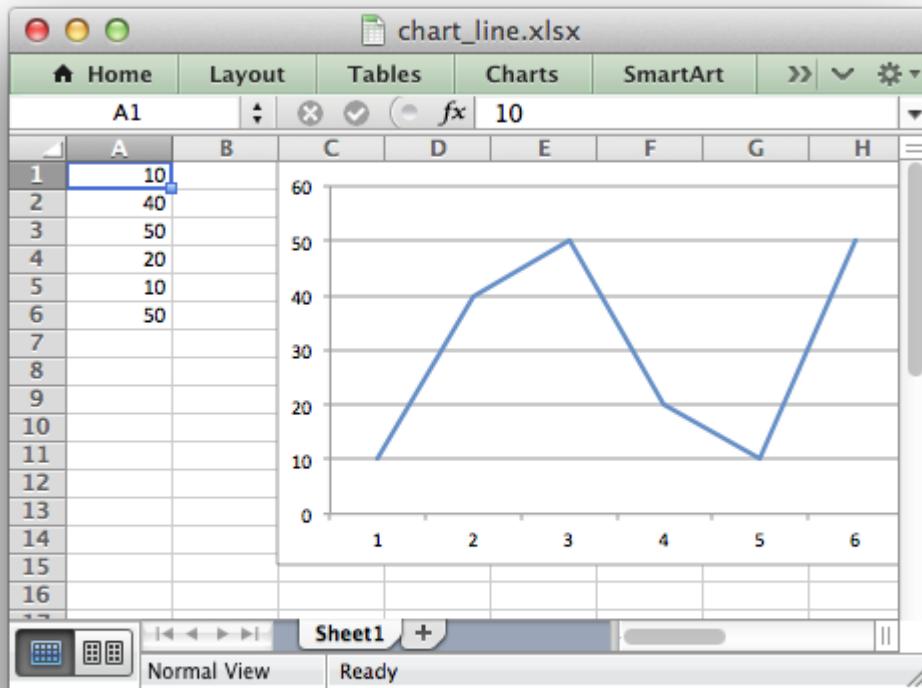
```python
import xlsxwriter

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()
```
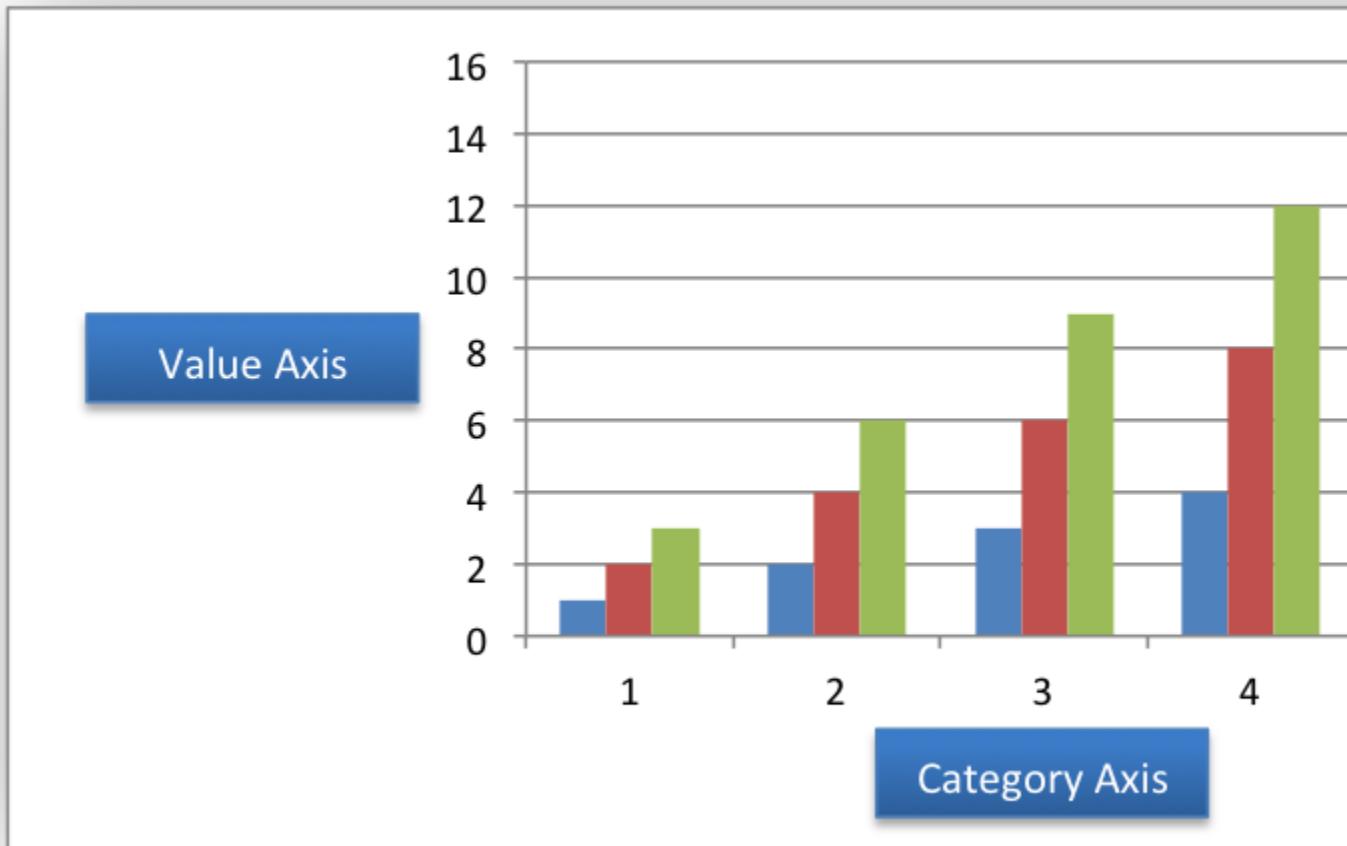
See also *Chart Examples*.
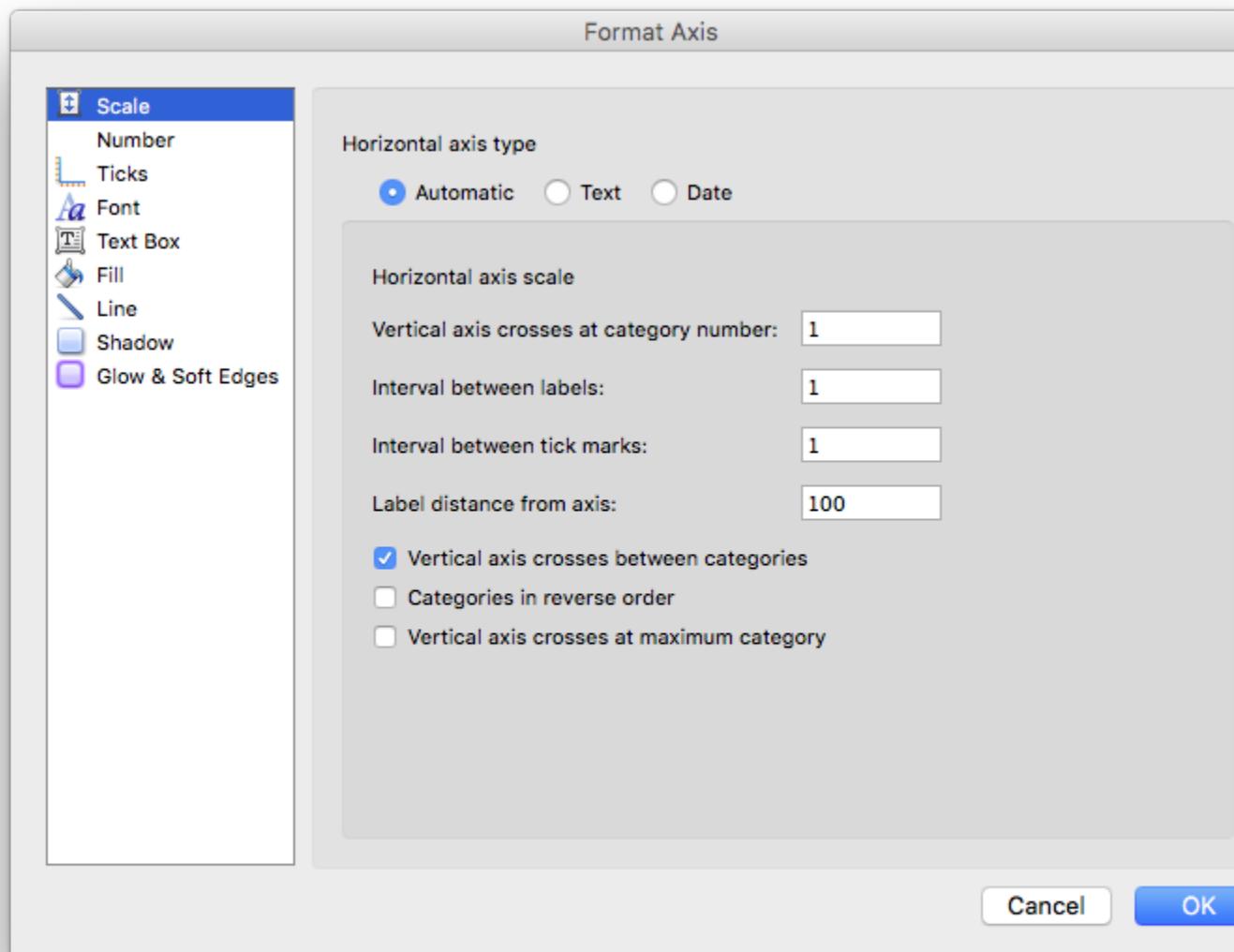
18.1 Chart Value and Category Axes

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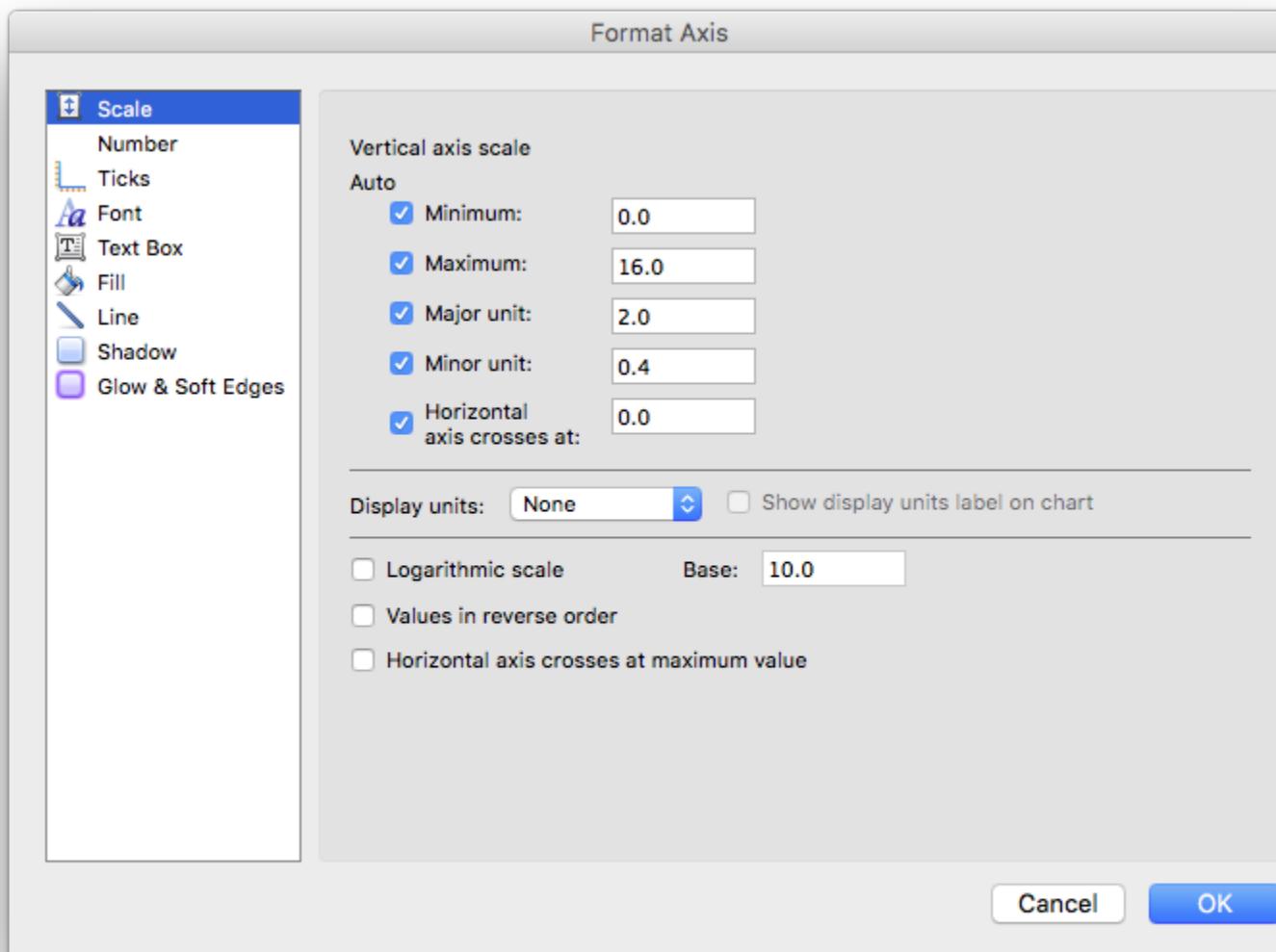
In the majority of Excel charts the X axis is the **category** axis and each of the values is evenly spaced and sequential. The Y axis is the **value** axis and points are displayed according to their value:

Excel treats these two types of axis differently and exposes different properties for each. For example, here are the properties for a category axis:
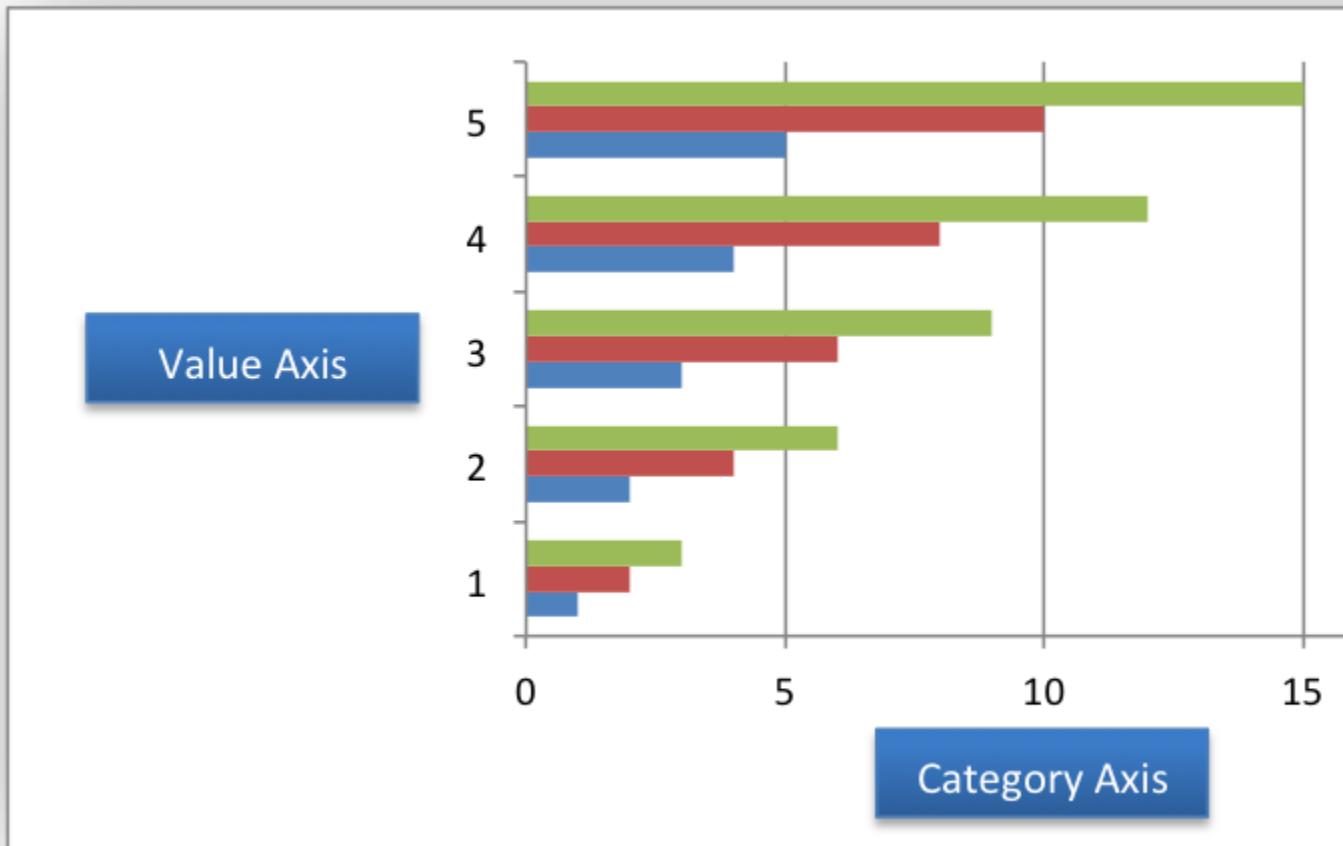
Here are properties for a value axis:
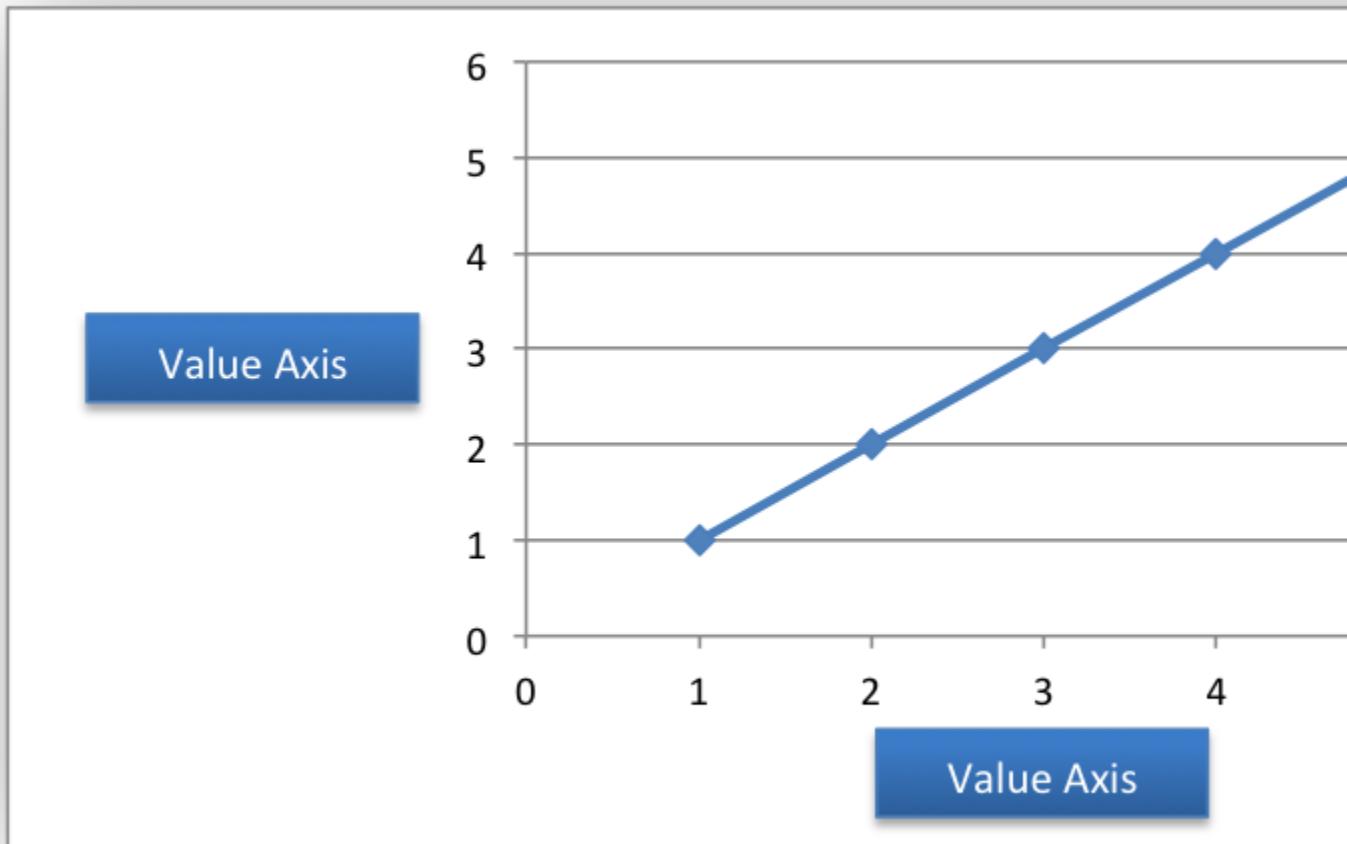
As such, some of the *XlsxWriter* axis properties can be set for a value axis, some can be set for a category axis and some properties can be set for both. For example `reverse` can be set for either category or value axes while the `min` and `max` properties can only be set for value axes (and Date Axes). The documentation calls out the type of axis to which properties apply.

For a Bar chart the Category and Value axes are reversed:

A Scatter chart (but not a Line chart) has 2 value 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.

18.2 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
```

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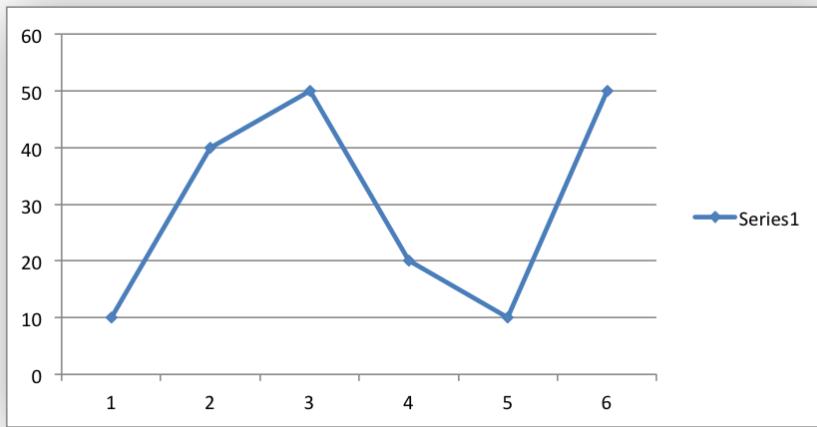
The marker format specifies the properties of the markers used to distinguish series on a chart. In general only Line and Scatter chart types and trendlines use markers.

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

```
type
size
border
fill
pattern
gradient
```

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

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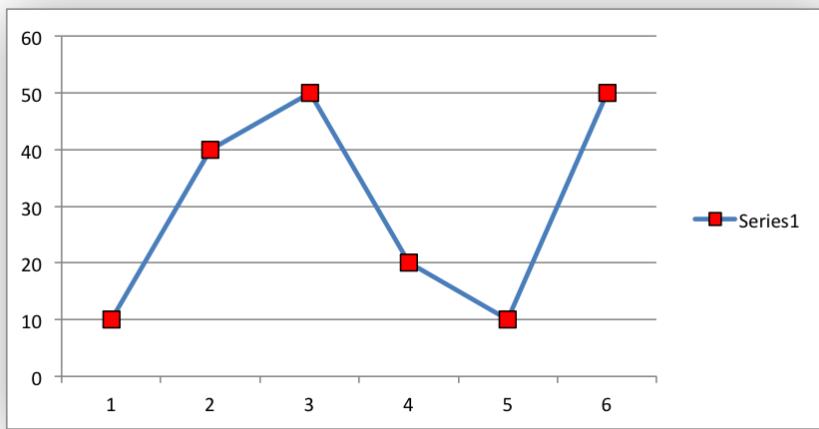
If `automatic` is on then other marker properties such as size, border or fill cannot be set.

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'},
    },
})
```


18.4 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
intercept            (for exponential, linear and polynomial only)
display_equation     (for all except moving average)
display_r_squared    (for all except moving average)
```

The `type` property sets the type of trendline in the series:
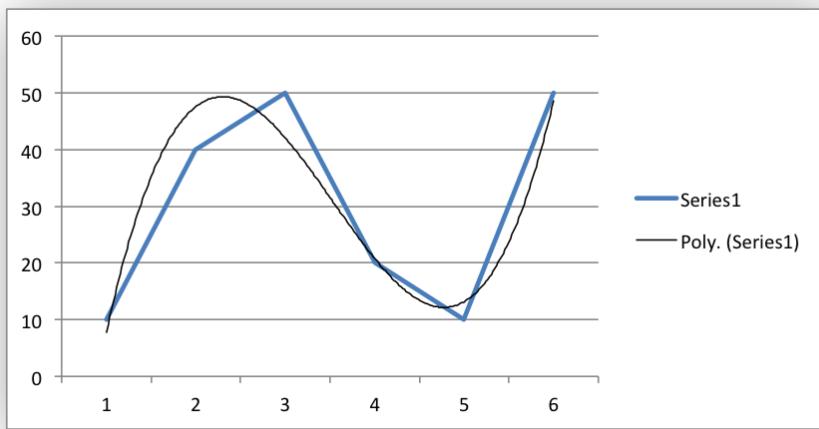
```python
chart.add_series({
    'values':    '=Sheet1!$A$1:$A$6',
    'trendline': {'type': 'linear'},
})
```

The available `trendline` types are:

```
exponential
linear
log
moving_average
polynomial
power
```
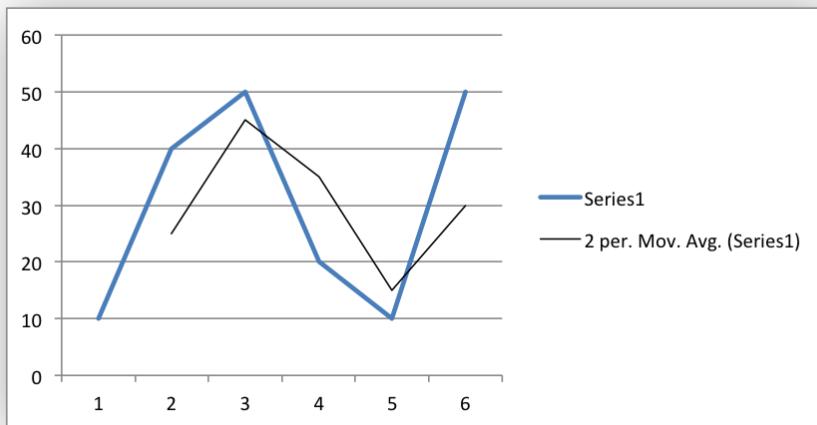
A `polynomial` trendline can also specify the `order` of the polynomial. The default value is 2:

```python
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',
        'order': 2,
        'forward': 0.5,
        'backward': 0.5,
    },
})
```

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',
        'name': 'My trend name',
        'order': 2,
    },
})
```

18.4. Chart series option: Trendline   **247**

The `intercept` property sets the point where the trendline crosses the Y (value) axis:

```python
chart.add_series({
    'values': '=Sheet1!$B$1:$B$6',
    'trendline': {'type': 'linear',
                  'intercept': 0.8,
    },
})
```

The `display_equation` property displays the trendline equation on the chart:
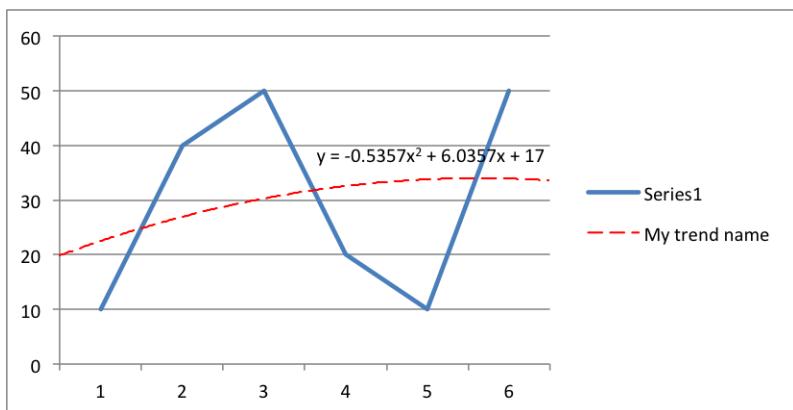
```python
chart.add_series({
    'values': '=Sheet1!$B$1:$B$6',
    'trendline': {'type': 'linear',
                  'display_equation': True,
    },
})
```

The `display_r_squared` property displays the R squared value of the trendline on the chart:

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

Several of these properties can be set in one go:

```python
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'trendline': {
        'type': 'polynomial',
        'name': 'My trend name',
        'order': 2,
        'forward': 0.5,
        'backward': 0.5,
        'display_equation': True,
        '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.

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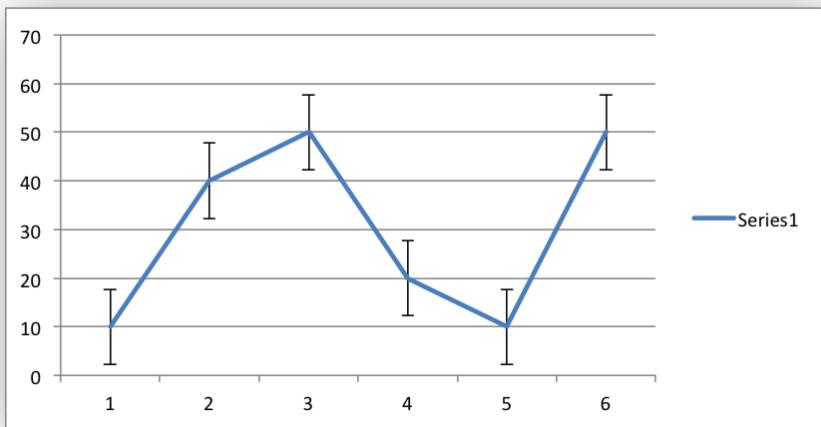
Error bars can be added to a chart series to indicate error bounds in the data. The error bars can be vertical `y_error_bars` (the most common type) or horizontal `x_error_bars` (for Bar and Scatter charts only).

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$6 type range formula or a list of values:

```
chart.add_series({
    'categories': '=Sheet1!$A$1:$A$6',
    'values':     '=Sheet1!$B$1:$B$6',
    'y_error_bars': {
        'type':          'custom',
        'plus_values':   '=Sheet1!$C$1:$C$6',
        'minus_values': '=Sheet1!$D$1:$D$6',
    },
})

# or

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

```
        'values':      '=Sheet1!$B$1:$B$6',
        '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.
```
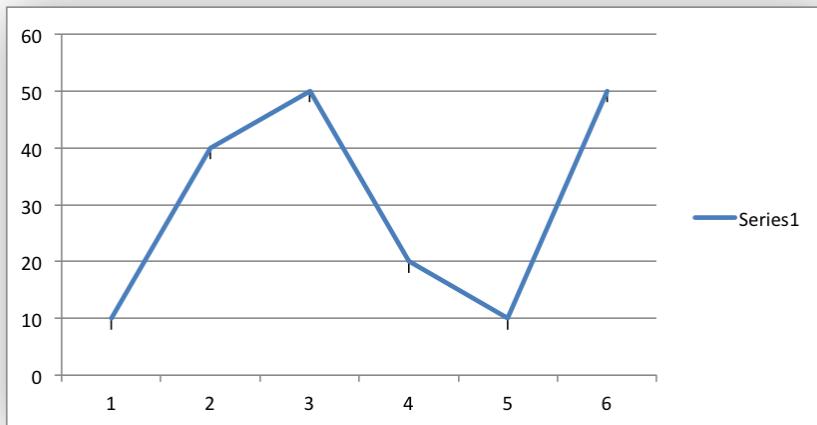
The `end_style` property sets the style of the error bar end cap. The options are 1 (the default) or 0 (for no end cap):

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


18.6 Chart series option: Data Labels
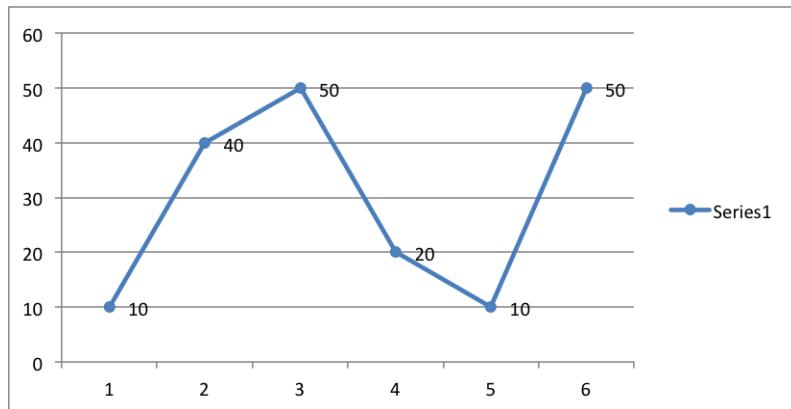
Data labels can be added to a chart series to indicate the values of the plotted data points.

The following properties can be set for `data_labels` formats in a chart:
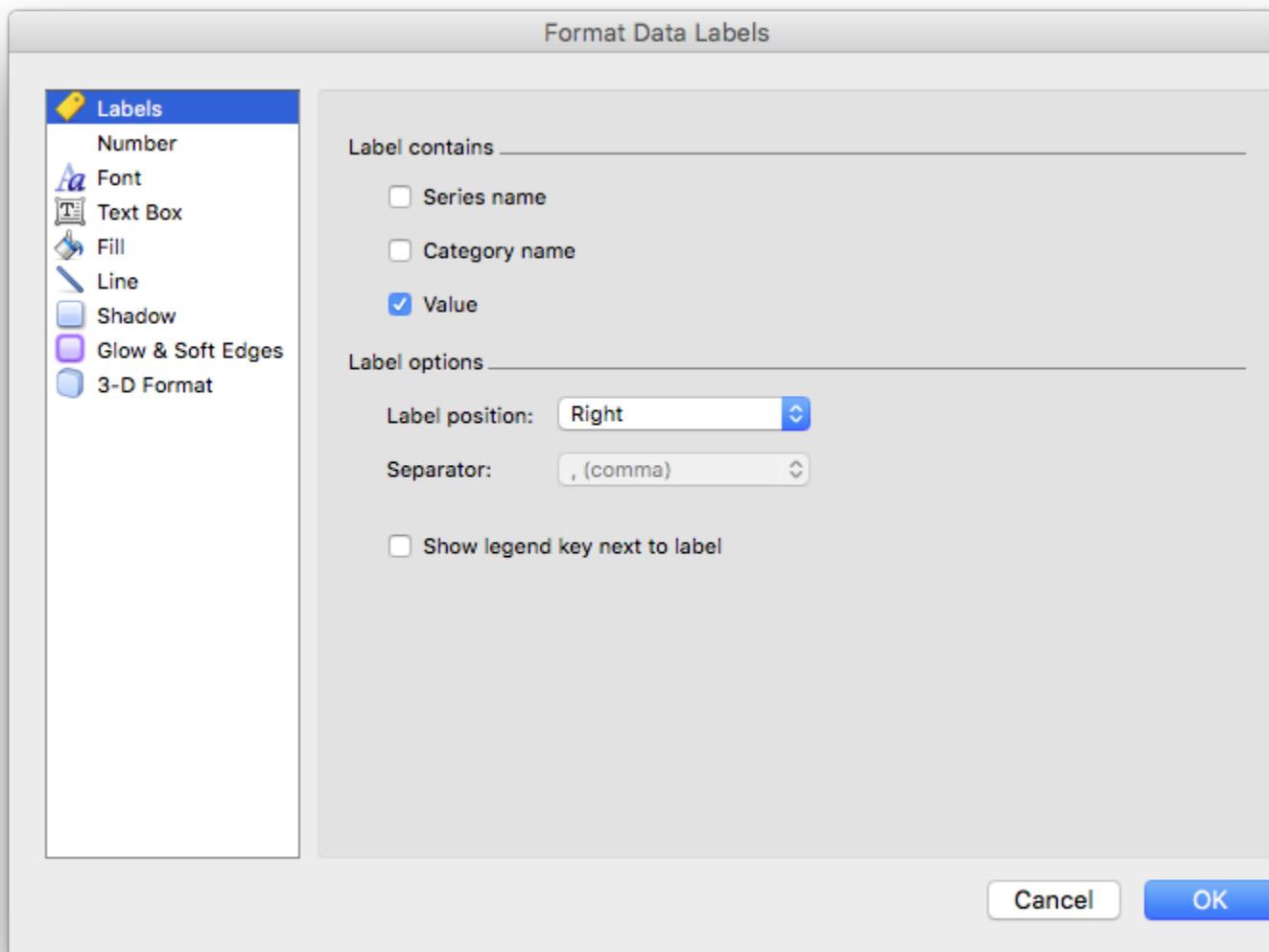
```
value
category
series_name
position
leader_lines
percentage
separator
legend_key
num_format
font
border
fill
pattern
gradient
custom
```

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},
})
```


By default data labels are displayed in Excel with only the values shown. However, it is possible to configure other display options, as shown below.
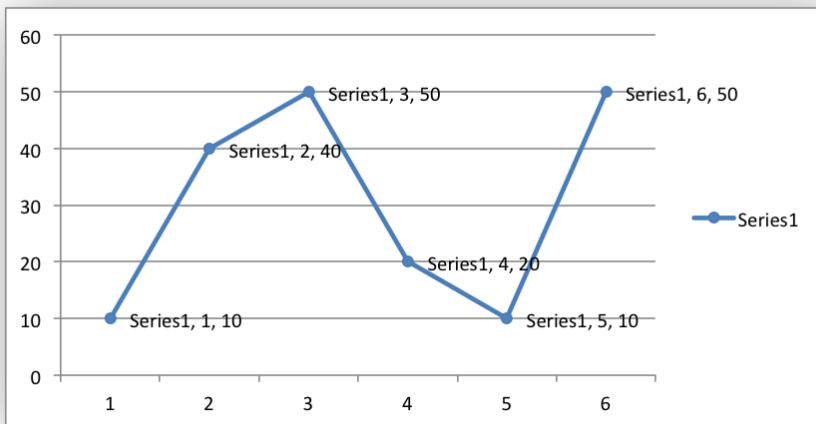
The `category` property turns on the *Category Name* data label for a series:

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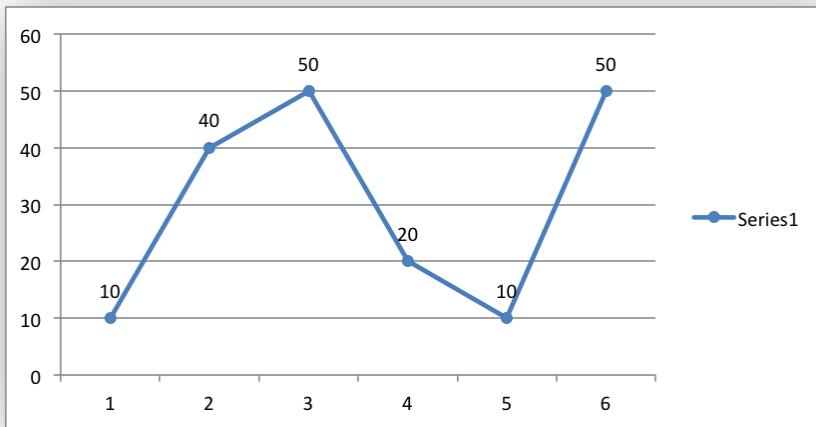
```
chart.add_series({
    'values':      '=Sheet1!$A$1:$A$6',
    'data_labels': {'series_name': True},
})
```

Here is an example with all three data label types shown:

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': 'above'},
})
```
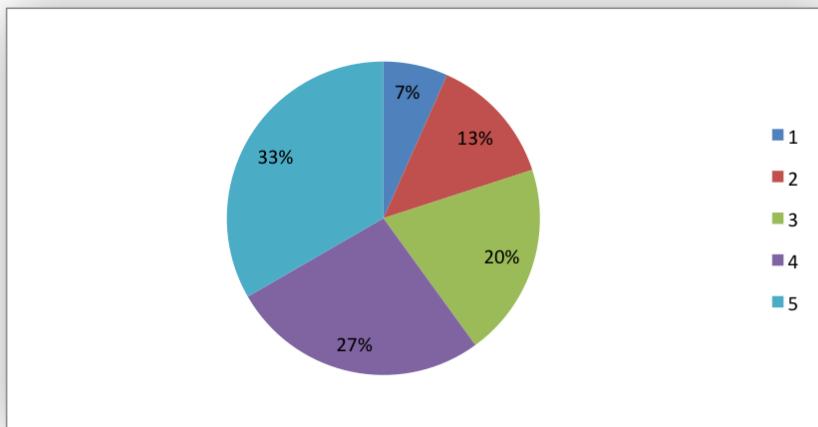

In Excel the allowable data label positions vary for different chart types. The allowable positions are:

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 by the user.

The `percentage` property is used to turn on the display of data labels as a *Percentage* for a series. In Excel the `percentage` data label option is only available for Pie and Doughnut chart variants:

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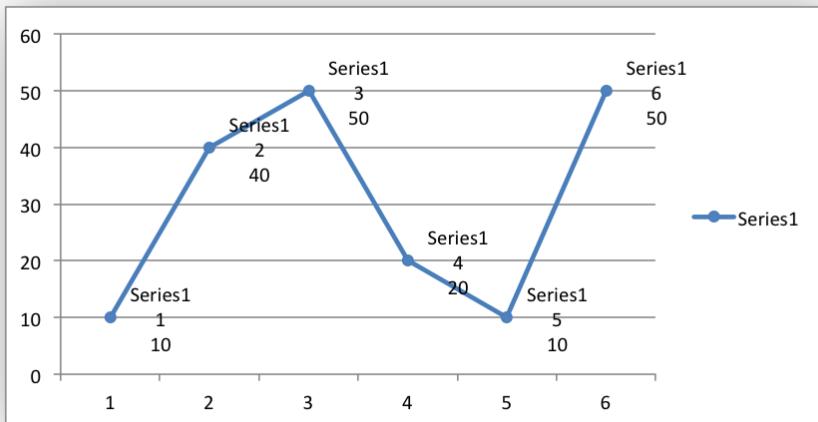

The `leader_lines` property is used to turn on *Leader Lines* for the data label of a series:

```
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 Xl-sxWriter. 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:

18.6. Chart series option: Data Labels

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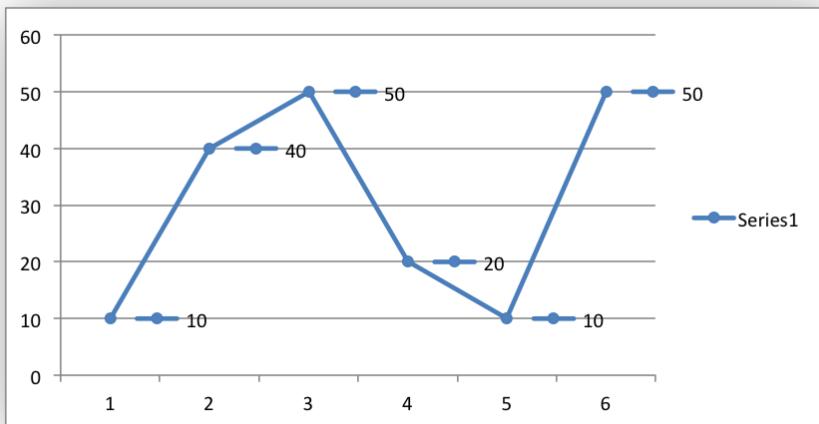

The separator value must be one of the following strings:

```
' '
';'
'.'
'\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$6',
    'data_labels': {'value': True, 'num_format': '#,##0.00'},
})
```
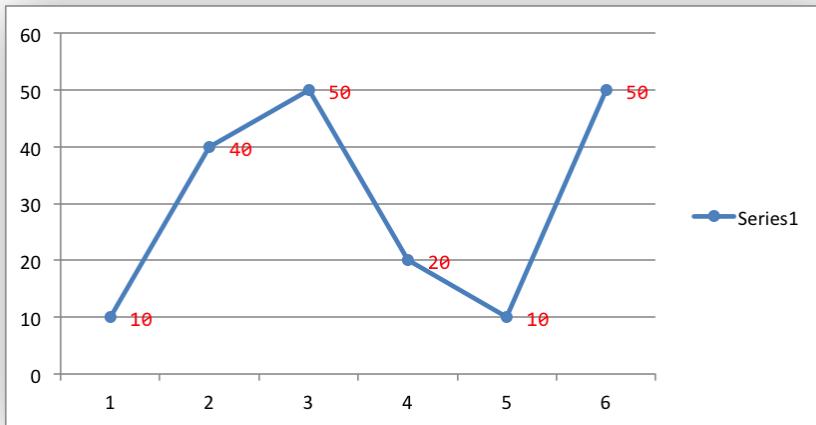

The number format is similar to the Worksheet Cell Format `num_format` apart from the fact that a format index cannot be used. An explicit format string must be used as shown above. See `set_num_format()` for more information.

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

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'data_labels': {
        'value': True,
        'font': {'name': 'Consolas', 'color': 'red'}
    },
```

```
    })
```
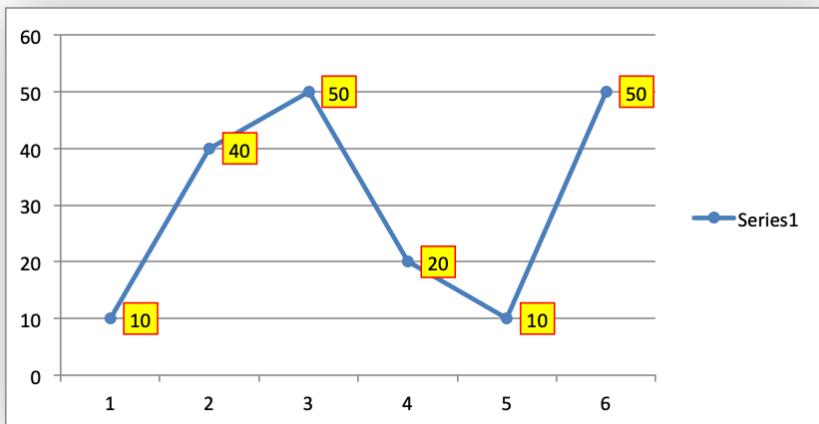

The `font` property is also used to rotate the data labels of a series:

```
    chart.add_series({

        'values': '=Sheet1!$A$1:$A$6',
        'data_labels': {
            'value': True,
            'font': {'rotation': 45}
        },
    })
```

See *Chart Fonts*.

Standard chart formatting such as `border` and `fill` can also be added to the data labels:

```
    chart.add_series({
        'values':      '=Sheet1!$A$1:$A$6',
        'data_labels': {'value': True,
                        'border': {'color': 'red'},
                        'fill':   {'color': 'yellow'}},
    })
```
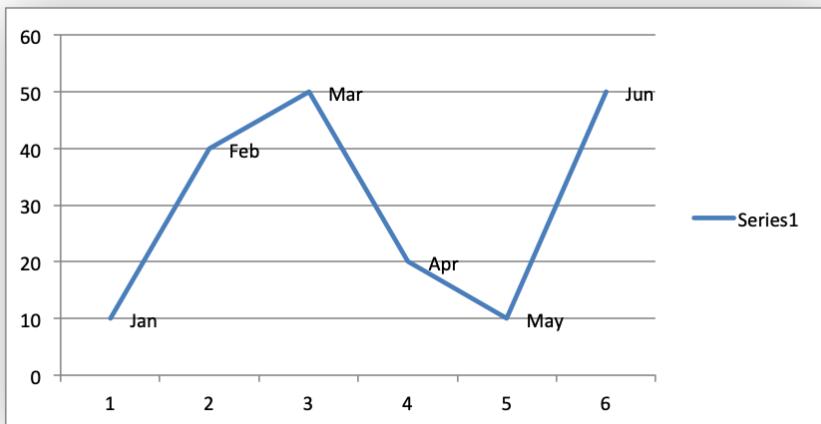
The `custom` property is used to set properties for individual data labels. This is explained in detail in the next section.

18.7 Chart series option: Custom Data Labels

The `custom` data label property is used to set the properties of individual data labels in a series. The most common use for this is to set custom text or number labels:

```python
custom_labels = [
    {'value': 'Jan'},
    {'value': 'Feb'},
    {'value': 'Mar'},
    {'value': 'Apr'},
    {'value': 'May'},
    {'value': 'Jun'},
]

chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'data_labels': {'value': True, 'custom': custom_labels},
})
```
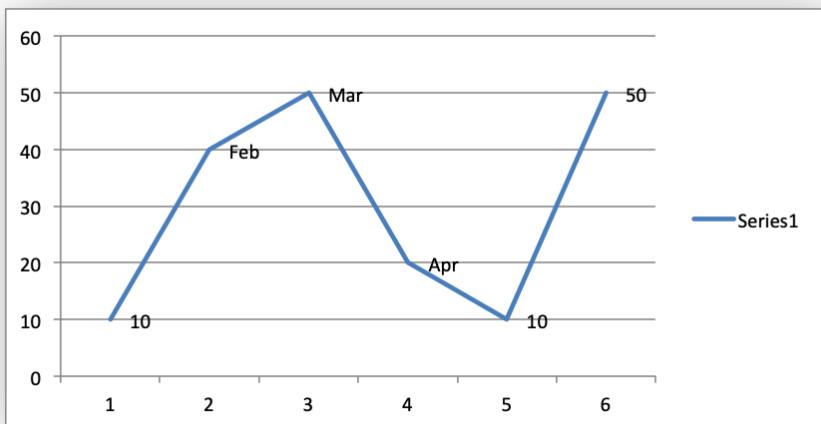
As shown in the previous examples th `custom` property should be a list of dicts. Any property dict that is set to None or not included in the list will be assigned the default data label value:

```
custom_labels = [
    None,
    {'value': 'Feb'},
    {'value': 'Mar'},
    {'value': 'Apr'},
]

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


The property elements of the `custom` lists should be dicts with the following allowable keys/sub-properties:

```
value
font
delete
```

The `value` property should be a string, number or formula string that refers to a cell from which the value will be taken:
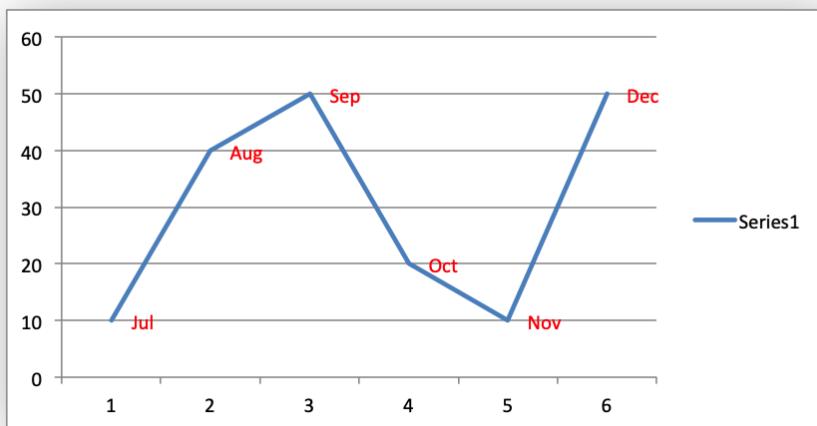
```
custom_labels = [
    {'value': '=Sheet1!$C$1'},
    {'value': '=Sheet1!$C$2'},
    {'value': '=Sheet1!$C$3'},
    {'value': '=Sheet1!$C$4'},
    {'value': '=Sheet1!$C$5'},
    {'value': '=Sheet1!$C$6'},
]
```

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

```
custom_labels = [
    {'value': '=Sheet1!$C$1', 'font': {'color': 'red'}},
    {'value': '=Sheet1!$C$2', 'font': {'color': 'red'}},
    {'value': '=Sheet1!$C$3', 'font': {'color': 'red'}},
    {'value': '=Sheet1!$C$4', 'font': {'color': 'red'}},
    {'value': '=Sheet1!$C$5', 'font': {'color': 'red'}},
    {'value': '=Sheet1!$C$6', 'font': {'color': 'red'}},
]

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

See *Chart Fonts* for details on the available font properties.
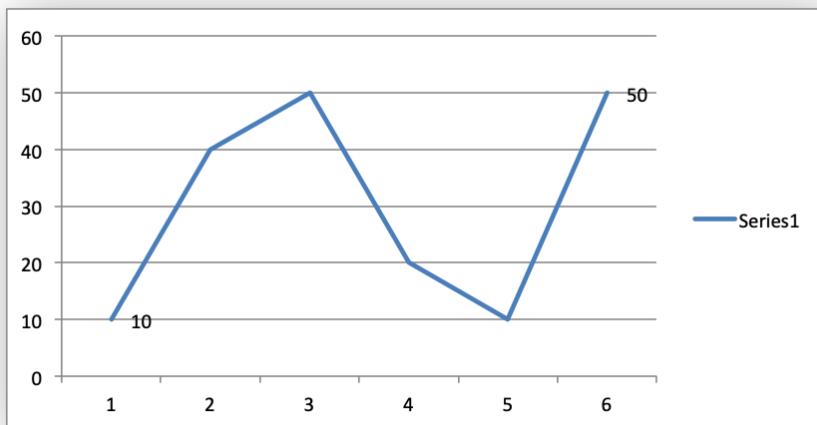
The `delete` property can be used to delete labels in a series. This can be useful if you want to highlight one or more cells in the series, for example the maximum and the minimum:

18.7. Chart series option: Custom Data Labels 261

```python
custom_labels = [
    None,
    {'delete': True},
    {'delete': True},
    {'delete': True},
    {'delete': True},
    None,
]

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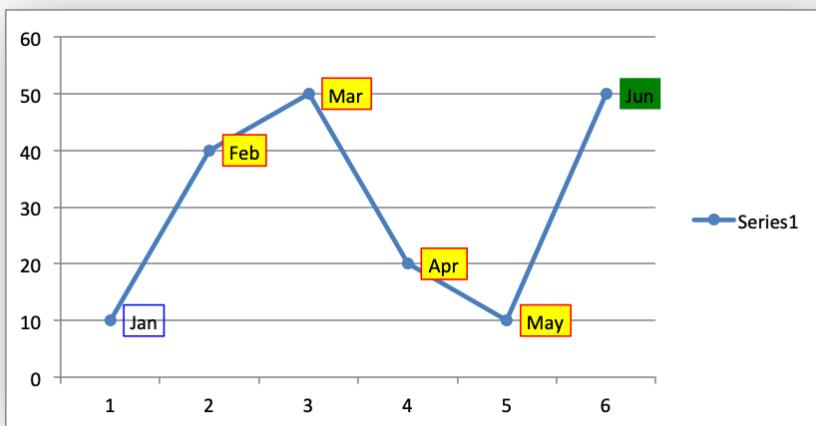

Standard chart formatting such as `border` and `fill` can also be added to the custom data labels:

```python
custom_labels = [
    {'value': 'Jan', 'border': {'color': 'blue'}},
    {'value': 'Feb'},
    {'value': 'Mar'},
    {'value': 'Apr'},
    {'value': 'May'},
    {'value': 'Jun', 'fill':   {'color': 'green'}},
]

chart.add_series({
    'values':       '=Sheet1!$A$1:$A$6',
    'marker': {'type': 'circle'},
    'data_labels': {'value': True,
                    'custom': custom_labels,
                    'border': {'color': 'red'},
                    'fill':   {'color': 'yellow'}},
})
```

18.8 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/Doughnut charts where each segment is represented by a point.

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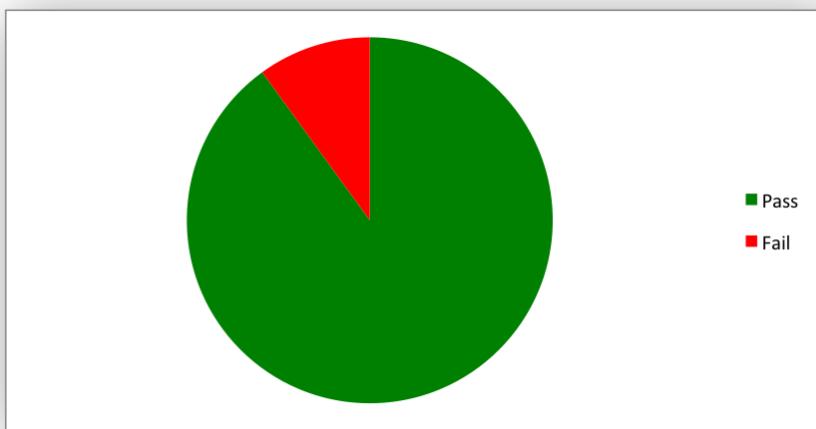
```
import xlsxwriter

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:

```python
# 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'}},
    ],
})
```

18.9 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:

```python
chart.add_series({
    'categories': '=Sheet1!$A$1:$A$6',
    'values':     '=Sheet1!$B$1:$B$6',
    'smooth':     True,
})
```
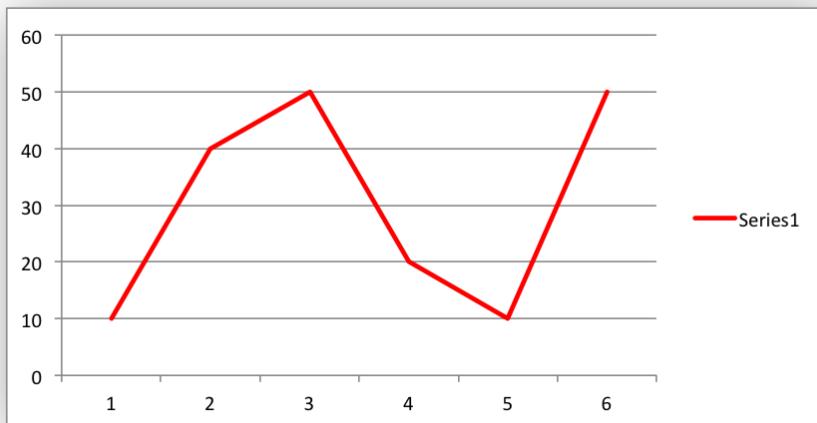
18.10 Chart Formatting

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
pattern
gradient
```
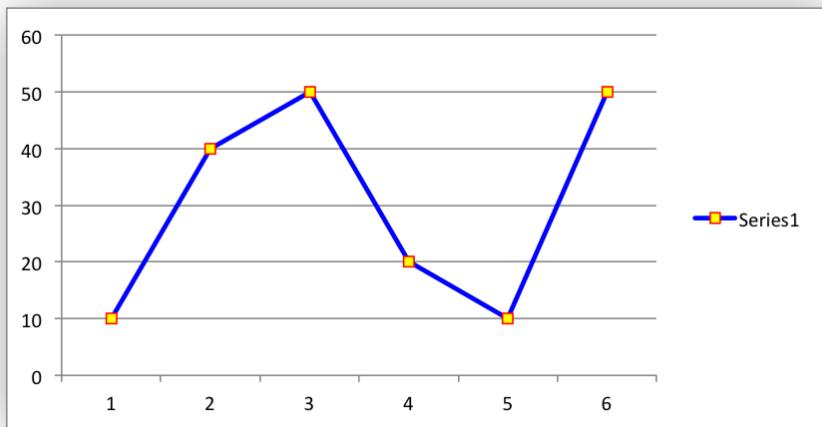
Chart formatting properties are generally set using dicts:

```
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'}
    },
})
```

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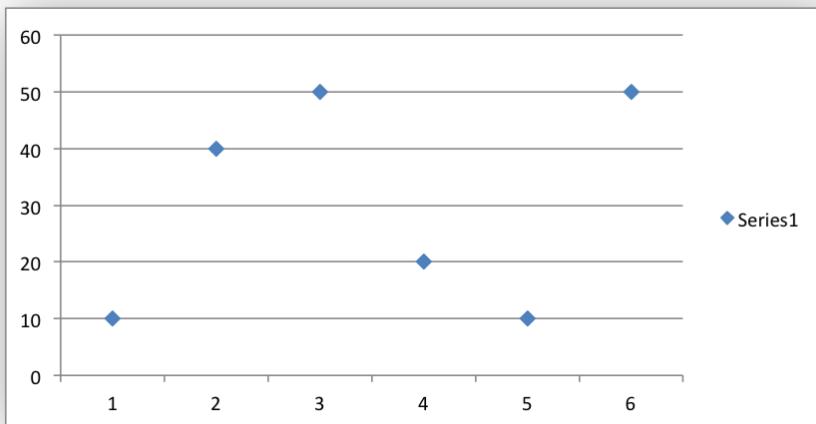
The line format is used to specify properties of line objects that appear in a chart such as a plotted line on a chart or a border.

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

```
none
color
width
dash_type
transparency
```

The none property is used 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:

```python
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'line':   {'none': True},
    'marker': {'type': 'automatic'},
})
```
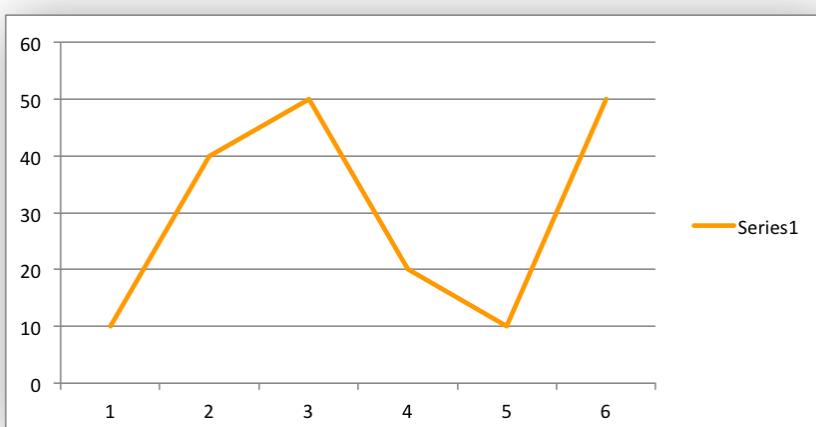
The `color` property sets the color of the `line`:

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

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

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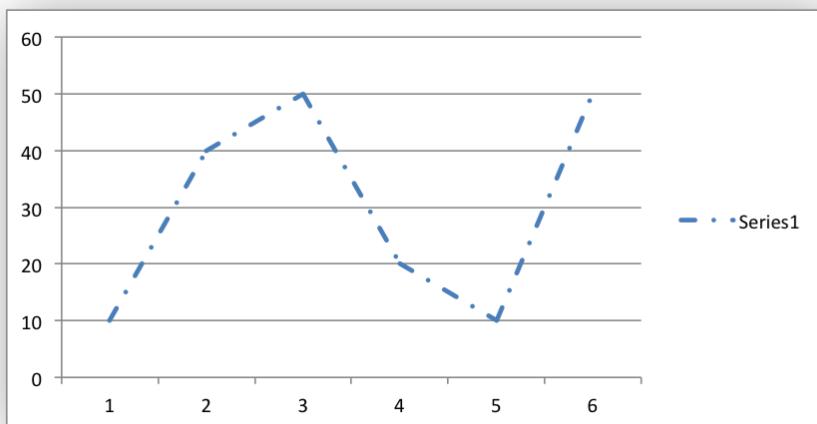

The `width` property sets the width of the `line`. It should be specified in increments of 0.25 of a point as in Excel:

```
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`.

The `transparency` property sets the transparency of the line color in the integer range 1 - 100. The color must be set for transparency to work, it doesn't work with an automatic/default color:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'line':   {'color': 'yellow', 'transparency': 50},
})
```

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',
    },
})
```

18.12 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.

18.13 Chart formatting: Solid Fill
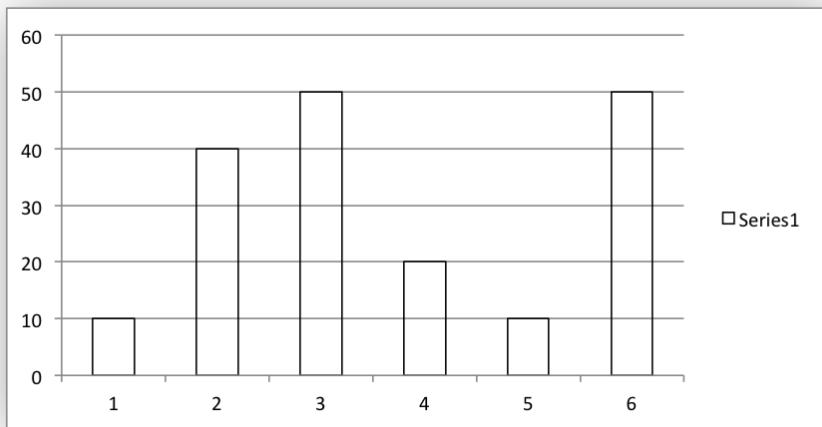
The solid fill format is used to specify filled areas of chart objects such as the interior of a column or the background of the chart itself.

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

```
none
color
transparency
```

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'}
})
```
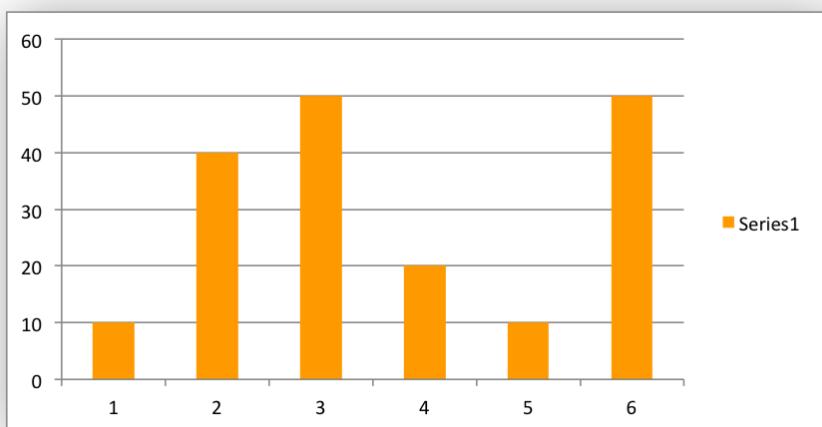
The `color` property sets the color of the `fill` area:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'fill':   {'color': 'red'}
})
```
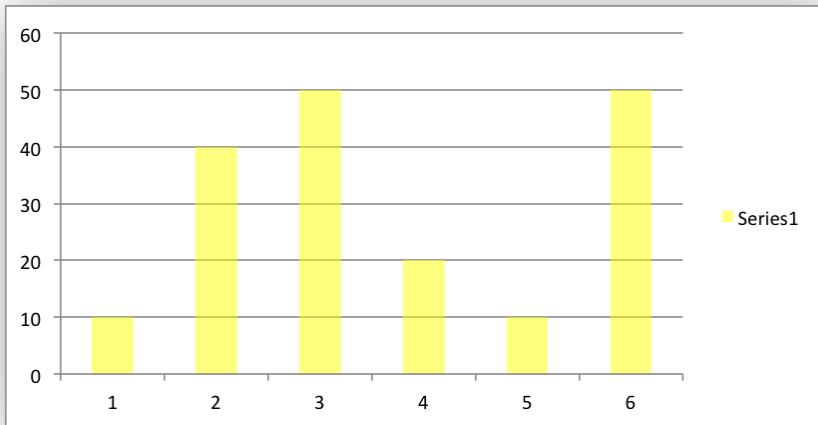
The available colors are shown in the main XlsxWriter documentation. It is also possible to set the color of a fill with a Html style #RRGGBB string or a limited number of named colors, see *Working with Colors*:

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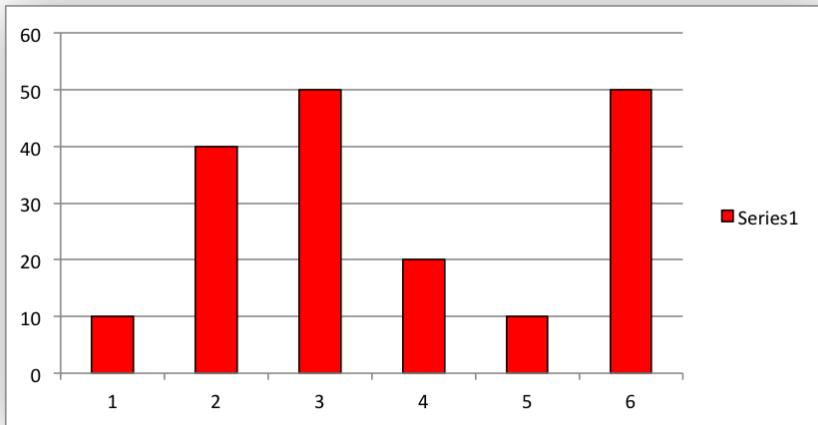

The `transparency` property sets the transparency of the solid fill color in the integer range 1 - 100. The color must be set for transparency to work, it doesn't work with an automatic/default color:

```
chart.set_chartarea({'fill': {'color': 'yellow', 'transparency': 50}})
```
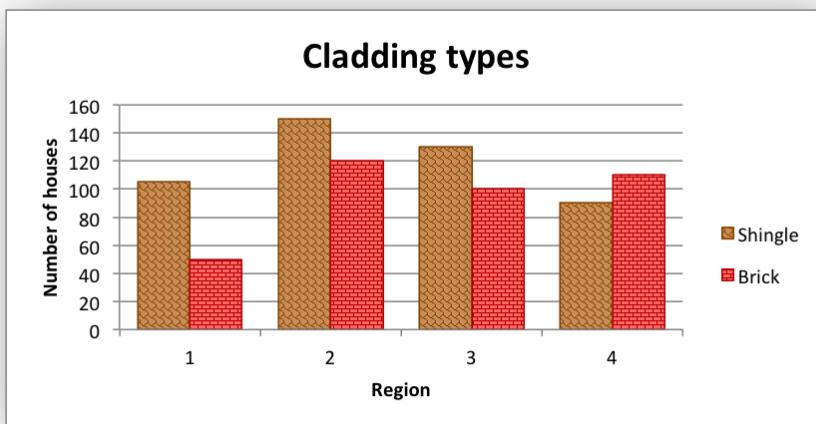

The `fill` format is generally used in conjunction with a `border` format which has the same properties as a `line` format:

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


18.14 Chart formatting: Pattern Fill

The pattern fill format is used to specify pattern 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 `pattern` fill formats in a chart:

```
pattern:   the pattern to be applied (required)
fg_color:  the foreground color of the pattern (required)
bg_color:  the background color (optional, defaults to white)
```

For example:

```
chart.set_plotarea({
    'pattern': {
        'pattern': 'percent_5',
        'fg_color': 'red',
        'bg_color': 'yellow',
    }
})
```

The following patterns can be applied:

- `percent_5`
- `percent_10`
- `percent_20`
- `percent_25`
- `percent_30`
- `percent_40`
- `percent_50`
- `percent_60`
- `percent_70`
- `percent_75`
- `percent_80`

- percent_90
- light_downward_diagonal
- light_upward_diagonal
- dark_downward_diagonal
- dark_upward_diagonal
- wide_downward_diagonal
- wide_upward_diagonal
- light_vertical
- light_horizontal
- narrow_vertical
- narrow_horizontal
- dark_vertical
- dark_horizontal
- dashed_downward_diagonal
- dashed_upward_diagonal
- dashed_horizontal
- dashed_vertical
- small_confetti
- large_confetti
- zigzag
- wave
- diagonal_brick
- horizontal_brick
- weave
- plaid
- divot
- dotted_grid
- dotted_diamond
- shingle
- trellis
- sphere
- small_grid

- large_grid
- small_check
- large_check
- outlined_diamond
- solid_diamond
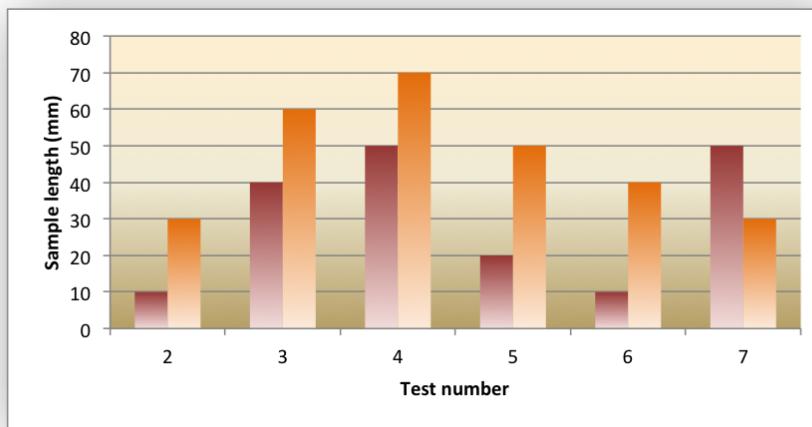
The foreground color, `fg_color`, is a required parameter and can be a Html style #RRGGBB string or a limited number of named colors, see *Working with Colors*.

The background color, `bg_color`, is optional and defaults to white.

If a pattern fill is used on a chart object it overrides the solid fill properties of the object.

18.15 Chart formatting: Gradient Fill

The gradient fill format is used to specify gradient 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 `gradient` fill formats in a chart:

```
colors:    a list of colors
positions: an optional list of positions for the colors
type:      the optional type of gradient fill
angle:     the optional angle of the linear fill
```

The `colors` property sets a list of colors that define the `gradient`:

```
chart.set_plotarea({
    'gradient': {'colors': ['#FFEFD1', '#F0EBD5', '#B69F66']}
})
```

Excel allows between 2 and 10 colors in a gradient but it is unlikely that you will require more than 2 or 3.

As with solid or pattern fill it is also possible to set the colors of a gradient with a Html style #RRGGBB string or a limited number of named colors, see *Working with Colors*:

```
chart.add_series({
    'values':    '=Sheet1!$A$1:$A$6',
    'gradient': {'colors': ['red', 'green']}
})
```

The `positions` defines an optional list of positions, between 0 and 100, of where the colors in the gradient are located. Default values are provided for `colors` lists of between 2 and 4 but they can be specified if required:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'gradient': {
        'colors':    ['#DDEBCF', '#156B13'],
        'positions': [10,        90],
    }
})
```

The `type` property can have one of the following values:

```
linear         (the default)
radial
rectangular
path
```

For example:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'gradient': {
        'colors': ['#DDEBCF', '#9CB86E', '#156B13'],
        'type': 'radial'
    }
})
```

If `type` isn't specified it defaults to `linear`.

For a `linear` fill the angle of the gradient can also be specified:

```
chart.add_series({
    'values': '=Sheet1!$A$1:$A$6',
    'gradient': {'colors': ['#DDEBCF', '#9CB86E', '#156B13'],
                 'angle': 45}
})
```

The default angle is 90 degrees.

If gradient fill is used on a chart object it overrides the solid fill and pattern fill properties of the object.

18.16 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* section 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 integer range -90 to 90 deg, and 270-271 deg:

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

    The font rotation angle is useful for displaying axis data such as dates in a more compact format.

    There are 2 special case angles outside the range -90 to 90:

    - 270: Stacked text, where the text runs from top to bottom.

    - 271: A special variant of stacked text for East Asian fonts.

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

```
        chart.set_x_axis({'num_font': {'color': 'red' }})
        chart.set_y_axis({'num_font': {'color': '#92D050'}})
```
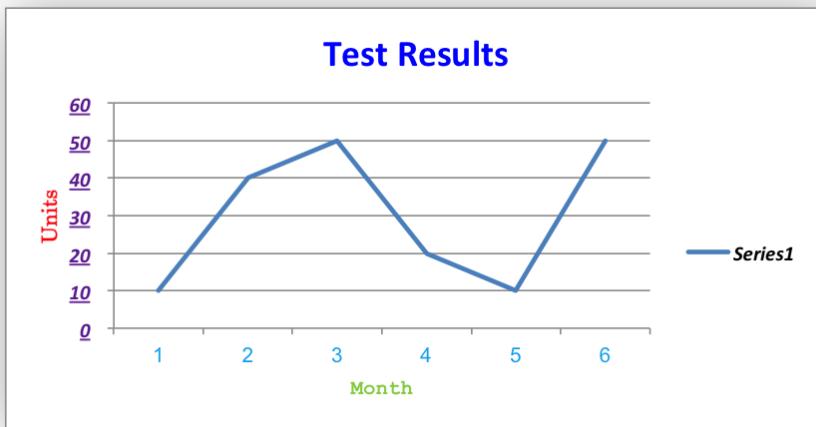
Here is an example of Font formatting in a Chart program:

```
    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}})
```

18.17 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,
        }
    })
```
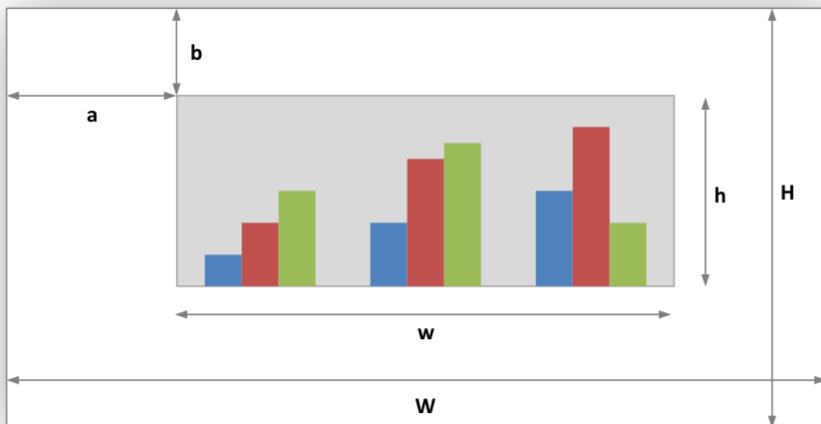
See set_plotarea(), set_legend(), set_title() and set_x_axis(),

---

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.

18.18 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*.

18.19 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()
```
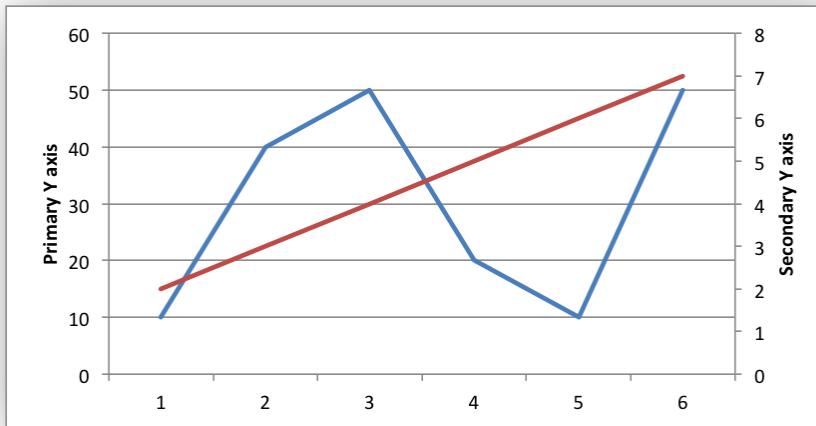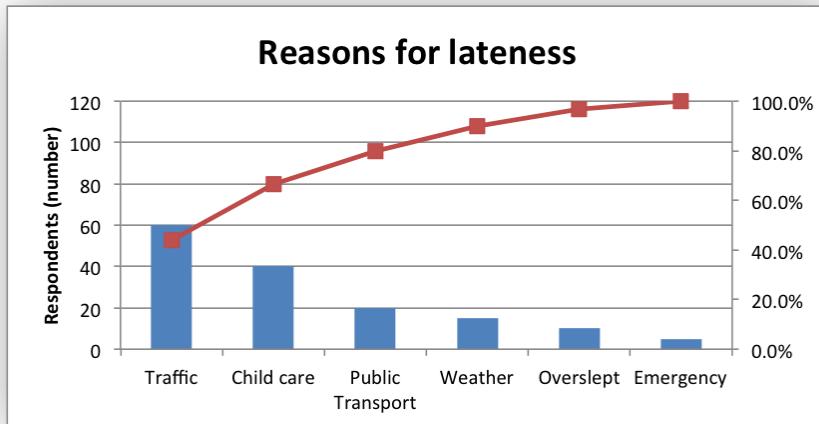

It is also possible to have a secondary, combined, chart either with a shared or secondary axis, see below.

18.20 Combined Charts

It is also possible to combine two different chart types, for example a column and line chart to create a Pareto chart using the Chart `combine()` method:

The combined charts can share the same Y axis like the following example:

```python
# Usual setup to create workbook and add data...

# Create a new column chart. This will use this as the primary chart.
column_chart = workbook.add_chart({'type': 'column'})

# Configure the data series for the primary chart.
column_chart.add_series({
    'name':       '=Sheet1!B1',
    'categories': '=Sheet1!A2:A7',
    'values':     '=Sheet1!B2:B7',
})

# Create a new column chart. This will use this as the secondary chart.
line_chart = workbook.add_chart({'type': 'line'})

# Configure the data series for the secondary chart.
line_chart.add_series({
    'name':       '=Sheet1!C1',
    'categories': '=Sheet1!A2:A7',
    'values':     '=Sheet1!C2:C7',
})

# Combine the charts.
column_chart.combine(line_chart)

# Add a chart title and some axis labels. Note, this is done via the
# primary chart.
column_chart.set_title({ 'name': 'Combined chart - same Y axis'})
column_chart.set_x_axis({'name': 'Test number'})
```
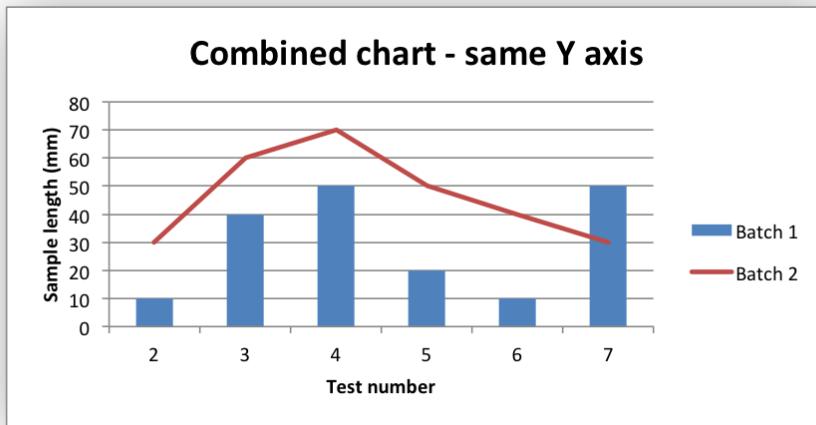
```
column_chart.set_y_axis({'name': 'Sample length (mm)'})

# Insert the chart into the worksheet
worksheet.insert_chart('E2', column_chart)
```
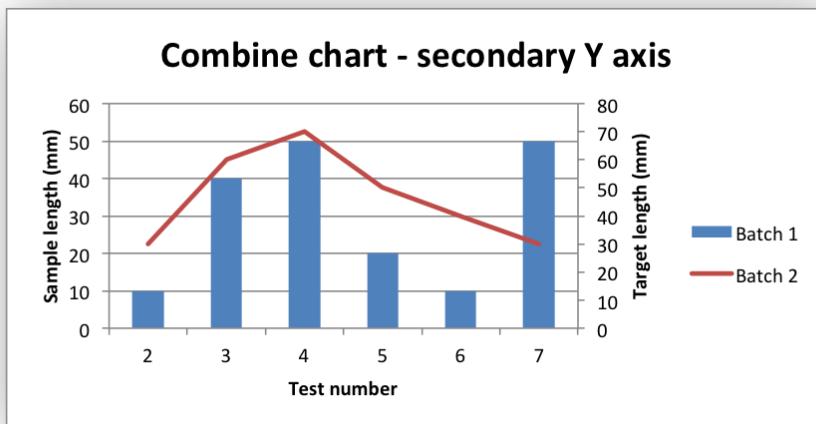

The secondary chart can also be placed on a secondary axis using the methods shown in the previous section.

In this case it is just necessary to add a y2_axis parameter to the series and, if required, add a title using set_y2_axis(). The following are the additions to the previous example to place the secondary chart on the secondary axis:

```
# ...
line_chart.add_series({
    'name':       '=Sheet1!C1',
    'categories': '=Sheet1!A2:A7',
    'values':     '=Sheet1!C2:C7',
    'y2_axis':    True,
})

# Add a chart title and some axis labels.
# ...
column_chart.set_y2_axis({'name': 'Target length (mm)'})
```

The examples above use the concept of a *primary* and *secondary* chart. The primary chart is the chart that defines the primary X and Y axis. It is also used for setting all chart properties apart from the secondary data series. For example the chart title and axes properties should be set via the primary chart.

See also *Example: Combined Chart* and *Example: Pareto Chart* for more detailed examples.
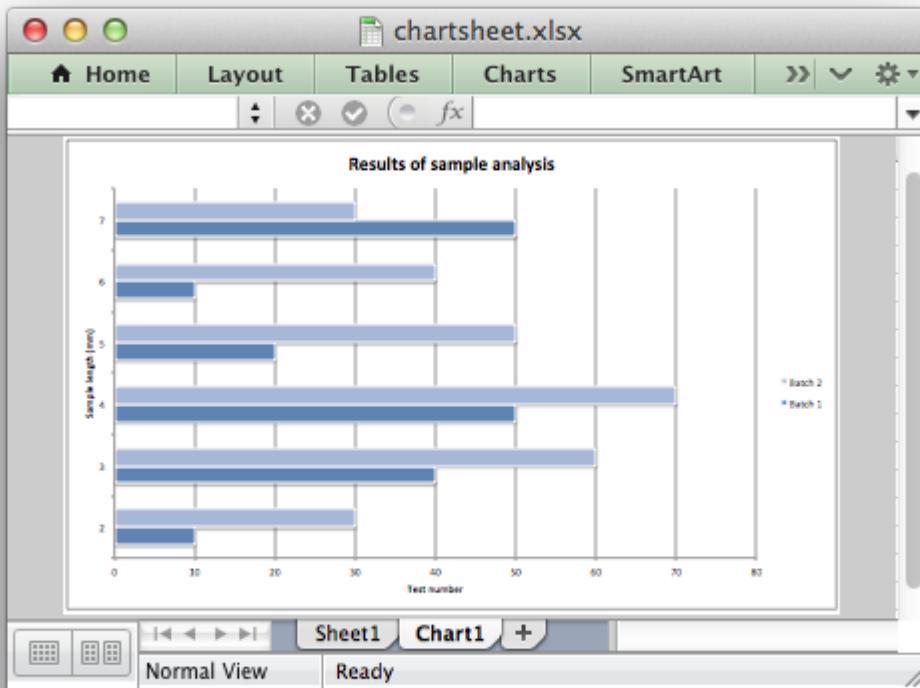
There are some limitations on combined charts:

- Only two charts can be combined.

- Pie charts cannot currently be combined.

- Scatter charts cannot currently be used as a primary chart but they can be used as a secondary chart.

- Bar charts can only combined secondary charts on a secondary axis. This is an Excel limitation.

18.21 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.

18.22 Charts from Worksheet Tables

Charts can by created from *Worksheet Tables*. However, Excel has a limitation where the data series name, if specified, must refer to a cell within the table (usually one of the headers).

To workaround this Excel limitation you can specify a user defined name in the table and refer to that from the chart:

```python
import xlsxwriter

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

worksheet = workbook.add_worksheet()

data = [
    ['Apple',  60],
    ['Cherry', 30],
    ['Pecan',  10],
]

worksheet.add_table('A1:B4', {'data': data,
                              'columns': [{'header': 'Types'},
```

```
                                                        {'header': 'Number'}]]
    )

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

    chart.add_series({
        'name':       '=Sheet1!$A$1',
        'categories': '=Sheet1!$A$2:$A$4',
        'values':     '=Sheet1!$B$2:$B$4',
    })

    worksheet.insert_chart('D2', chart)

    workbook.close()
```

18.23 Chart Limitations

The following chart features aren't supported in XlsxWriter:

- 3D charts and controls.
- Bubble, Surface or other chart types not listed in *The Chart Class*.

18.24 Chart Examples

See *Chart Examples*.

WORKING WITH OBJECT POSITIONING

XlsxWriter positions worksheet objects such as images, charts and textboxes in worksheets by calculating precise coordinates based on the object size, it's DPI (for images) and any scaling that the user specifies. It also takes into account the heights and widths of the rows and columns that the object crosses. In this way objects maintain their original sizes even if the rows or columns underneath change size or are hidden.

For example:
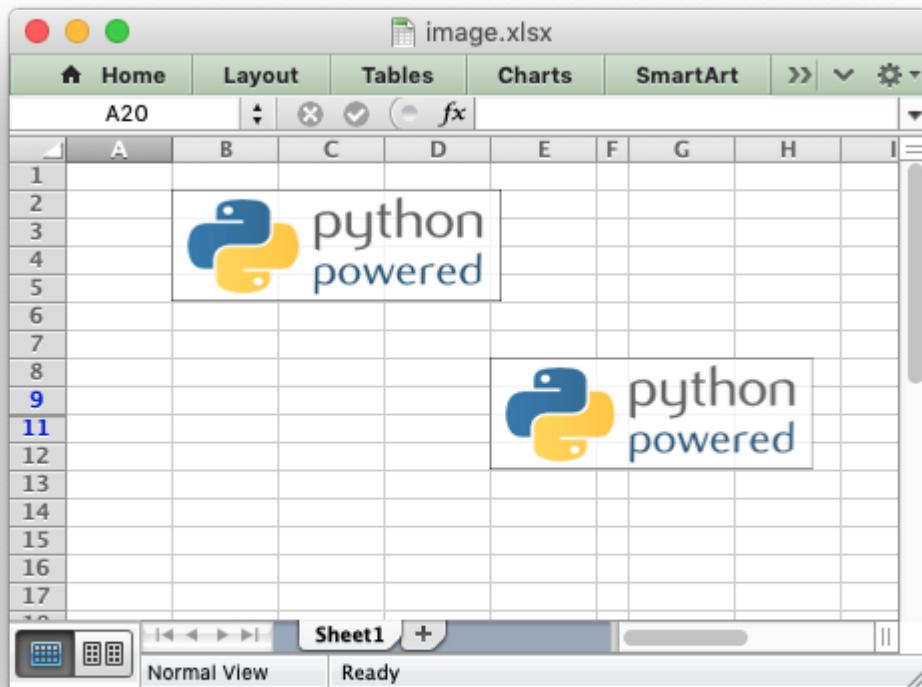
```python
import xlsxwriter

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

# Original image.
worksheet.insert_image('B2', 'logo.png')

# Same size as original, despite row/col changes.
worksheet.insert_image('E8', 'logo.png')

# Make column F narrower.
worksheet.set_column('F:F', 2)

# Hide row 10 (zero indexed).
worksheet.set_row(9, None, None, {'hidden': True})

workbook.close()
```

As can be seen the inserted image sizes are the same even though the second image crosses changed rows and columns.

However, there are two cases where the image scale may change with row or columns changes. These are explained in the next two sections.

19.1 Object scaling due to automatic row height adjustment

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. In these cases Excel will automatically calculate a row height based on the text when it loads the file. Since this row height isn't available to XlsxWriter when it creates the file the object may appear as if it is sized incorrectly. For example:
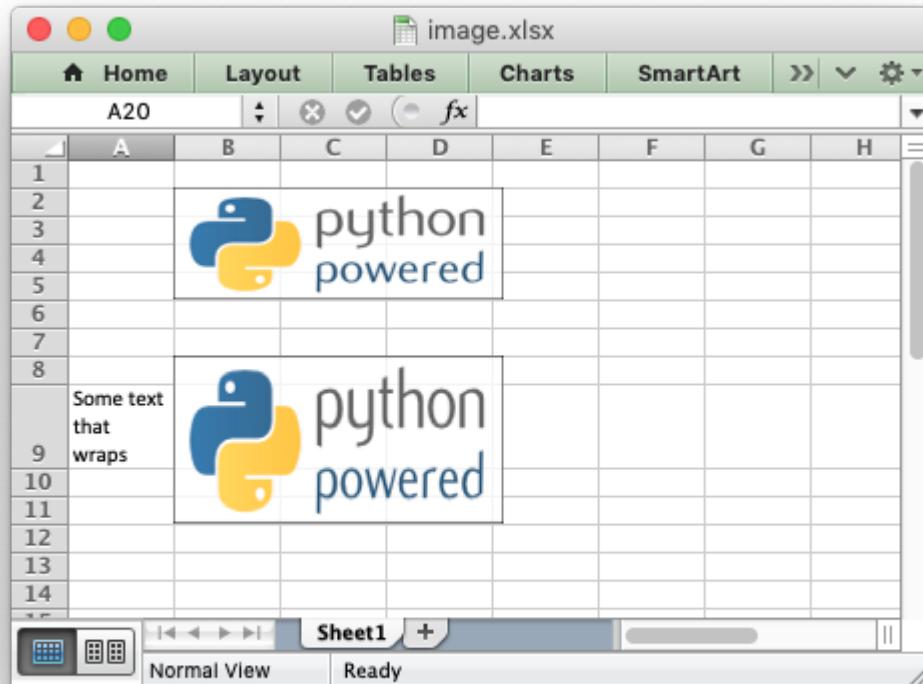
```python
import xlsxwriter

workbook = xlsxwriter.Workbook('image.xlsx')
worksheet = workbook.add_worksheet()
wrap_format = workbook.add_format({'text_wrap': True})

worksheet.write('A9', 'Some text that wraps', wrap_format)

worksheet.insert_image('B2', 'logo.png')
```
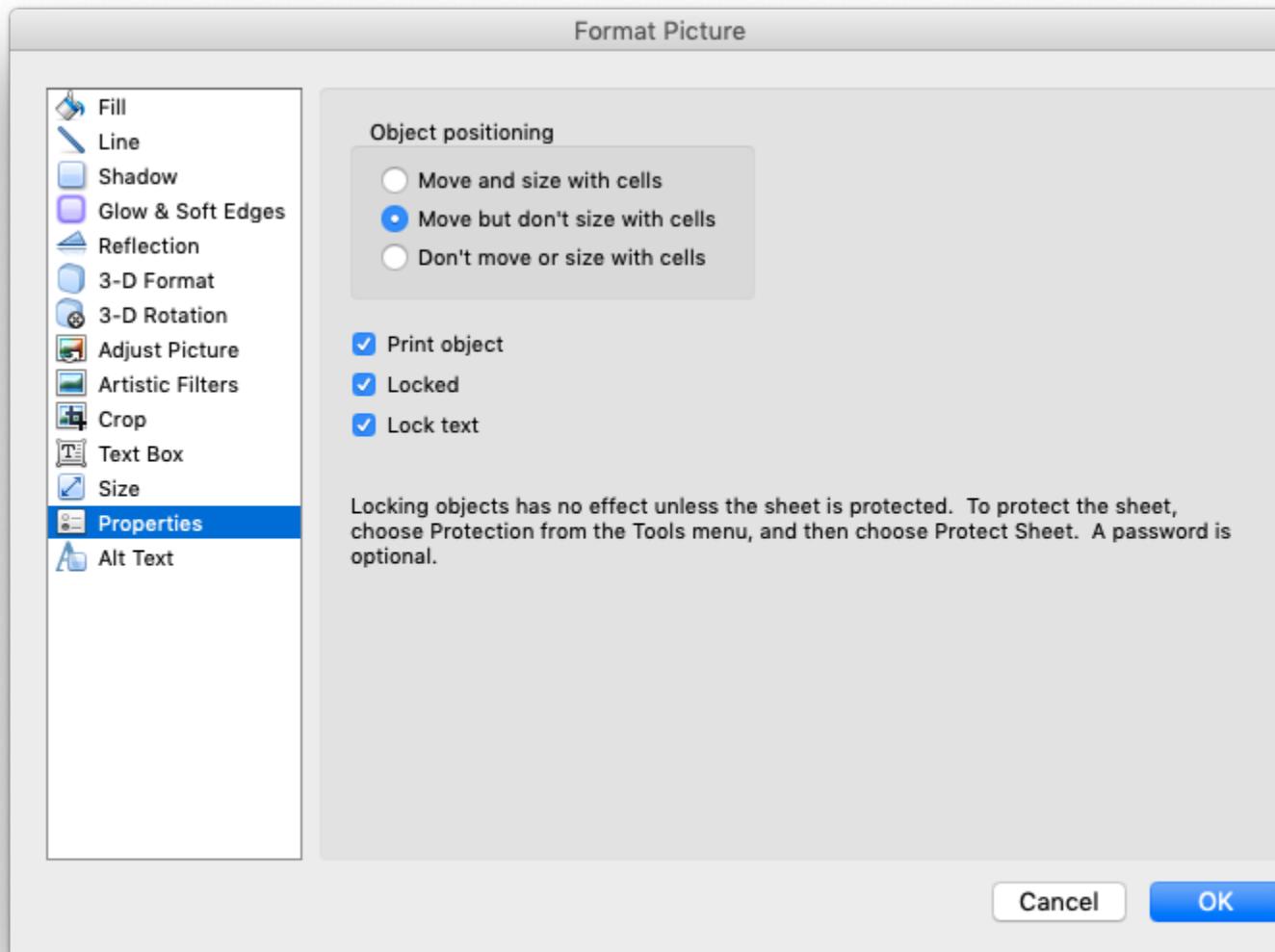
```
worksheet.insert_image('B8', 'logo.png')

workbook.close()
```


As can be seen the second inserted image is distorted, compared to the first, due to the row being scaled automatically. To avoid this you should explicitly set the height of the row using `set_row()` if it crosses an inserted object.

19.2 Object Positioning with Cell Moving and Sizing

Excel supports three options for "Object Positioning" within a worksheet:

Image, chart and textbox objects in XlsxWriter emulate these options using the `ob-ject_position` parameter:

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

Where `object_position` has one of the following allowable values:

1. Move and size with cells.

2. Move but don't size with cells.

3. Don't move or size with cells.

4. Same as Option 1 to "move and size with cells" except XlsxWriter applies hidden cells after the object is inserted.

Option 4 appears in Excel as Option 1. However, the worksheet object is sized to take hidden

rows or columns into account. This allows the user to hide an image in a cell, possibly as part of an autofilter. For example:
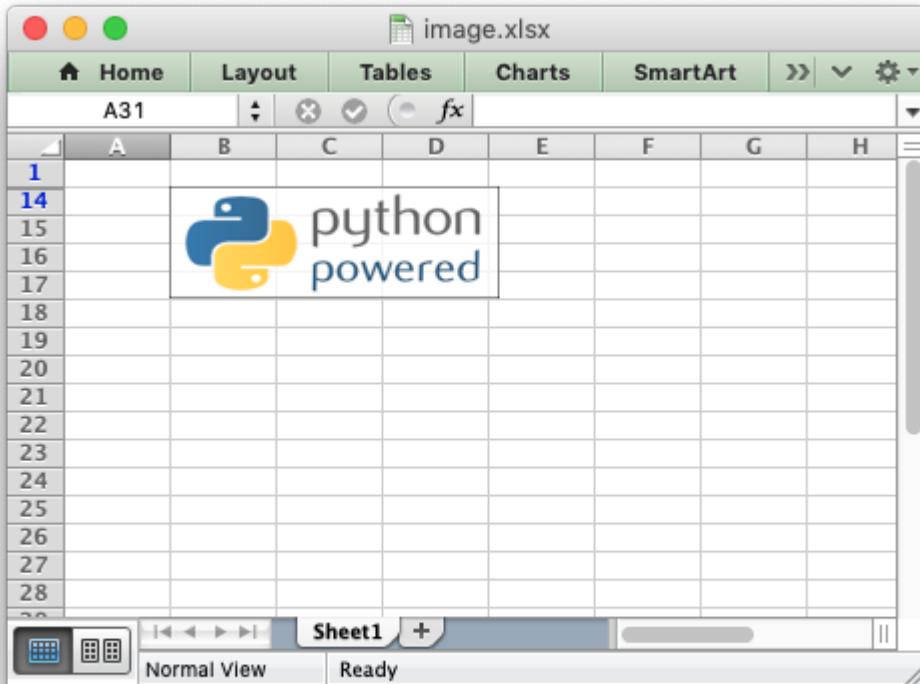
```python
import xlsxwriter

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

worksheet.insert_image('B2', 'logo.png')
worksheet.insert_image('B9', 'logo.png', {'object_position': 4})

# Hide some rows.
for row in range(1, 13):
    worksheet.set_row(row, None, None, {'hidden': True})

workbook.close()
```


In this example the first inserted image is visible over the hidden rows whilst the second image is hidden with the rows. Unhiding the rows in Excel would reveal the second image.

19.3 Image sizing and DPI

When an image is imported into Excel the DPI (dots per inch) resolution of the image is taken into account. Excel sizes the image according to a base DPI of 96. Therefore an image with a

DPI of 72 may appear slightly larger when imported into Excel while an image with a DPI of 200 may appear twice as small. XlsxWriter also reads the DPI of the images that the user inserts into a worksheet and stores the image dimensions in the same way that Excel does. If it cannot determine the DPI of the image it uses a default of 96.
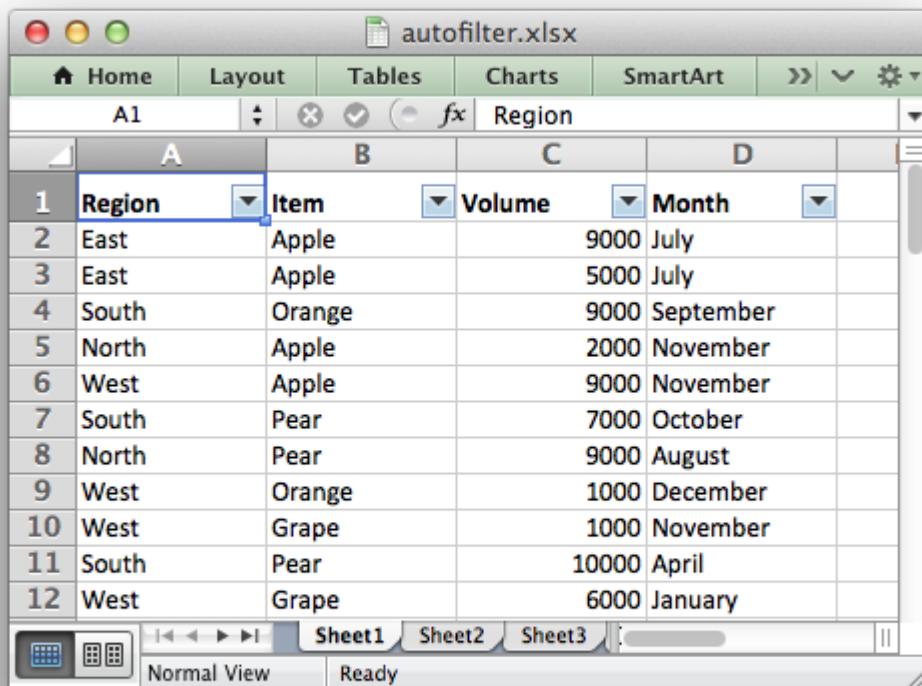
19.4 Reporting issues with image insertion

A lot of work has gone into ensuring that XlsxWriter inserts images into worksheets in exactly the same way that Excel does, even though the required calculations and units are arcane. There are over 80 test cases that check image insertion against files created in Excel to ensure that XlsxWriter's handling of images is correct.

As such, before reporting any issues with image handling in XlsxWriter please check how the same image is handled in Excel (not OpenOffice, LibreOffice or other third party applications). If you do report an issue please use the XlsxWriter Issue tracker is on GitHub and attach the image that demonstrates the issue.

WORKING WITH AUTOFILTERS

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

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 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 se-
lectors 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 crite-
ria:

```
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 contain 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:
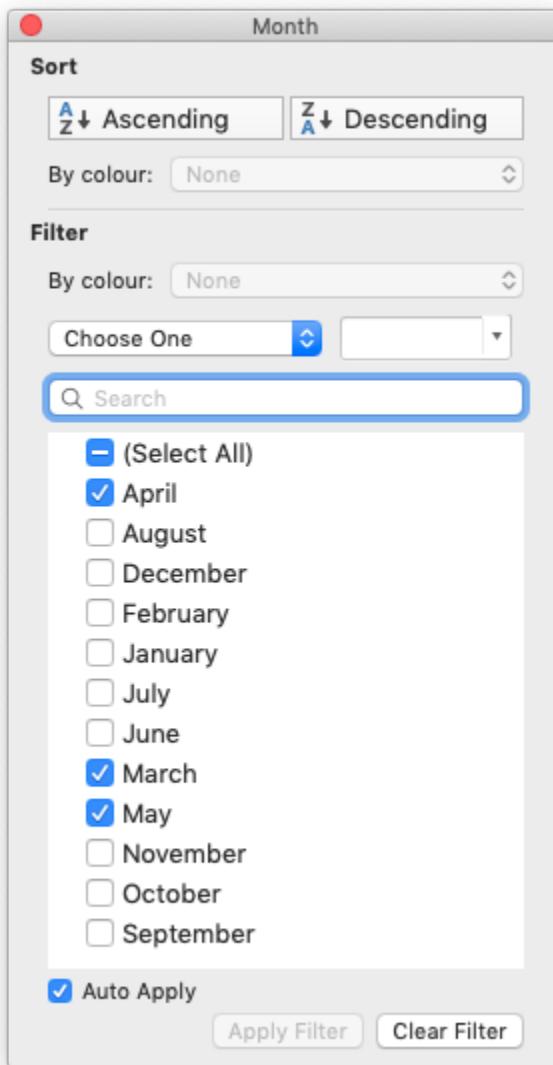
```
'x     < 2000'
'col   < 2000'
'Price < 2000'
```

A filter condition can only be applied to a column in a range specified by the `autofilter()` method.

20.3. Setting a filter criteria for a column   **295**

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])
```

To filter blanks as part of the list use *Blanks* as a list item:

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

As explained above, it isn't sufficient to just specify filters. You must also hide any rows that don't match the filter condition.

20.5 Example

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

TWENTYONE
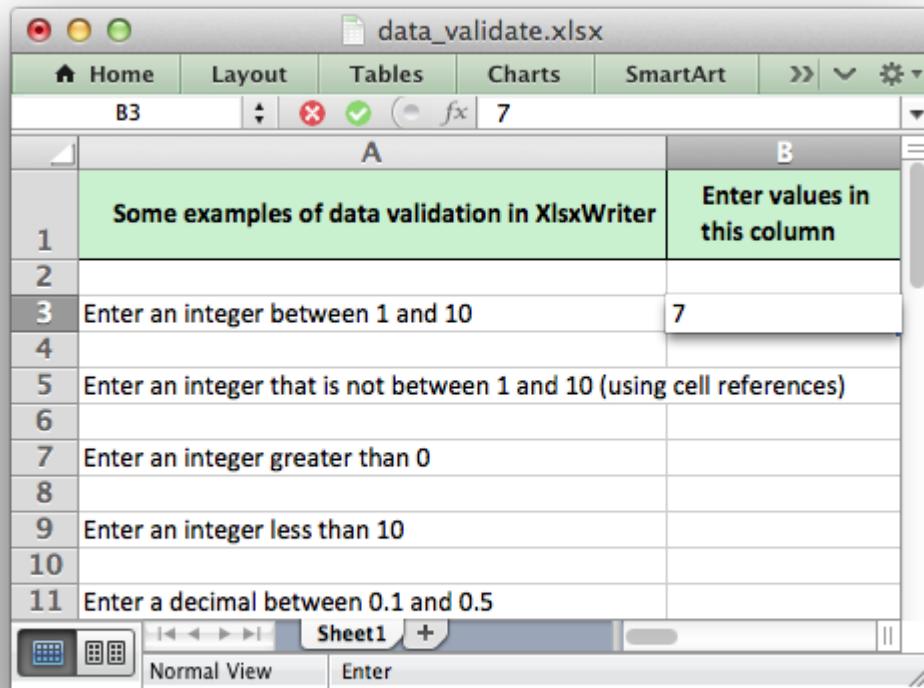
WORKING WITH DATA VALIDATION

Data validation is a feature of Excel which allows you to restrict the data that a user enters in a cell and to display associated help and warning messages. It also allows you to restrict input to values in a dropdown 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:

```python
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 multi_range	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* or a date formula.

- **time**: restricts the cell to time values specified as a datetime object as shown in *Working with Dates and Time* or a time formula.

- **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. It is mainly used for specifying cell input messages without a data validation.

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`, `custom` and `any` 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`, `custom` and `any` 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:

```python
# Using 'value'.
worksheet.data_validation('A1', {'validate': 'integer',
                                 'criteria': 'greater than',
                                 'value': 100})

# Using 'minimum'.
worksheet.data_validation('B11', {'validate': 'decimal',
                                  'criteria': 'between',
                                  'minimum': 0.1,
                                  'maximum': 0.5})

# Using 'source'.
worksheet.data_validation('B10', {'validate': 'list',
                                  'source': '=$E$4:$G$4'})

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

Note, when using the `list` validation with a list of strings, like in the last example above, Excel stores the strings internally as a Comma Separated Variable string. The total length for this string, including commas, cannot exceed the Excel limit of 255 characters. For longer sets of data you should use a range reference like the prior example above. Also any double quotes in strings like `'"Hello"'` must be double quoted like this `'""Hello""'`.

21.1.4 maximum

The `maximum` parameter is used to set the upper limiting value when the `criteria` is either `'between'` or `'not between'`:

```python
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:

```python
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.
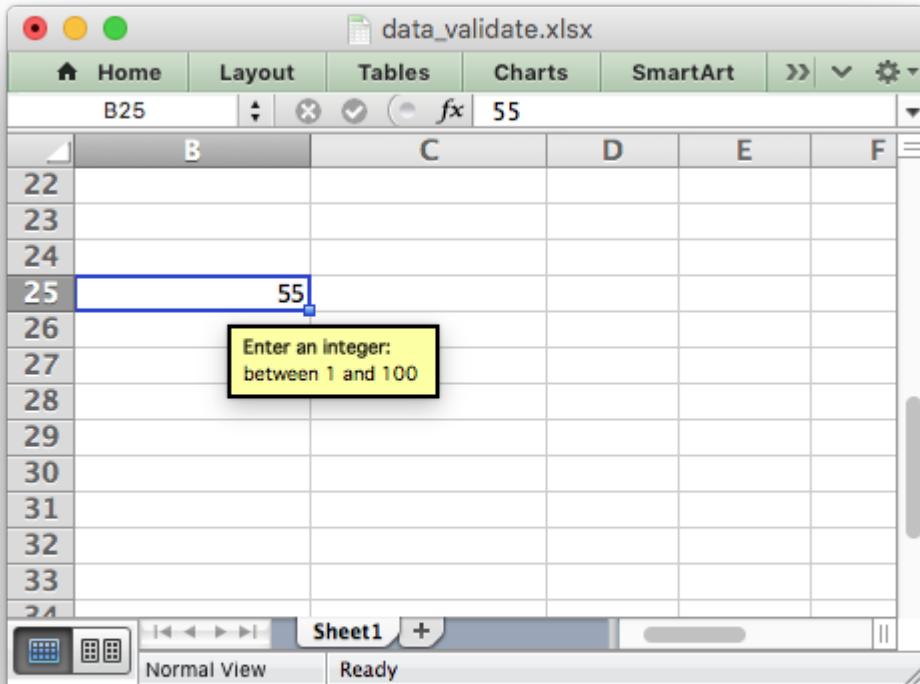
The maximum title length is 32 characters.

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.
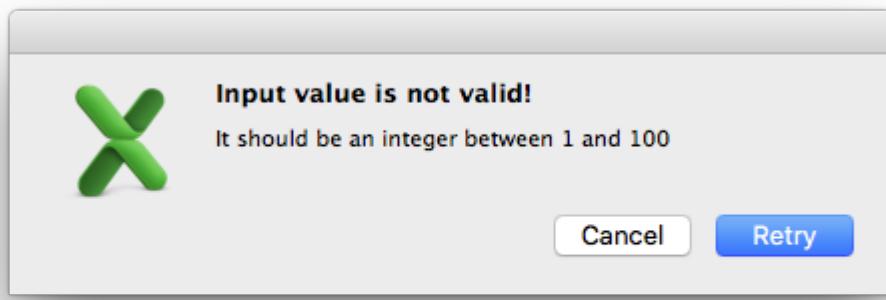
21.1.11 error_message

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': 'It should be an integer between 1
```

Which give the following message:

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.

21.1.14 multi_range

The `multi_range` option is used to extend a data validation over non-contiguous ranges.

It is possible to apply the data validation to different cell ranges in a worksheet using multiple calls to `data_validation()`. However, as a minor optimization it is also possible in Excel to apply the same data validation to different non-contiguous cell ranges.

21.1. data_validation()

This is replicated in `data_validation()` using the `multi_range` option. The range must contain the primary range for the data validation and any others separated by spaces.

For example to apply one data validation to two ranges, `'B3:K6'` and `'B9:K12'`:

```
worksheet.data_validation('B3:K6', {'validate': 'integer',
                                    'criteria': 'between',
                                    'minimum': 1,
                                    'maximum': 100,
                                    'multi_range': 'B3:K6 B9:K12'})
```

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*.
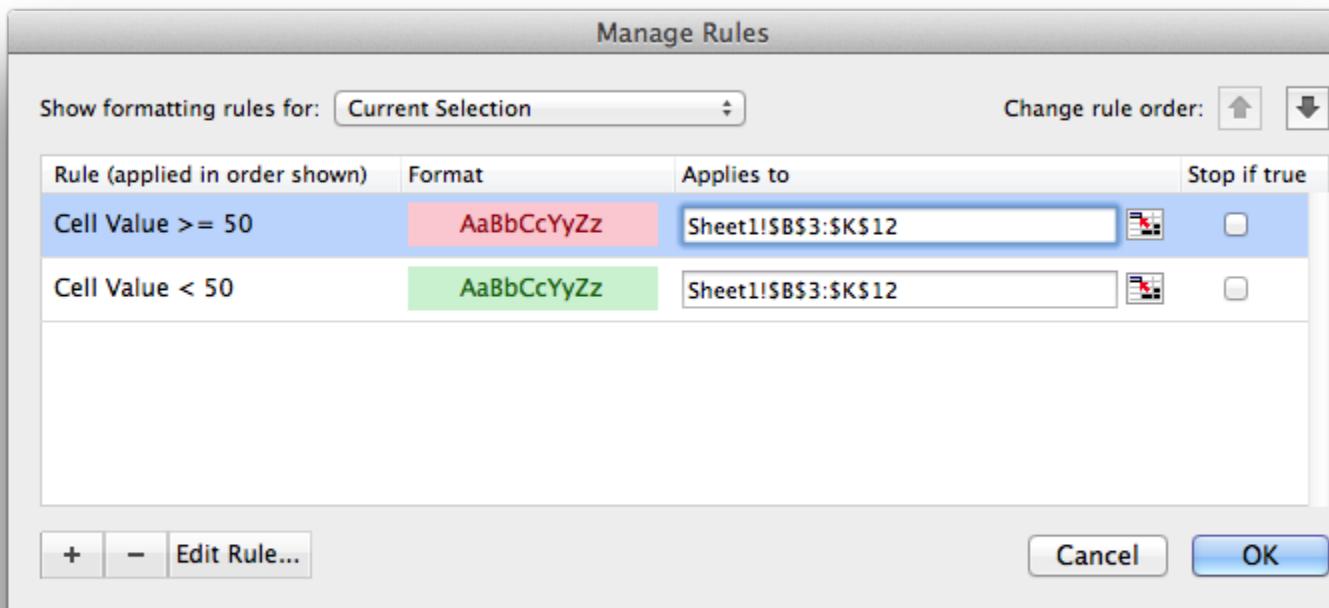
WORKING WITH CONDITIONAL FORMATTING

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.
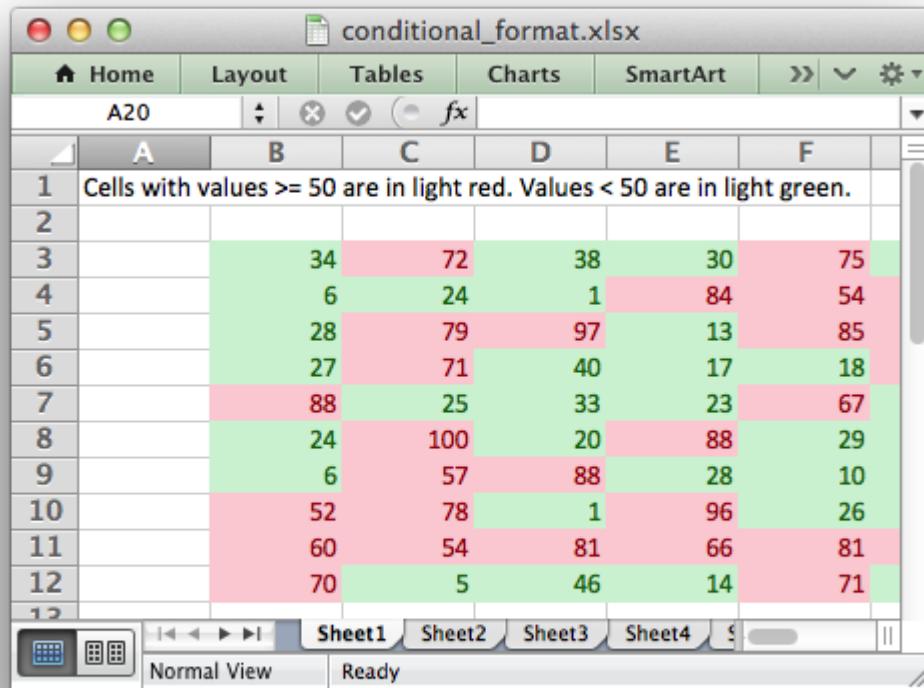
For example the following rules are used to highlight cells in the *conditional_format.py* example:

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

worksheet.conditional_format('B3:K12', {'type':     'cell',
                                        'criteria': '<',
                                        'value':    50,
                                        'format':   format2})
```
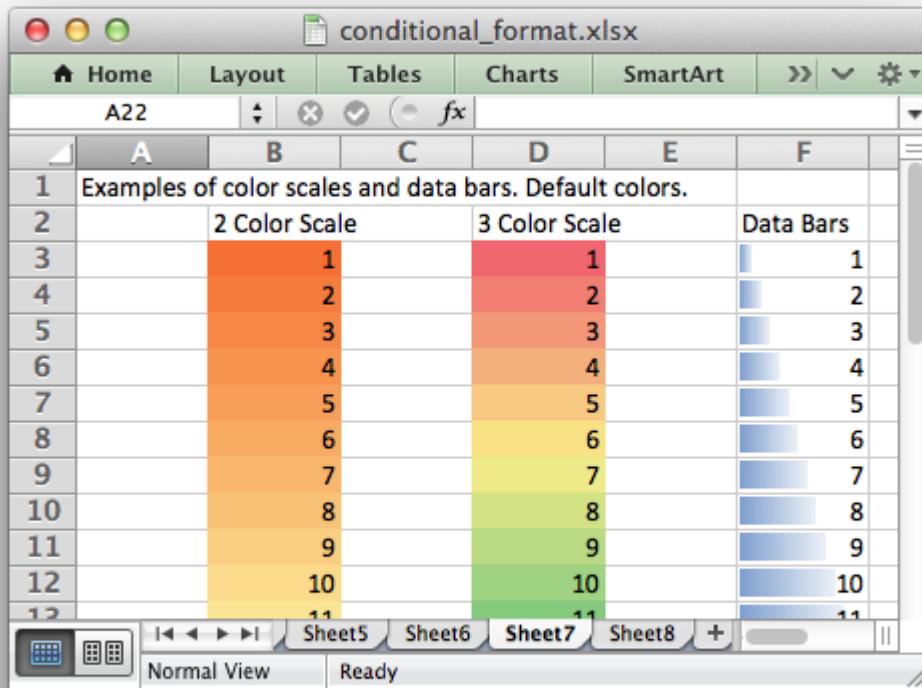
Which gives criteria like this:

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
- bar_only
- bar_solid
- bar_negative_color
- bar_border_color
- bar_negative_border_color
- bar_negative_color_same
- bar_negative_border_color_same
- bar_no_border
- bar_direction
- bar_axis_position
- bar_axis_color
- data_bar_2010
- icon_style
- icons
- reverse_icons
- icons_only
- stop_if_true

- `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
	format
date	criteria
	value
	minimum
	maximum
	format
time_period	criteria
	format
text	criteria
	value
	format
average	criteria
	format
duplicate	format
unique	format
top	criteria
	value
	format
bottom	criteria
	value
	format
blanks	format
no_blanks	format
errors	format
no_errors	format
formula	criteria
	format
2_color_scale	min_type
	max_type
	min_value
Continued on next page	

Table 22.1 – continued from previous page

Type	Parameters
3_color_scale	max_value
	min_color
	max_color
	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_only
	bar_color
	bar_solid*
	bar_negative_color*
	bar_border_color*
	bar_negative_border_color*
	bar_negative_color_same*
	bar_negative_border_color_same*
	bar_no_border*
	bar_direction*
	bar_axis_position*
	bar_axis_color*
	data_bar_2010*
icon_set	icon_style
	reverse_icons
	icons
	icons_only

Note: Data bar parameters marked with (*) are only available in Excel 2010 and later. Files that use these properties can still be opened in Excel 2007 but the data bars will be displayed without them.

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': 'equal to',
                                    'value':     5,
                                    'format':    red_format})
```

If the type is cell and the value is a string then it should be double quoted, as required by Excel:

```
worksheet.conditional_format('A1', {'type':     'cell',
                                    'criteria': 'equal to',
                                    'value':     '"Failed"',
                                    'format':    red_format})
```

The `value` property can also be an cell reference:

```
worksheet.conditional_format('A1', {'type':     'cell',
                                    'criteria': 'equal to',
                                    '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, apart 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 *The Format Class*.

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': 'next week',
'criteria': 'last month',
'criteria': 'this month',
'criteria': 'next 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: 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()`. See *Non US Excel functions and syntax* for a full explanation:

```
# This formula will cause an Excel error on load due to
# non-English language and use of semi-colons.
worksheet.conditional_format('A2:C9' ,
    {'type':     'formula',
     'criteria': '=ODER($B2<$C2;UND($B2="";$C2>HEUTE()))',
     'format':   format1
    })

# This is the correct syntax.
worksheet.conditional_format('A2:C9' ,
    {'type':     'formula',
     'criteria': '=OR($B2<$C2,AND($B2="",$C2>TODAY()))',
     'format':   format1
    })
```
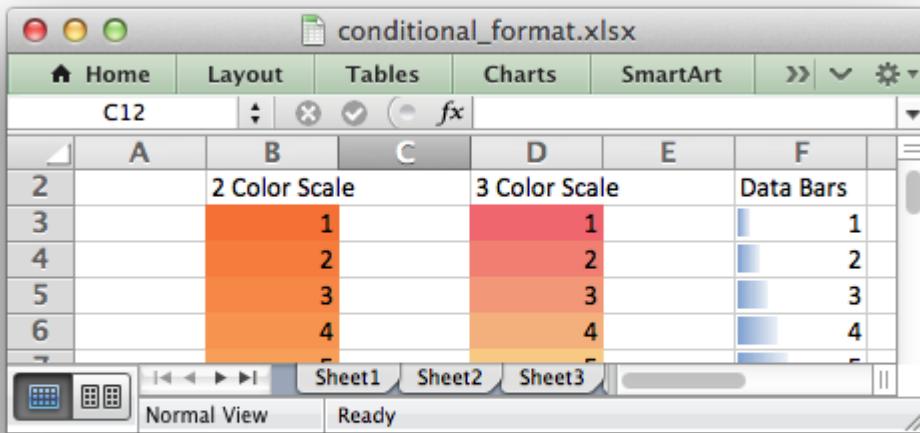
Also, any cell or range references in the formula should be *absolute references* if they are applied to the full range of the conditional format. See the note in the `value` section above.

22.2.21 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.

22.2.22 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.

22.2.23 type: data_bar

The `data_bar` type is used to specify Excel's "Data Bar" style conditional format:
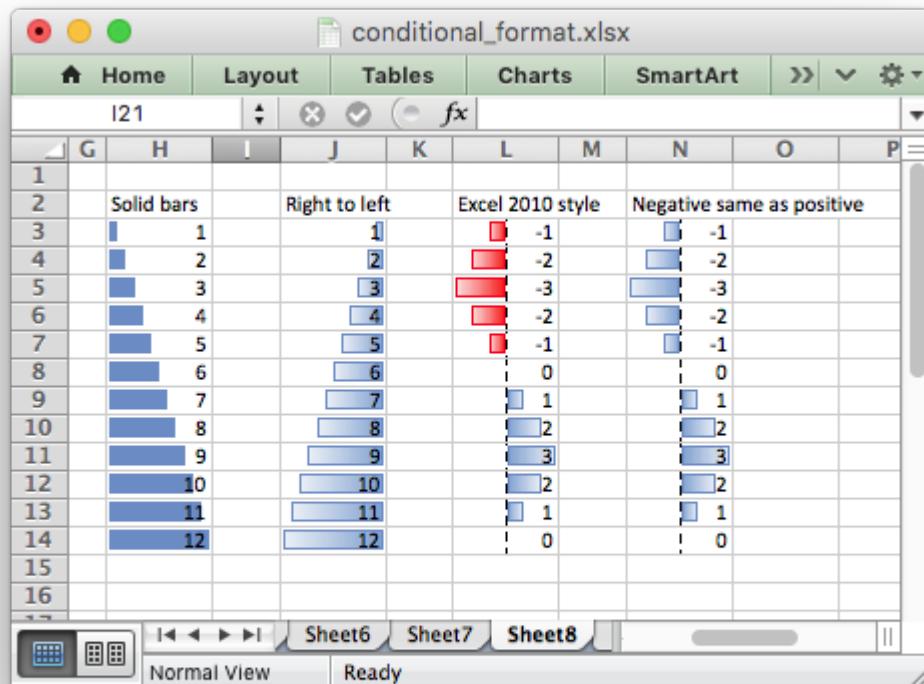
```
worksheet.conditional_format('A1:A12', {'type': 'data_bar'})
```

This conditional type can be modified with the following parameters, which are explained in the sections below. These properties were available in the original xlsx file specification used in Excel 2007:

```
min_type
max_type
min_value
max_value
bar_color
bar_only
```

In Excel 2010 additional data bar properties were added such as solid (non-gradient) bars and control over how negative values are displayed. These properties can be set using the following parameters:

```
bar_solid
bar_negative_color
bar_border_color
bar_negative_border_color
bar_negative_color_same
bar_negative_border_color_same
bar_no_border
bar_direction
bar_axis_position
bar_axis_color
data_bar_2010
```


Files that use these Excel 2010 properties can still be opened in Excel 2007 but the data bars will be displayed without them.

22.2.24 type: icon_set

The `icon_set` type is used to specify a conditional format with a set of icons such as traffic lights or arrows:
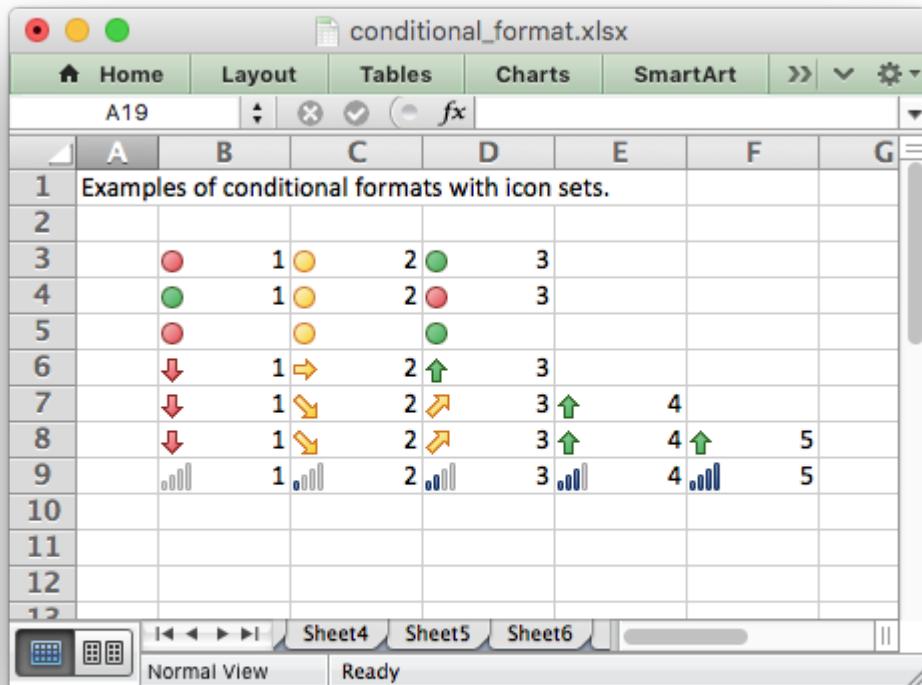
```
worksheet.conditional_format('A1:C1', {'type': 'icon_set',
                                       'icon_style': '3_traffic_lights'})
```

The icon set style is specified by the `icon_style` parameter. Valid options are:

```
3_arrows
3_arrows_gray
3_flags
3_signs
3_symbols
3_symbols_circled
3_traffic_lights
3_traffic_lights_rimmed

4_arrows
4_arrows_gray
4_ratings
4_red_to_black
4_traffic_lights

5_arrows
5_arrows_gray
5_quarters
5_ratings
```


The criteria, type and value of each icon can be specified using the `icon` array of dicts with optional `criteria`, `type` and `value` parameters:

```
worksheet.conditional_format(
    'A1:D1',
    {'type': 'icon_set',
     'icon_style': '4_red_to_black',
     'icons': [{'criteria': '>=', 'type': 'number',     'value': 90},
               {'criteria': '<',  'type': 'percentile', 'value': 50},
               {'criteria': '<=', 'type': 'percent',    'value': 25}]}
)
```

- The icons `criteria` parameter should be either >= or <. The default `criteria` is >=.

- The icons `type` parameter should be one of the following values:

    ```
    number
    percentile
    percent
    formula
    ```

    The default `type` is `percent`.

- The icons `value` parameter can be a value or formula:

    ```
    worksheet.conditional_format('A1:D1',
                                 {'type': 'icon_set',
                                  'icon_style': '4_red_to_black',
                                  'icons': [{'value': 90},
                                            {'value': 50},
                                            {'value': 25}]})
    ```

Note: The `icons` parameters should start with the highest value and with each subsequent one being lower. The default `value` is (n * 100) / number_of_icons. The lowest number icon in an icon set has properties defined by Excel. Therefore in a n icon set, there is no n-1 hash of parameters.

The order of the icons can be reversed using the `reverse_icons` parameter:

```
worksheet.conditional_format('A1:C1',
                             {'type': 'icon_set',
                              'icon_style': '3_arrows',
                              'reverse_icons': True})
```

The icons can be displayed without the cell value using the `icons_only` parameter:

```
worksheet.conditional_format('A1:C1',
                             {'type': 'icon_set',
                              'icon_style': '3_flags',
                              'icons_only': True})
```

22.2.25 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:

```
min          (for min_type only)
num
percent
percentile
formula
max          (for max_type only)
```

22.2.26 mid_type:

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

22.2.27 max_type:

Same as `min_type`, see above.

22.2.28 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})
```

22.2.29 mid_value:

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

22.2.30 max_value:

Same as `min_value`, see above.

22.2.31 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 color can be a Html style #RRGGBB string or a limited number named colors, see *Working with Colors*.

22.2.32 mid_color:

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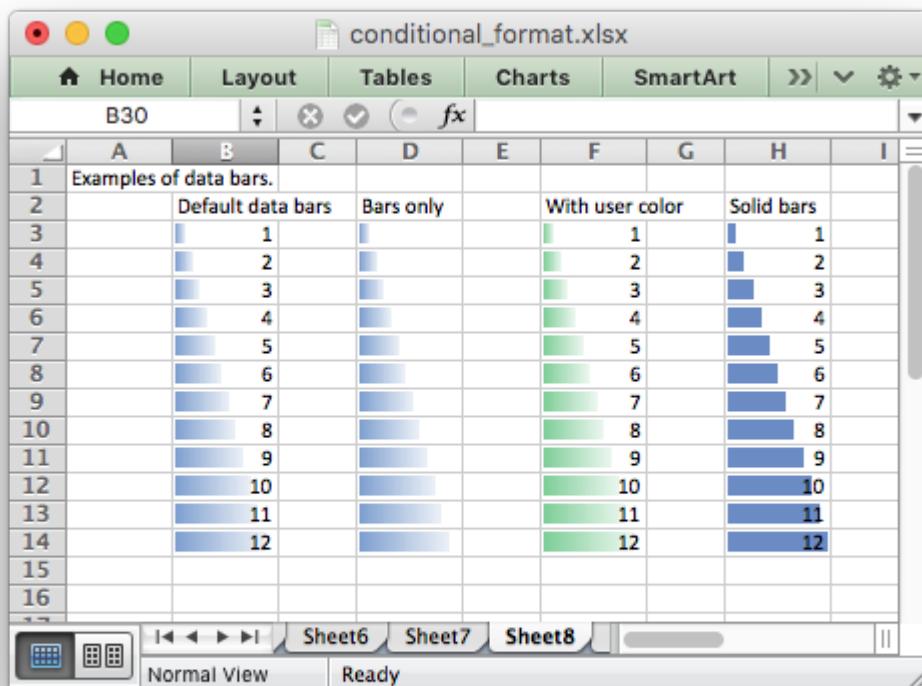
22.2.33 max_color:

Same as `min_color`, see above.

22.2.34 bar_color:

The `bar_color` parameter sets the fill color for data bars:

```
worksheet.conditional_format('F3:F14', {'type': 'data_bar',
                                        'bar_color': '#63C384'})
```


The color can be a Html style #RRGGBB string or a limited number named colors, see *Working with Colors*.

22.2.35 bar_only:

The `bar_only` property displays a bar data but not the data in the cells:

```
worksheet.conditional_format('D3:D14', {'type': 'data_bar',
                                        'bar_only': True})
```

```
worksheet.conditional_format('F3:F14', {'type': 'data_bar',
                                        'bar_negative_color_same': True})
```


Note, this property is only visible in Excel 2010 and later.

22.2.41 bar_negative_border_color_same:

The `bar_negative_border_color_same` property sets the border color for the negative portion of a data bar to be the same as the border color for the positive portion of the data bar:

```
worksheet.conditional_format('F3:F14', {'type': 'data_bar',
                                        'bar_negative_border_color_same': True})
```

See the image above.

Note, this property is only visible in Excel 2010 and later.

22.2.42 bar_no_border:

The `bar_no_border` property turns off the border for data bars:
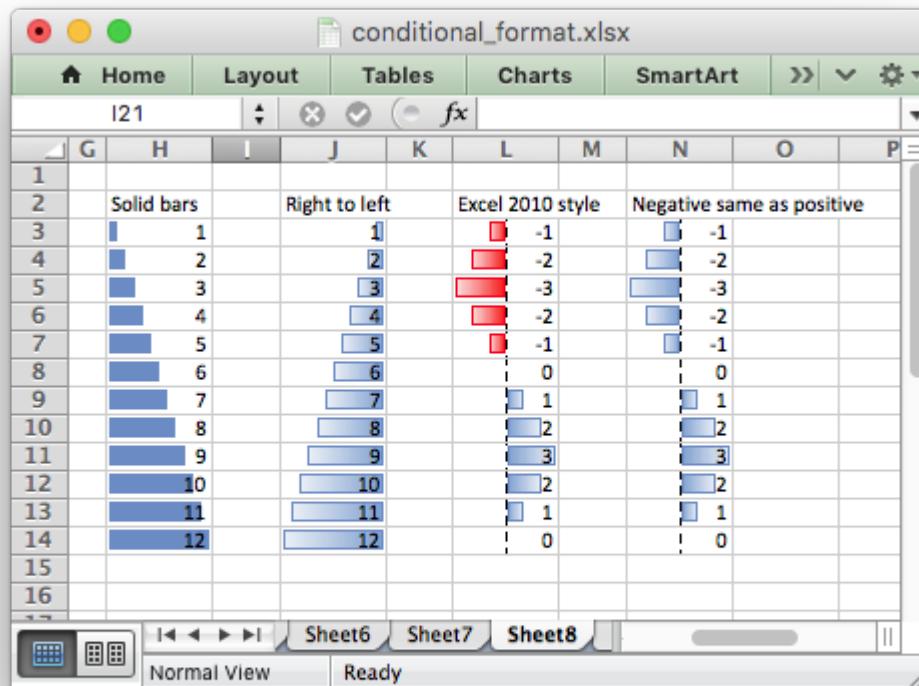
```
worksheet.conditional_format('F3:F14', {'type': 'data_bar',
                                        'bar_no_border': True})
```

Note, this property is only visible in Excel 2010 and later, however the default in Excel 2007 is to not have a border.

22.2.43 bar_direction:

The `bar_direction` property sets the direction for data bars. This property can be either `left` for left-to-right or `right` for right-to-left. If the property isn't set then Excel will adjust the position automatically based on the context:

```
worksheet.conditional_format('J3:J14', {'type': 'data_bar',
                                        'bar_direction': 'right'})
```


Note, this property is only visible in Excel 2010 and later.

22.2.44 bar_axis_position:

The `bar_axis_position` property sets the position within the cells for the axis that is shown in data bars when there are negative values to display. The property can be either `middle` or `none`. If the property isn't set then Excel will position the axis based on the range of positive and negative values:

```
worksheet.conditional_format('J3:J14', {'type': 'data_bar',
                                        'bar_axis_position': 'middle'})
```

Note, this property is only visible in Excel 2010 and later.

22.2.45 bar_axis_color:

The `bar_axis_color` property sets the color for the axis that is shown in data bars when there are negative values to display:

```
worksheet.conditional_format('J3:J14', {'type': 'data_bar',
                                        'bar_axis_color': '#0070C0'})
```

Note, this property is only visible in Excel 2010 and later.

22.2.46 data_bar_2010:

The `data_bar_2010` property sets Excel 2010 style data bars even when Excel 2010 specific properties aren't used. This can be used for consistency across all the data bar formatting in a worksheet:

```
worksheet.conditional_format('L3:L14', {'type': 'data_bar',
                                        'data_bar_2010': True})
```

22.2.47 stop_if_true

The `stop_if_true` parameter can be used to set the "stop if true" feature of a conditional formatting rule when more than one rule is applied to a cell or a range of cells. When this parameter is set then subsequent rules are not evaluated if the current rule is true:

```
worksheet.conditional_format('A1',
                                  {'type': 'cell',
                                   'format': cell_format,
                                   'criteria': '>',
                                   'value': 20,
                                   'stop_if_true': True
                                  })
```

22.2.48 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 optimization 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:

```python
worksheet.conditional_format('A1:F10', {'type':     'cell',
                                        'criteria': 'greater than',
                                        'value':     5,
                                        'format':   format1})
```

Highlight cells greater than a value in a reference cell:

```python
worksheet.conditional_format('A1:F10', {'type':     'cell',
                                        'criteria': 'greater than',
                                        'value':     'H1',
                                        'format':   format1})
```

Highlight cells more recent (greater) than a certain date:

```python
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:

```python
worksheet.conditional_format('A1:F10', {'type':     'time_period',
                                        'criteria': 'last 7 days',
                                        'format':   format1})
```

Highlight cells with strings starting with the letter b:

```python
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:

```python
worksheet.conditional_format('A1:F10', {'type':    'average',
                                        'format': format1})
```

Highlight duplicate cells in a range:

```python
worksheet.conditional_format('A1:F10', {'type':    'duplicate',
                                        'format': format1})
```

Highlight unique cells in a range:

```python
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})
```

Set traffic light icons in 3 cells:

```
worksheet.conditional_format('B3:D3', {'type': 'icon_set',
                                       'icon_style': '3_traffic_lights'})
```

See also *Example: Conditional Formatting*.
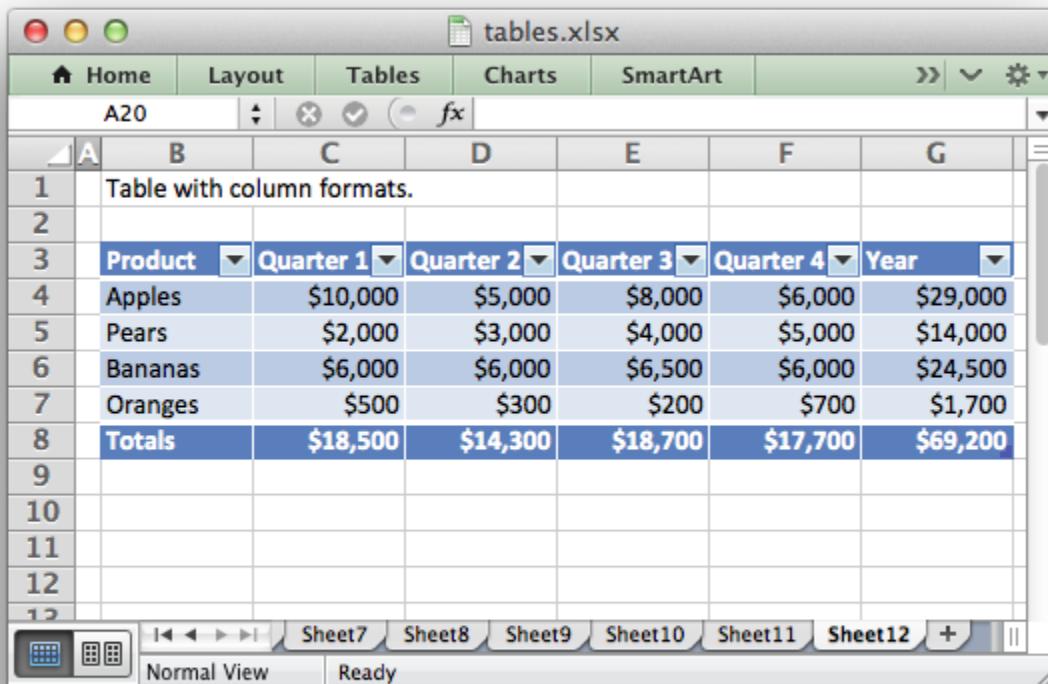
WORKING WITH WORKSHEET TABLES

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.
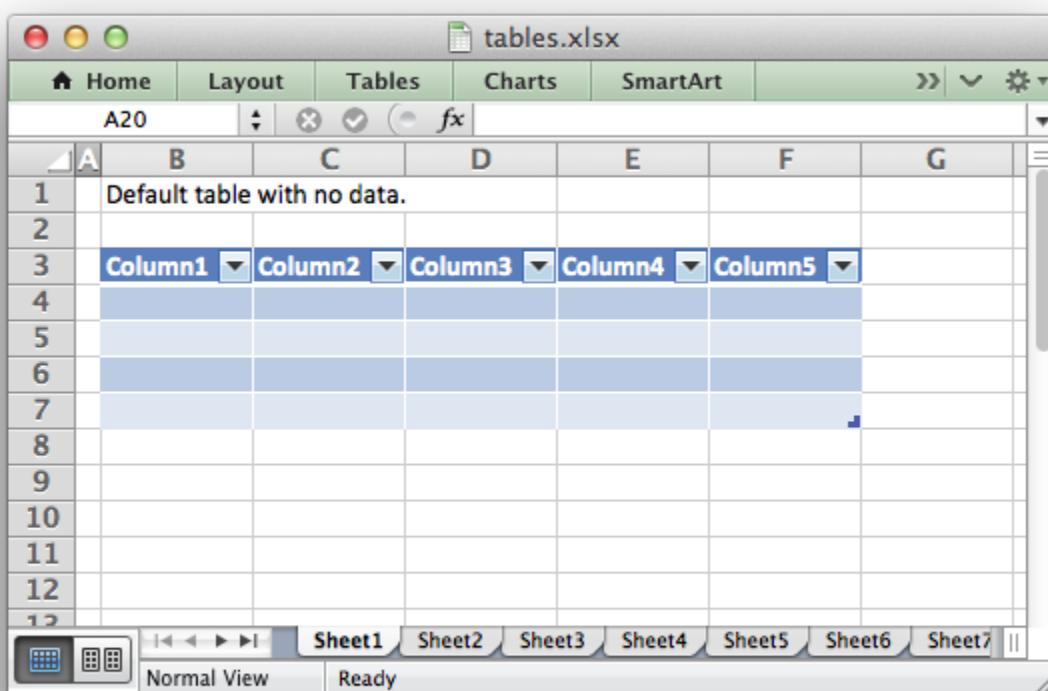
23.1 add_table()

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:

```python
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})
```
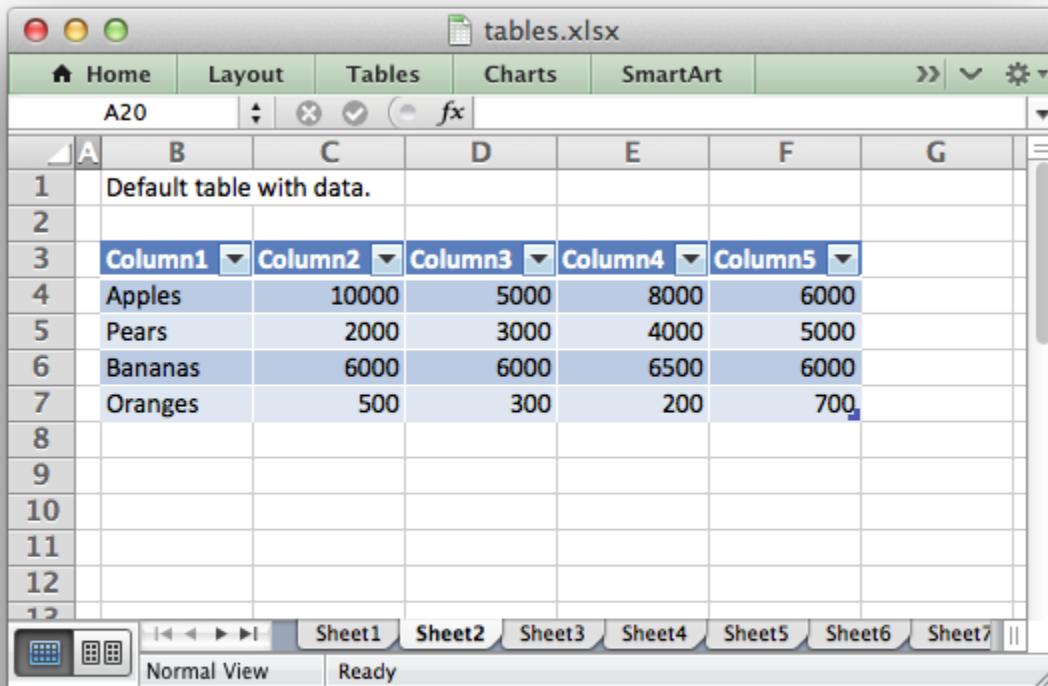
Table data can also be written separately, as an array or individual cells:

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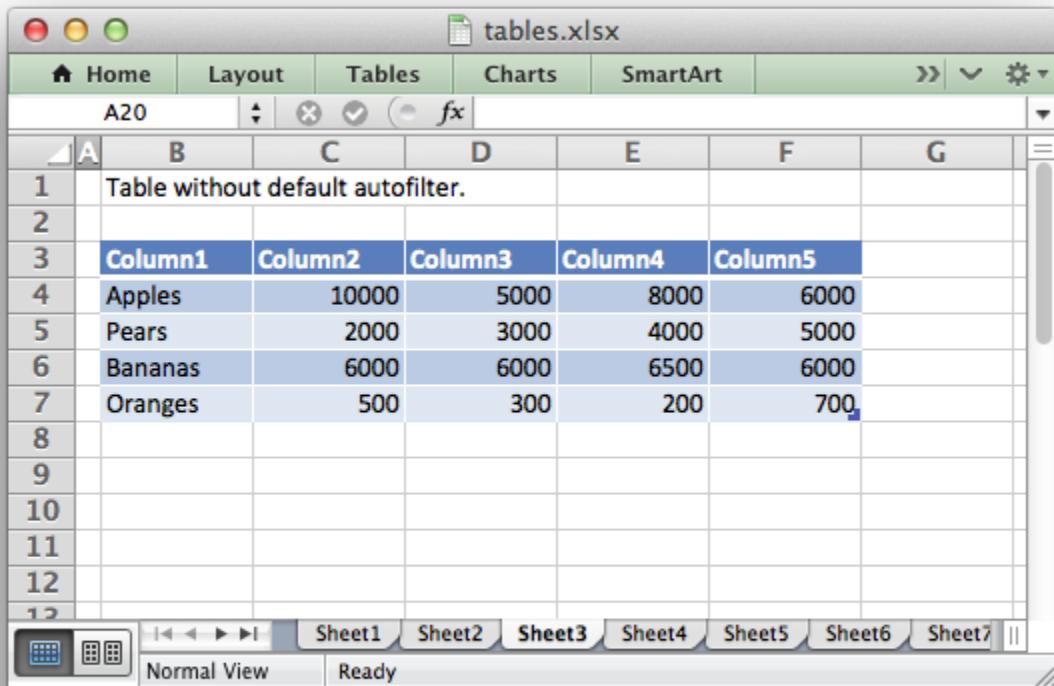
The header row will contain default captions such as `Column 1`, `Column 2`, etc. These captions can be overridden using the `columns` parameter below.

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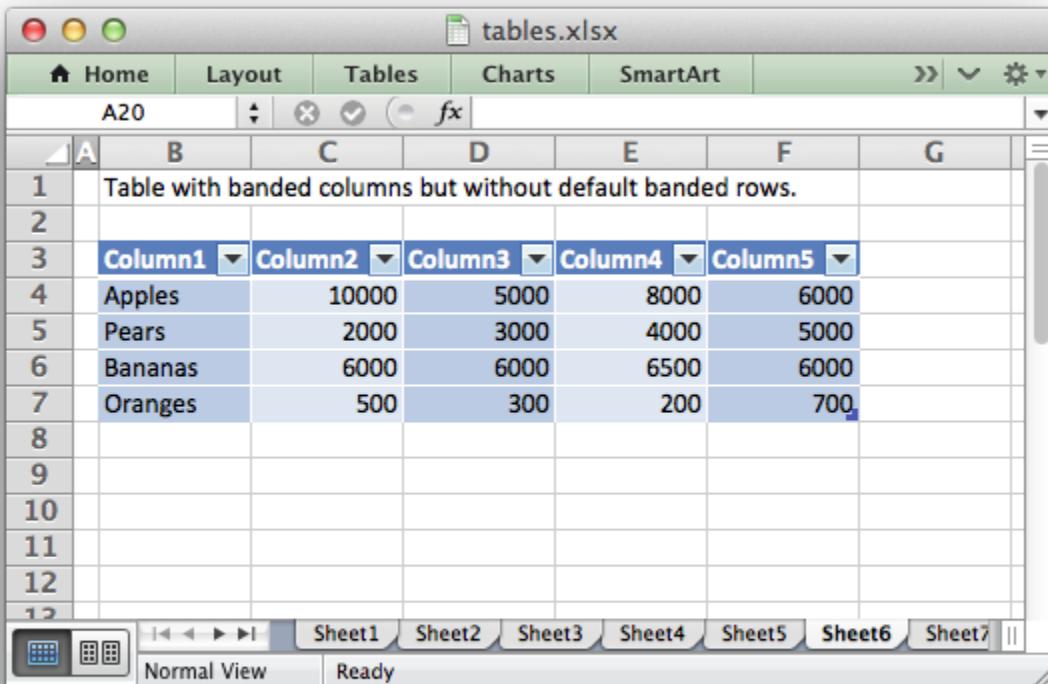
The `autofilter` is only shown if the `header_row` is on. Filter conditions within the table are not supported.

23.5 banded_rows

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

```python
# 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 color in the table. It is off by default:
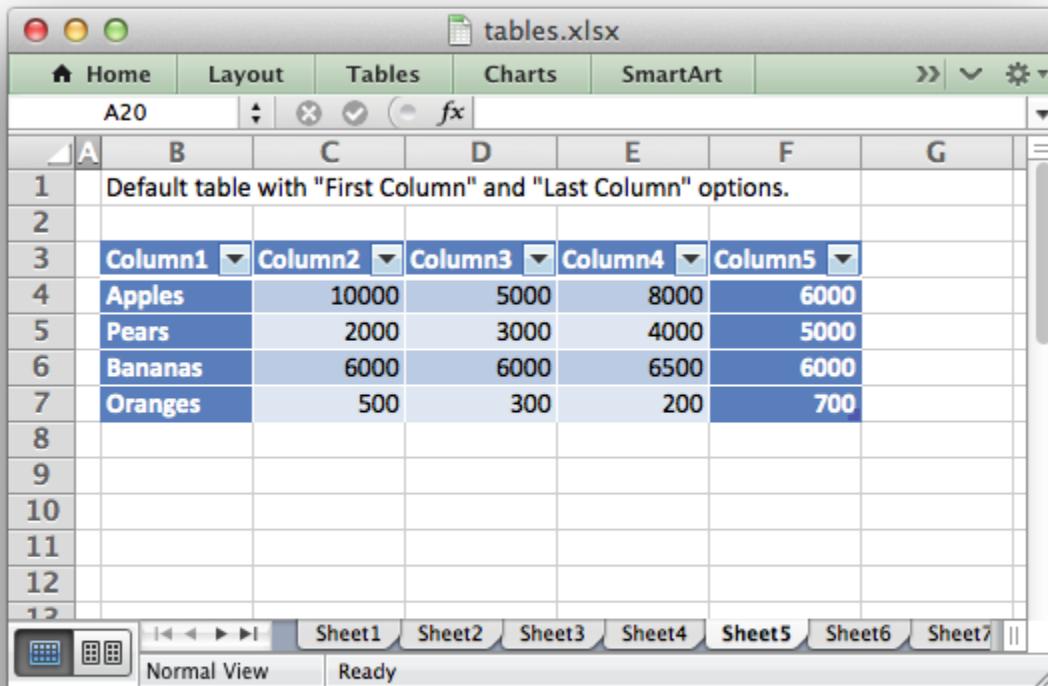
```
# Turn on banded columns.
worksheet.add_table('B3:F7', {'banded_columns': True})
```

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 color. 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 color. It is off by default:
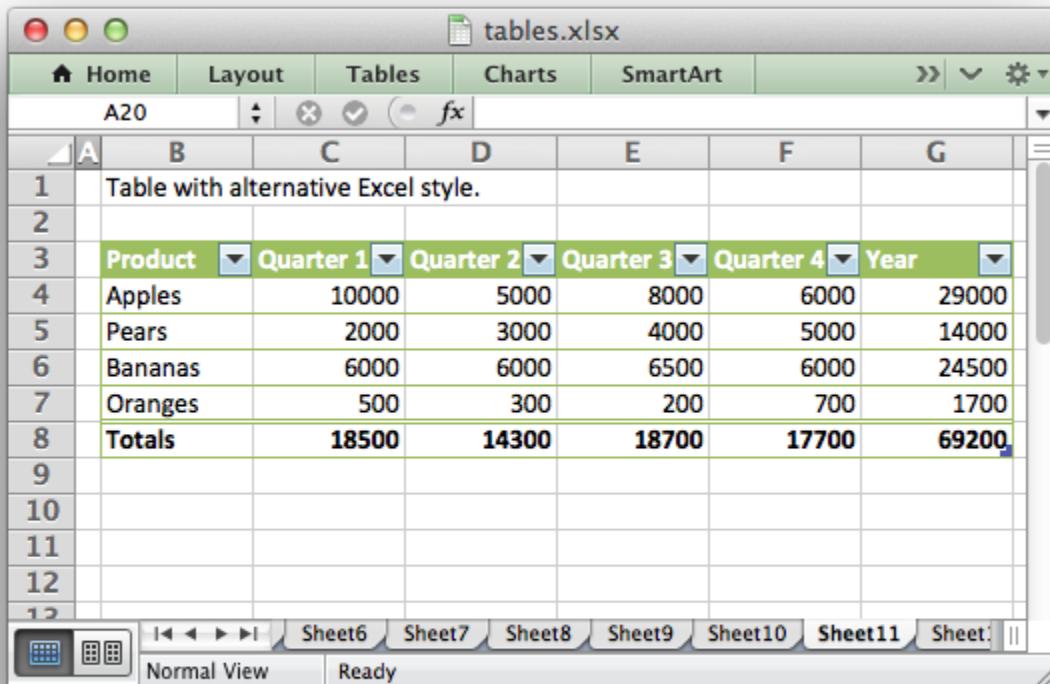
```
# Turn on highlighting for the last column in the table.
worksheet.add_table('B3:F7', {'last_column': True})
```

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 capitalization):

```
worksheet.add_table('B3:F7', {'data': data,
                              'style': 'Table Style Light 11'})
```

The default table style is 'Table Style Medium 9'.

You can also turn the table style off by setting it to None:

```
worksheet.add_table('B3:F7', {'data': data, 'style': None})
```

23.10 name

By default tables are named `Table1`, `Table2`, etc. The `name` parameter can be used to set the name of the table:

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

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, see the Microsoft Office documentation.

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
header_format
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 can also be added.
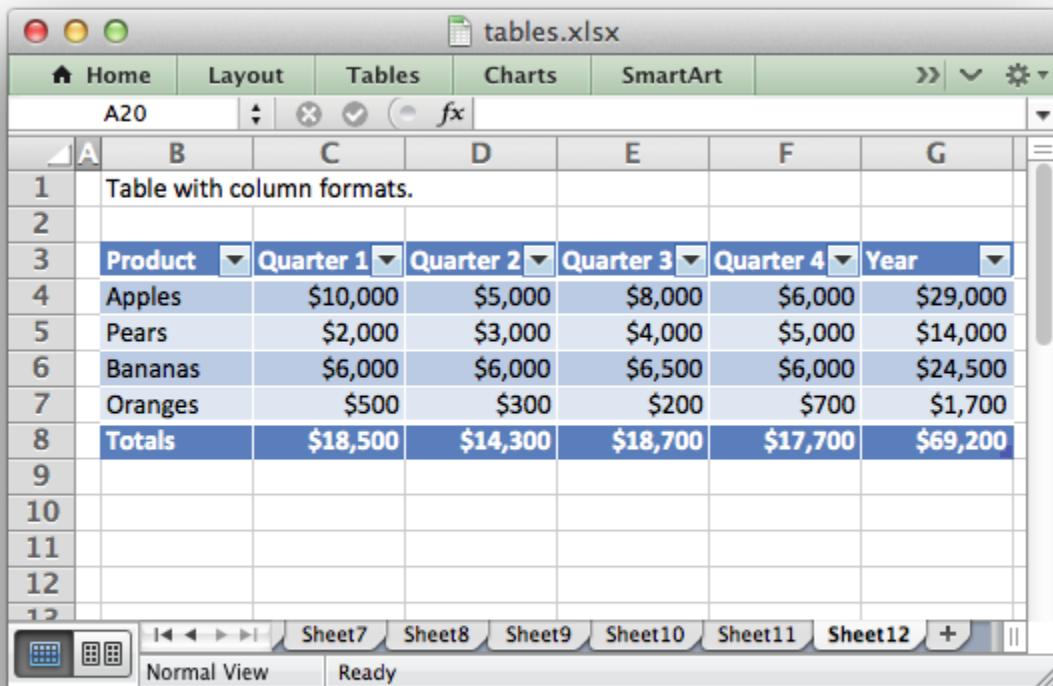
It is also possible to set a calculated value for the `total_function` using the `total_value` sub property. This is only necessary when creating workbooks for applications that cannot calculate the value of formulas automatically. This is similar to setting the `value` optional property in `write_formula()`:

```
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}]}
```

Formatting can also be applied to columns, to the column data using `format` and to the header using `header_format`:

```
currency_format = workbook.add_format({'num_format': '$#,##0'})
wrap_format     = workbook.add_format({'text_wrap': 1})

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',
                                           'header_format': wrap_format,
                                           'total_function': 'sum',
                                           'format': currency_format}]})
```


Standard XlsxWriter *Format object* objects are used for this formatting. However, they should be limited to numerical formats for the columns and simple formatting like text wrap for the headers. Overriding other table formatting may produce inconsistent results.

23.13 Example

All of the images shown above are taken from *Example: Worksheet Tables*.

WORKING WITH TEXTBOXES

This section explains how to work with some of the options and features of textboxes in XlsxWriter:
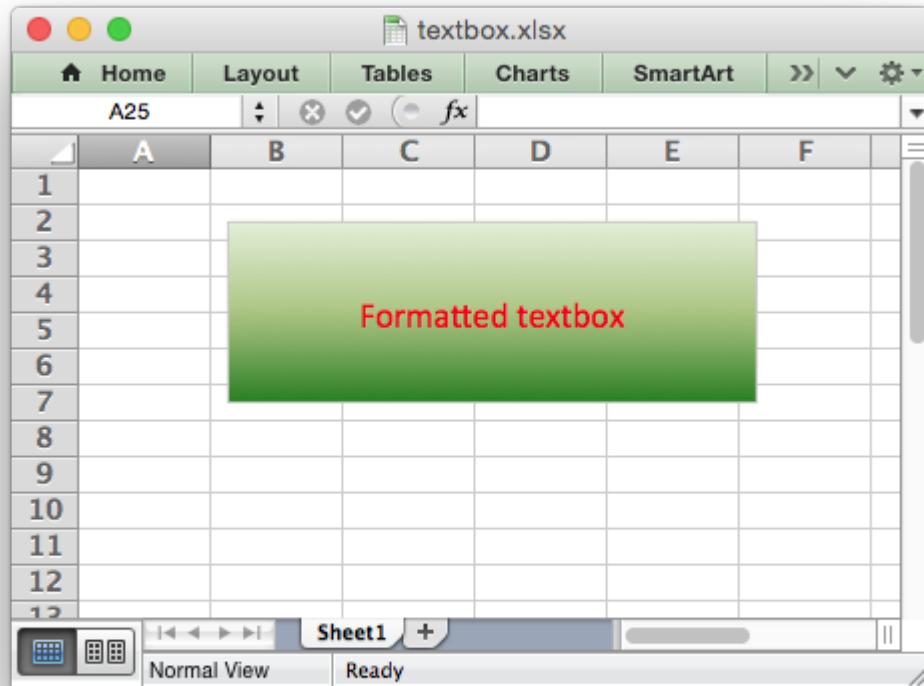
```python
import xlsxwriter

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

text = 'Formatted textbox'

options = {
    'width': 256,
    'height': 100,
    'x_offset': 10,
    'y_offset': 10,

    'font': {'color': 'red',
             'size': 14},
    'align': {'vertical': 'middle',
              'horizontal': 'center'
              },
    'gradient': {'colors': ['#DDEBCF',
                            '#9CB86E',
                            '#156B13']},
}

worksheet.insert_textbox('B2', text, options)

workbook.close()
```
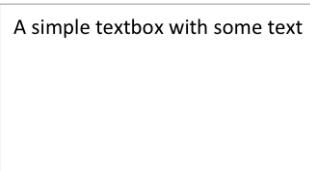
See also *Example: Insert Textboxes into a Worksheet*.

24.1 Textbox options

This Worksheet `insert_textbox()` method is used to insert a textbox into a worksheet:

```
worksheet.insert_textbox('B2', 'A simple textbox with some text')
```


The text can contain newlines to wrap the text:

```
worksheet.insert_textbox('B2', 'Line 1\nLine 2\n\nMore text')
```

```
Line 1
Line 2

More text
```

This `insert_textbox()` takes an optional `dict` parameter that can be used to control the size, positioning and format of the textbox:

```
worksheet.insert_textbox('B2', 'Some text', {'width': 256, 'height': 100})
```

The available options are:

```
# Size and position
width
height
x_scale
y_scale
x_offset
y_offset
object_position

# Formatting
line
border
fill
gradient
font
align
text_rotation

# Links
textlink
url
tip

# Accessibility
description
decorative
```

These options are explained in the sections below. They are similar or identical to position and formatting parameters used in charts.
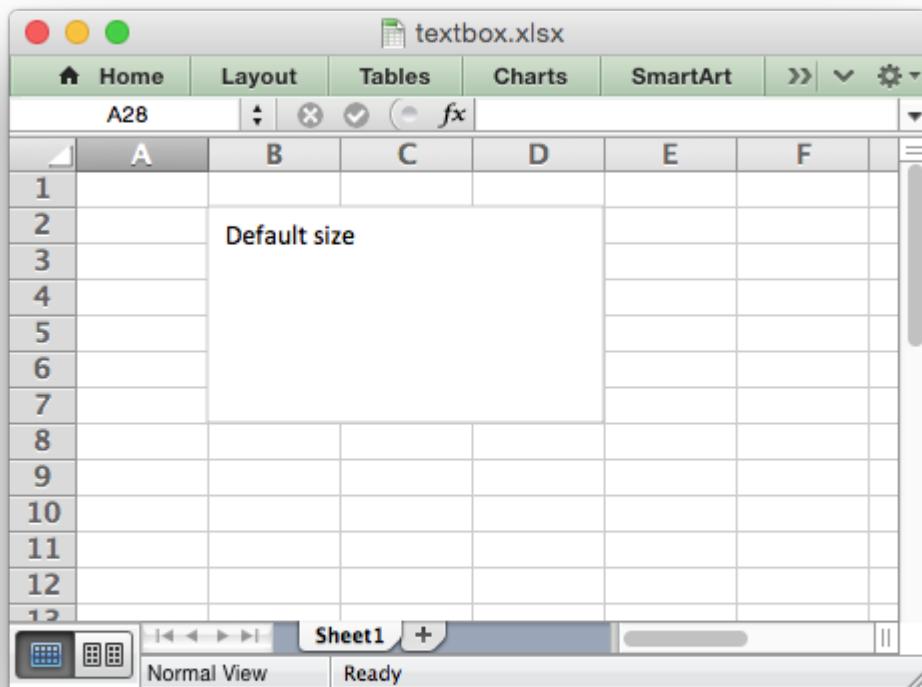
24.2 Textbox size and positioning

The `insert_textbox()` options to control the size and positioning of a textbox are:

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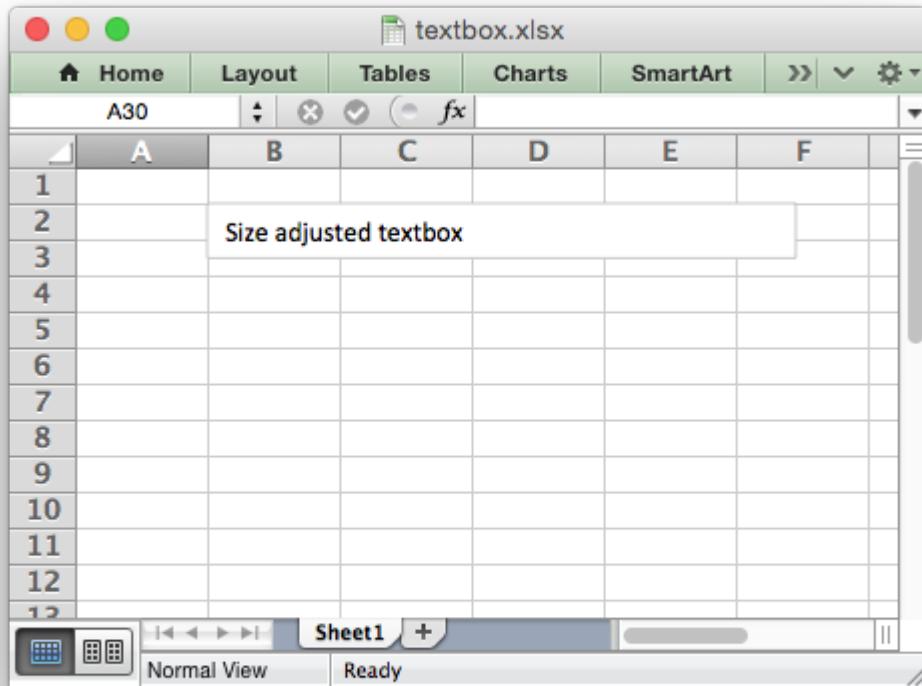
```
y_scale
x_offset
y_offset
object_position
```

The `width` and `height` are in pixels. The default textbox size is 192 x 120 pixels (or equivalent to 3 default columns x 6 default rows).
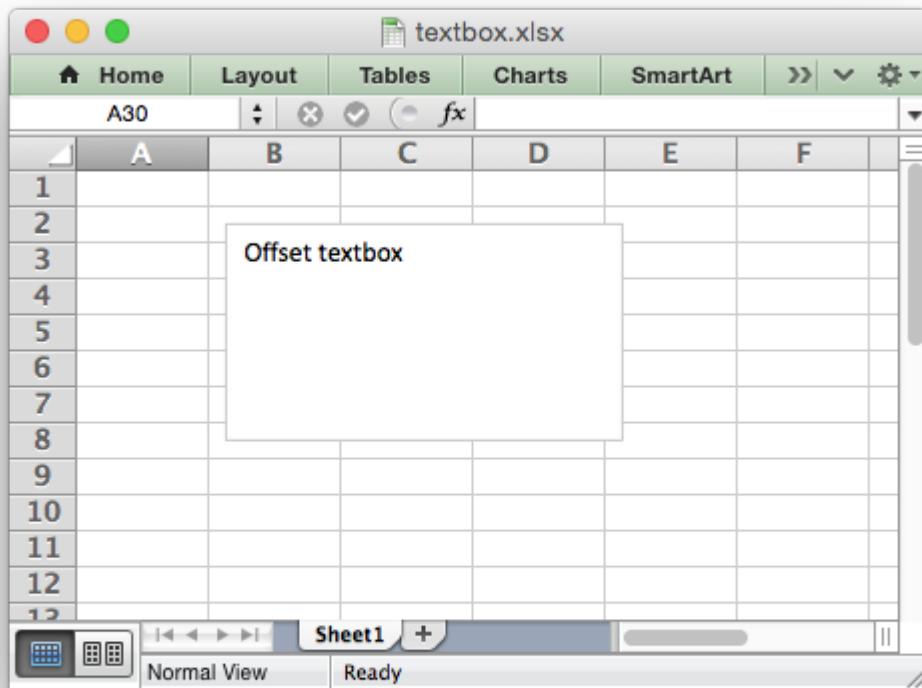
The size of the textbox can be modified by setting the `width` and `height` or by setting the `x_scale` and `y_scale`:

```
worksheet.insert_textbox('B2', 'Size adjusted textbox',
                         {'width': 288, 'height': 30})


# or ...
worksheet.insert_textbox('B2', 'Size adjusted textbox',
                         {'x_scale': 1.5, 'y_scale': 0.25})
```

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

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

```
worksheet.insert_textbox('B2', "Don't move or size with cells",
                         {'object_position': 3})
```

Where `object_position` has the following allowable values:

1. Move and size with cells (the default).

2. Move but don't size with cells.

3. Don't move or size with cells.

See *Working with Object Positioning* for more detailed information about the positioning and scaling of images within a worksheet.

24.3 Textbox Formatting

The following formatting properties can be set for textbox objects:

```
line
border
fill
```

```
gradient
font
align
text_rotation
```

Textbox formatting properties are set using the options dict:

```
worksheet.insert_textbox('B2', 'A textbox with a color text',
                         {'font': {'color': 'green'}})
```

A textbox with a color text

In some cases the format properties can be nested:

```
worksheet.insert_textbox('B2', 'Some text in a textbox with formatting',
                         {'font': {'color': 'white'},
                          'align': {'vertical': 'middle',
                                    'horizontal': 'center'
                                   },
                          'gradient': {'colors': ['green', 'white']}})
```

Some text in a textbox with formatting

24.4 Textbox formatting: Line

The line format is used to specify properties of the border in a textbox. The following properties can be set for `line` formats in a textbox:

```
none
color
width
dash_type
```

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

```
worksheet.insert_textbox('B2', 'A textbox with no border line',
                         {'line': {'none': True}})
```

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

```
worksheet.insert_textbox('B2', 'A textbox with a color border',
                         {'line': {'color': 'red'}})
```

A textbox with a color border

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

```
worksheet.insert_textbox('B2', 'A textbox with a color border',
                         {'line': {'color': '#FF9900'}})
```

A textbox with a color border

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

```
worksheet.insert_textbox('B2', 'A textbox with larger border',
                         {'line': {'width': 3.25}})
```

A textbox with larger border

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

```
worksheet.insert_textbox('B2', 'A textbox a dash border',
                         {'line': {'dash_type': 'dash_dot'}})
```

A textbox a dash border
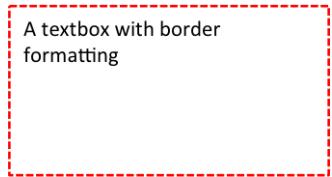
The following `dash_type` values are available. They are shown in the order that they appear in the Excel dialog:

```
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:

```
worksheet.insert_textbox('B2', 'A textbox with border formatting',
                         {'line': {'color': 'red',
                                   'width': 1.25,
                                   'dash_type': 'square_dot'}})
```

A textbox with border
formatting

24.5 Textbox formatting: Border

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

Excel uses a common dialog for setting object formatting but depending on context it may refer to a *line* or a *border*. For formatting these can be used interchangeably.

24.6 Textbox formatting: Solid Fill
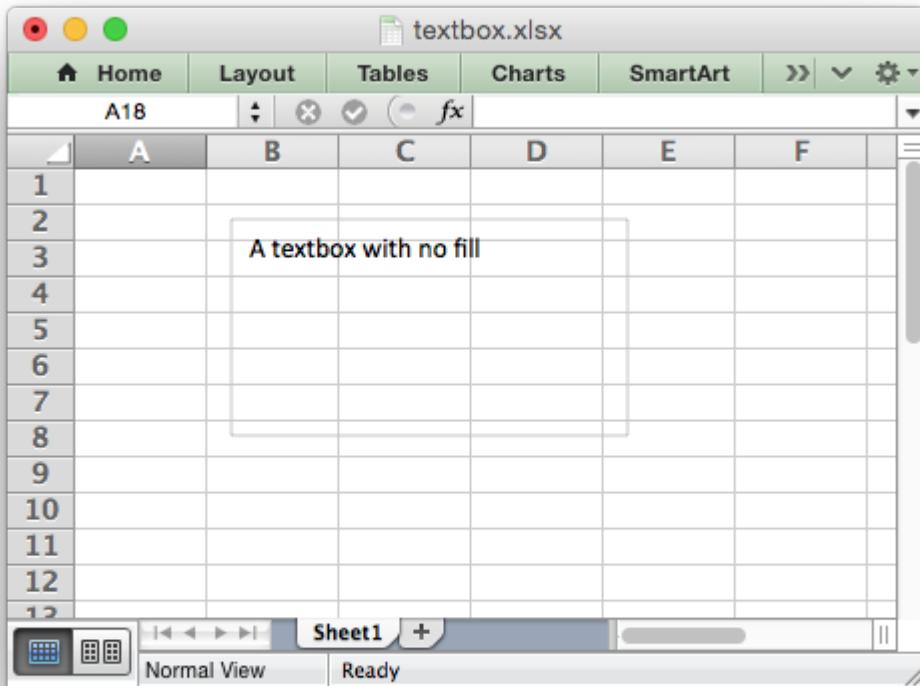
The solid fill format is used to specify a fill for a textbox object.

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

```
none
color
```
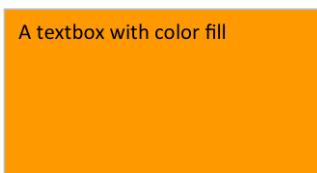
The none property is used to turn the `fill` property off (to make the textbox transparent):

```
worksheet.insert_textbox('B2', 'A textbox with no fill',
                         {'fill': {'none': True}})
```

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

```
worksheet.insert_textbox('B2', 'A textbox with color fill',
                         {'fill': {'color': '#FF9900'}})
```


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

```
worksheet.insert_textbox('B2', 'A textbox with color fill',
                         {'fill': {'color': 'red'}})
```

24.7 Textbox formatting: Gradient Fill

The gradient fill format is used to specify a gradient fill for a textbox. The following properties can be set for `gradient` fill formats in a textbox:

```
colors:    a list of colors
positions: an optional list of positions for the colors
type:      the optional type of gradient fill
angle:     the optional angle of the linear fill
```

If gradient fill is used on a textbox object it overrides the solid fill properties of the object.

The `colors` property sets a list of colors that define the `gradient`:
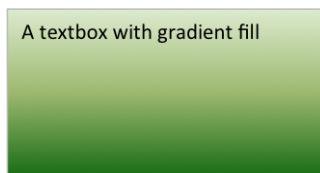
```
worksheet.insert_textbox('B2', 'A textbox with gradient fill',
                         {'gradient': {'colors': ['gray', 'white']}})
```


A textbox with gradient fill

Excel allows between 2 and 10 colors in a gradient but it is unlikely that you will require more than 2 or 3.

As with solid fill it is also possible to set the colors of a gradient with a Html style #RRGGBB string or a limited number of named colors, see *Working with Colors*:

```
worksheet.insert_textbox('B2', 'A textbox with gradient fill',
                         {'gradient': {'colors': ['#DDEBCF',
                                                  '#9CB86E',
                                                  '#156B13']}})
```


A textbox with gradient fill

The `positions` defines an optional list of positions, between 0 and 100, of where the colors in the gradient are located. Default values are provided for `colors` lists of between 2 and 4 but they can be specified if required:

```
worksheet.insert_textbox('B2', 'A textbox with gradient fill',
                         {'gradient': {'colors':    ['#DDEBCF', '#156B13'],
                                       'positions': [10,        90]}})
```

The `type` property can have one of the following values:

```
linear        (the default)
radial
rectangular
path
```

For example:

```
worksheet.insert_textbox('B2', 'A textbox with gradient fill',
                            {'gradient': {'colors': ['#DDEBCF', '#9CB86E', '#156B13'],
                                          'type': 'radial'}})
```

A textbox with radial gradient

If `type` isn't specified it defaults to `linear`.

For a `linear` fill the angle of the gradient can also be specified (the default angle is 90 degrees):

```
worksheet.insert_textbox('B2', 'A textbox with angle gradient',
                            {'gradient': {'colors': ['#DDEBCF', '#9CB86E', '#156B13'],
                                          'angle': 45}})
```

24.8 Textbox formatting: Fonts

The following font properties can be set for the entire textbox:
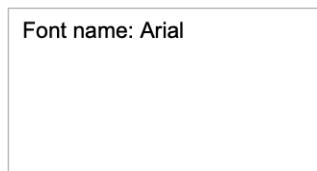
```
name
size
bold
italic
underline
color
```

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

The font properties are:
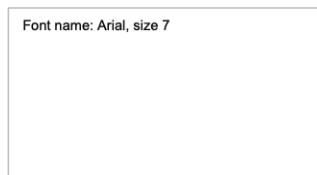
• `name`: Set the font name:

```
{'font':  {'name': 'Arial'}}
```

Font name: Arial

• `size`: Set the font size:

```
{'font':  {'name': 'Arial', 'size': 7}}
```

Font name: Arial, size 7

- `bold`: Set the font bold property:

    ```
    {'font':  {'bold': True}}
    ```


    Font properties: bold
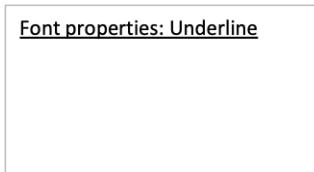
- `italic`: Set the font italic property:

    ```
    {'font':  {'italic': True}}
    ```


    Font properties: Italic

- `underline`: Set the font underline property:

    ```
    {'font':  {'underline': True}}
    ```


    Font properties: Underline

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

    ```
    {'font': {'color': 'red' }}
    {'font': {'color': '#92D050'}}
    ```

Here is an example of Font formatting in a textbox:
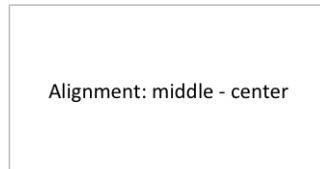
```
worksheet.insert_textbox('B2', 'Some font formatting',
                         {'font': {'bold': True,
                                   'italic': True,
                                   'underline': True,
                                   'name': 'Arial',
                                   'color': 'red',
                                   'size': 14}})
```


Some font formatting

24.9 Textbox formatting: Align

The `align` property is used to set the text alignment for the entire textbox:

```
worksheet.insert_textbox('B2', 'Alignment: middle - center',
                         {'align': {'vertical': 'middle',
                                    'horizontal': 'center'}})
```

Alignment: middle - center

The alignment properties that can be set in Excel for a textbox are:
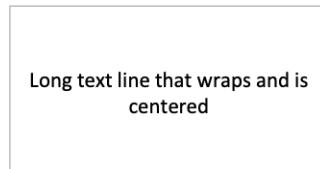
```
{'align': {'vertical': 'top'}}       # Default
{'align': {'vertical': 'middle'}}
{'align': {'vertical': 'bottom'}}

{'align': {'horizontal': 'left'}}    # Default
{'align': {'horizontal': 'center'}}

{'align': {'text': 'left'}}          # Default
{'align': {'text': 'center'}}
{'align': {'text': 'right'}}
```

The `vertical` and `horizontal` alignments set the layout for the text area within the textbox. The `text` alignment sets the layout for the text within that text area:

```
worksheet.insert_textbox('H2',
                         'Long text line that wraps and is centered',
                         {'align': {'vertical': 'middle',
                                    'horizontal': 'center',
                                    'text': 'center'}})
```

Long text line that wraps and is centered

The default textbox alignment is:

```
worksheet.insert_textbox('B2', 'Default alignment',
                         {'align': {'vertical': 'top',
                                    'horizontal': 'left',
                                    'text': 'left'}})

# Same as this:
worksheet.insert_textbox('B2', 'Default alignment')
```

Default alignment

24.10 Textbox formatting: Text Rotation

The `text_rotation` option can be used to set the text rotation for the entire textbox:

```
worksheet.insert_textbox('B2', 'Text rotated up',
                         {'text_rotation': 90})
```

Text rotated up

Textboxes in Excel only support a limited number of rotation options. These are:

```
 90:      Rotate text up
-90:      Rotate text down
270:      Vertical text (stacked)
271:      Vertical text (stacked) - for East Asian fonts
```
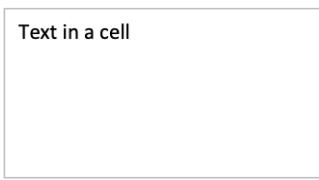
24.11 Textbox Textlink

The `textlink` property is used to link/get the text for a textbox from a cell in the worksheet. When you use this option the actual text in the textbox can be left blank or set to None:

```
worksheet.insert_textbox('A1', '', {'textlink': '=$A$1'})
```

The reference can also be to a cell in another worksheet:

```
worksheet.insert_textbox('A2', None, {'textlink': '=Sheet2!A1'})
```

Text in a cell

24.12 Textbox Hyperlink

The `url` parameter can used to add a hyperlink/url to a textbox:

```
worksheet.insert_textbox('A1', 'This is some text',
                         {'url': 'https://github.com/jmcnamara'})
```

The `tip` parameter adds an optional mouseover tooltip:
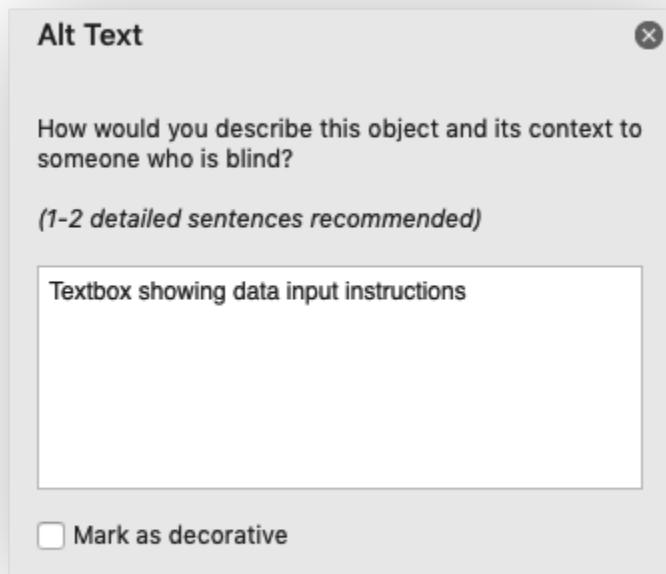
```
worksheet.insert_textbox('A1', 'This is some text',
                         {'url': 'https://github.com/jmcnamara',
                          'tip': 'GitHub'})
```

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

24.13 Textbox Description

The `description` property can be used to specify a description or "alt text" string for the textbox. In general this would be used to provide a text description of the textbox to help accessibility. It is an optional parameter and has no default. It can be used as follows:

```
worksheet.insert_textbox('A1', 'This is some text',
                         {'description': 'Textbox showing data input instructions'})
```


24.14 Textbox Decorative

The optional `decorative` property is also used to help accessibility. It is used to mark the object as decorative, and thus uninformative, for automated screen readers. As in Excel, if this parameter is in use the `description` field isn't written. It is used as follows:

```
worksheet.insert_textbox('A1', 'This is some text', {'decorative': True})
```
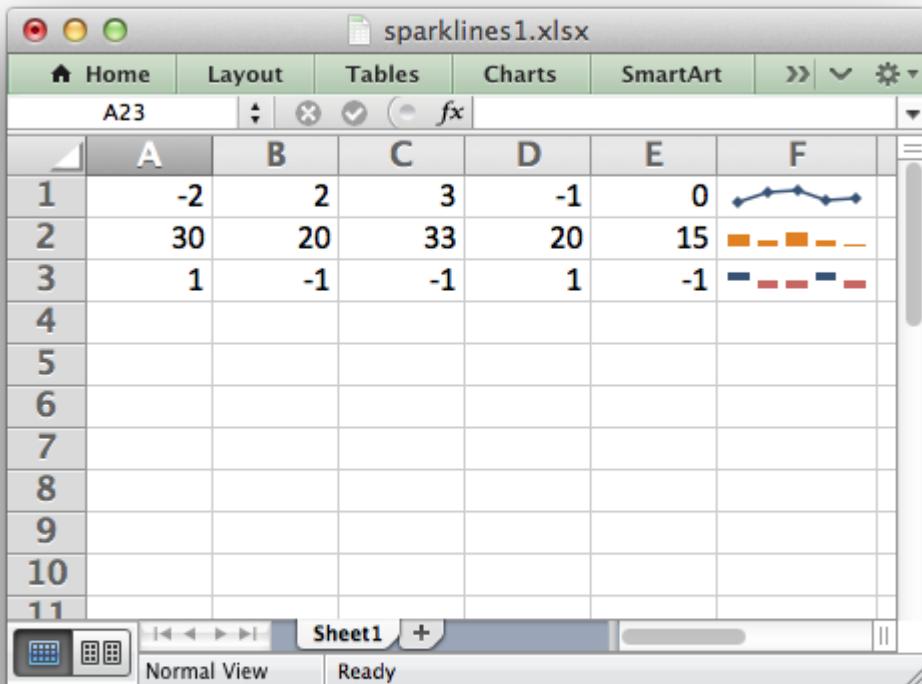
WORKING WITH SPARKLINES

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: https://en.wikipedia.org/wiki/Sparklines

25.1 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'})
```
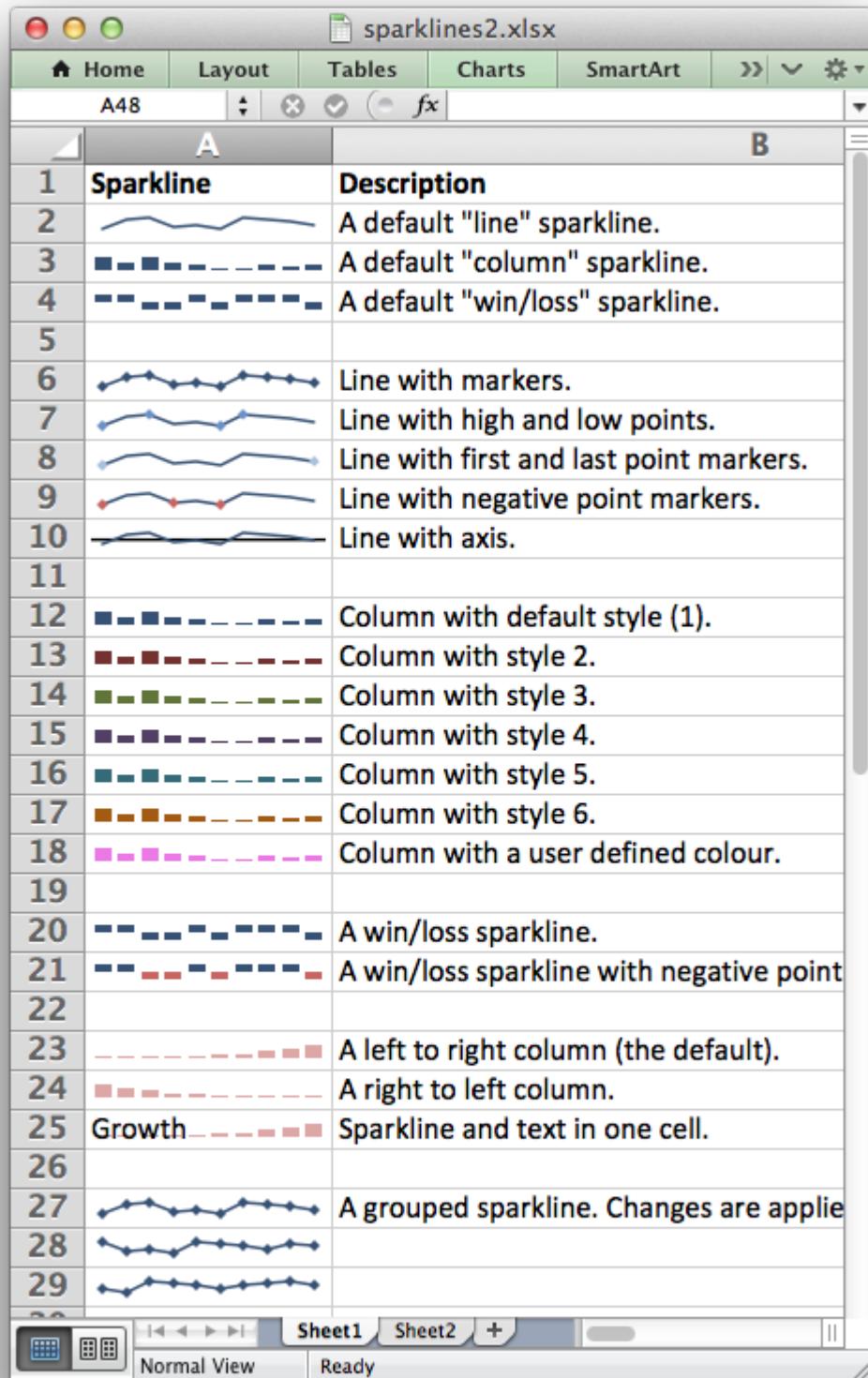
Both row-column and A1 style notation are supported. See *Working with Cell Notation* for more details.

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
```

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

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.

25.2 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"})
```

25.3 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'})
```

25.4 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 color elements of the sparklines using the `_color` parameters below.

25.5 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.

25.6 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})
```

25.7 axis

Display a horizontal axis in the sparkline:

```
worksheet.add_sparkline('A10', {'range': 'Sheet2!A1:J1',
                                 'axis': True})
```

25.8 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})
```

25.9 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
```

25.10 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})
```

25.11 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.

25.12 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).

25.13 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.

25.14 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.

25.15 series_color

It is possible to override the color 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'})
```

25.16 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.

25.17 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']})
```

25.18 Sparkline examples
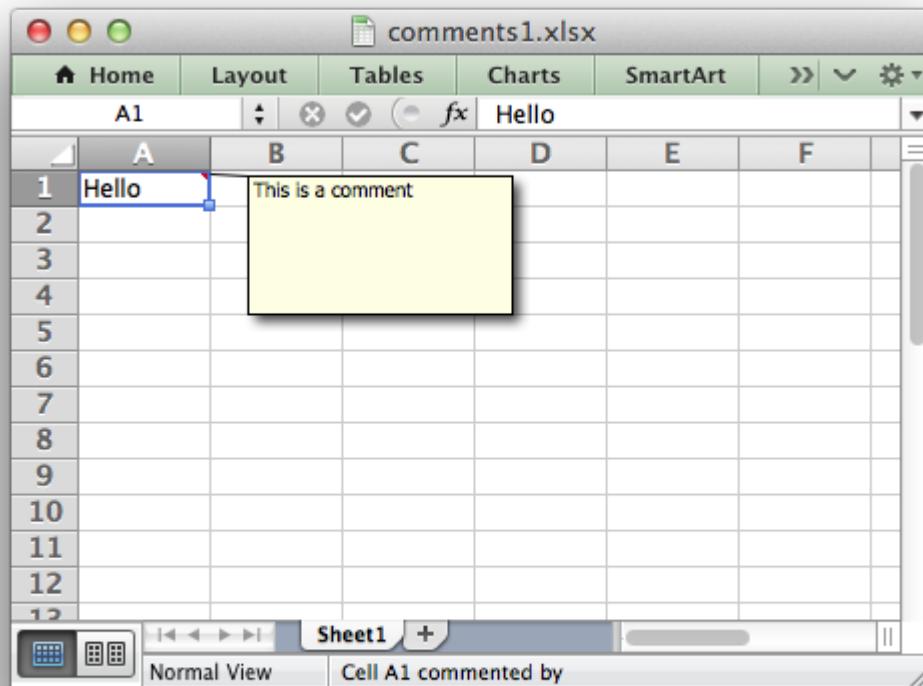
See *Example: Sparklines (Simple)* and *Example: Sparklines (Advanced)*.

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')
```


26.1 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
font_name
font_size
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 behavior 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 color of cell comment box. You can use one of the named colors recognized by XlsxWriter or a Html color. See *Working with Colors*:

```
worksheet.write_comment('C3', 'Hello', {'color': 'green'  })
worksheet.write_comment('C4', 'Hello', {'color': '#CCFFCC'})
```

- font_name: This option is used to set the font for the comment:

```
worksheet.write_comment('C3', 'Hello', {'font_name': 'Courier'})
```

The default font is 'Tahoma'.

- font_size: This option is used to set the font size for the comment:

```
worksheet.write_comment('C3', 'Hello', {'font_size': 10})
```

The default font size is 8.

- 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 behavior 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)*.
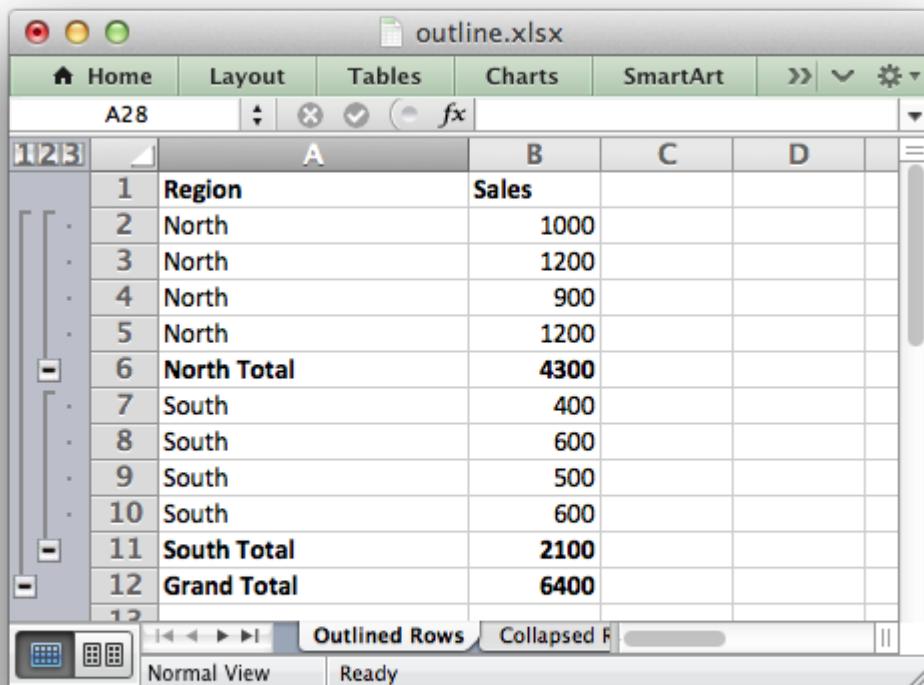
TWENTYSEVEN

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 to 11 are grouped at level 1 and rows 2 to 5 and 7 to 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.
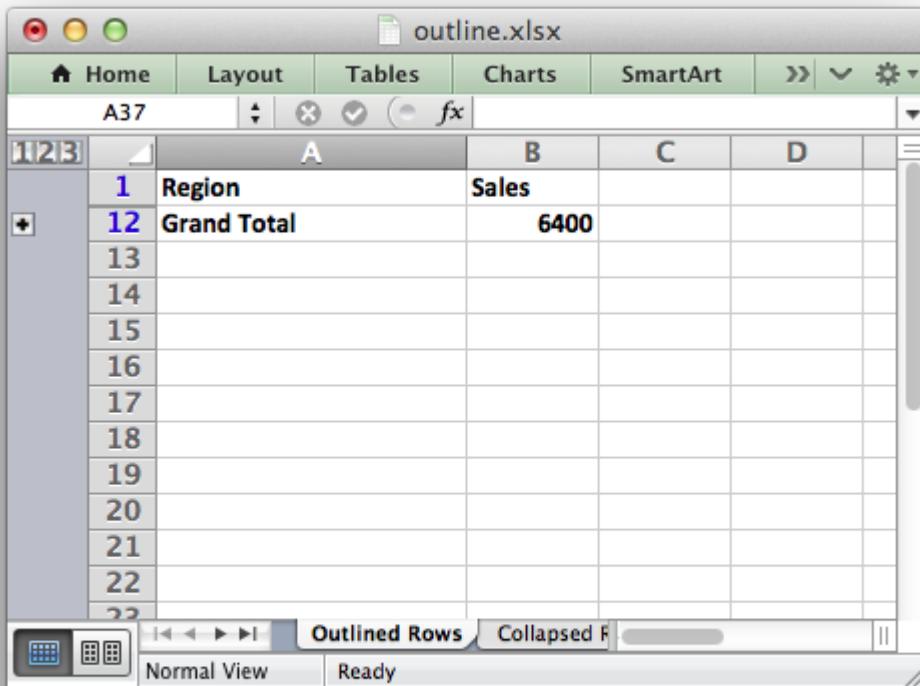
The minus sign changes to a plus sign to indicate that the data in the outline is hidden. This shows the usefulness of outlines: with 2 mouse clicks we have reduced the amount of visual data down to 2 sub-totals and the overall total.

Finally, clicking on the minus sign on the level 1 outline will collapse the remaining rows as follows:

27.1 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, cell_format, options)
worksheet.set_column(first_col, last_col, width, cell_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:
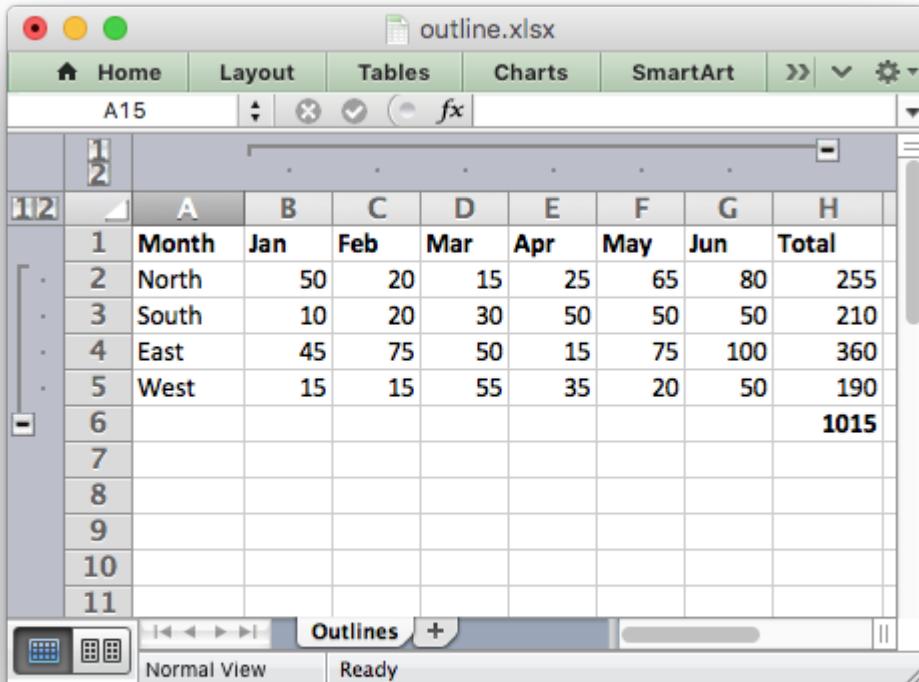
```
worksheet.set_row(0, 20, cell_format, {'hidden': True})

# 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 to 4 (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_row(3, None, None, {'level': 1})
worksheet.set_row(4, None, None, {'level': 1})

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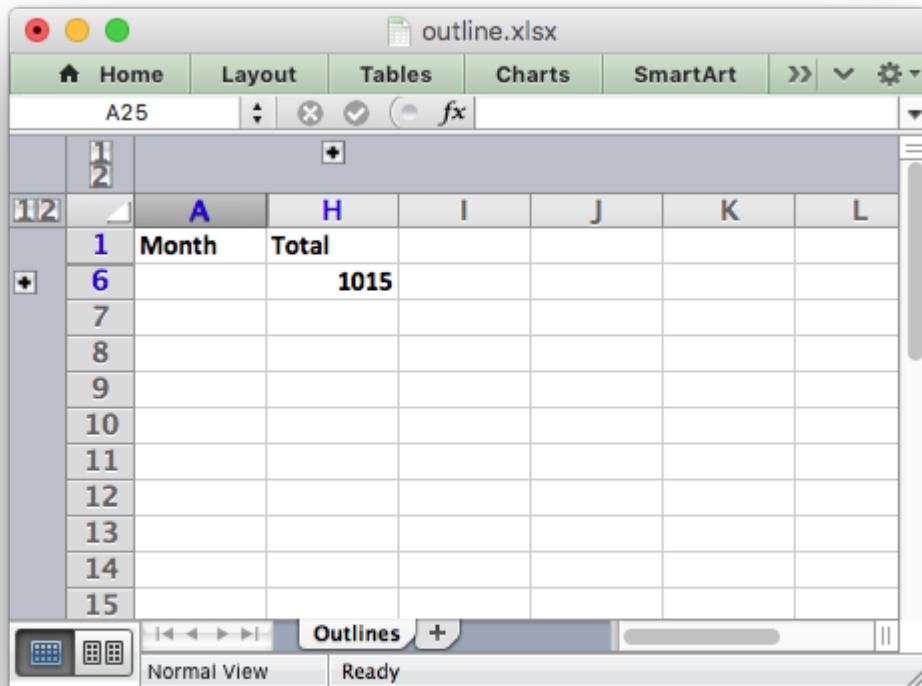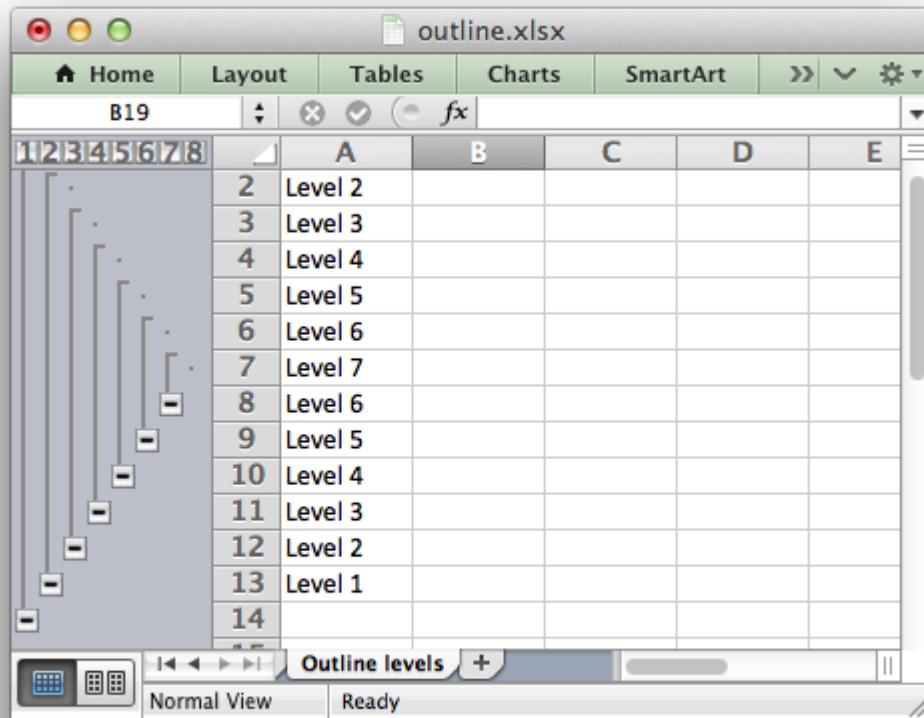

Rows and columns can be collapsed by setting the `hidden` flag for the hidden rows/columns and setting the `collapsed` flag for the row/column that has the collapsed '+' symbol:

```
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, {'level': 1, 'hidden': True})
worksheet.set_row(4, None, None, {'level': 1, 'hidden': True})
worksheet.set_row(5, None, None, {'collapsed': True})

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

Excel allows up to 7 outline levels. Therefore the `level` parameter should be in the range 0 <= level <= 7.

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.

TWENTYEIGHT

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 optimization 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:

```
# Ok. 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)

# Not ok. With 'constant_memory' this will 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 optimization 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.

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 add_table() method doesn't work in this mode and merge_range() and set_row() only work for the current row.

28.1 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:
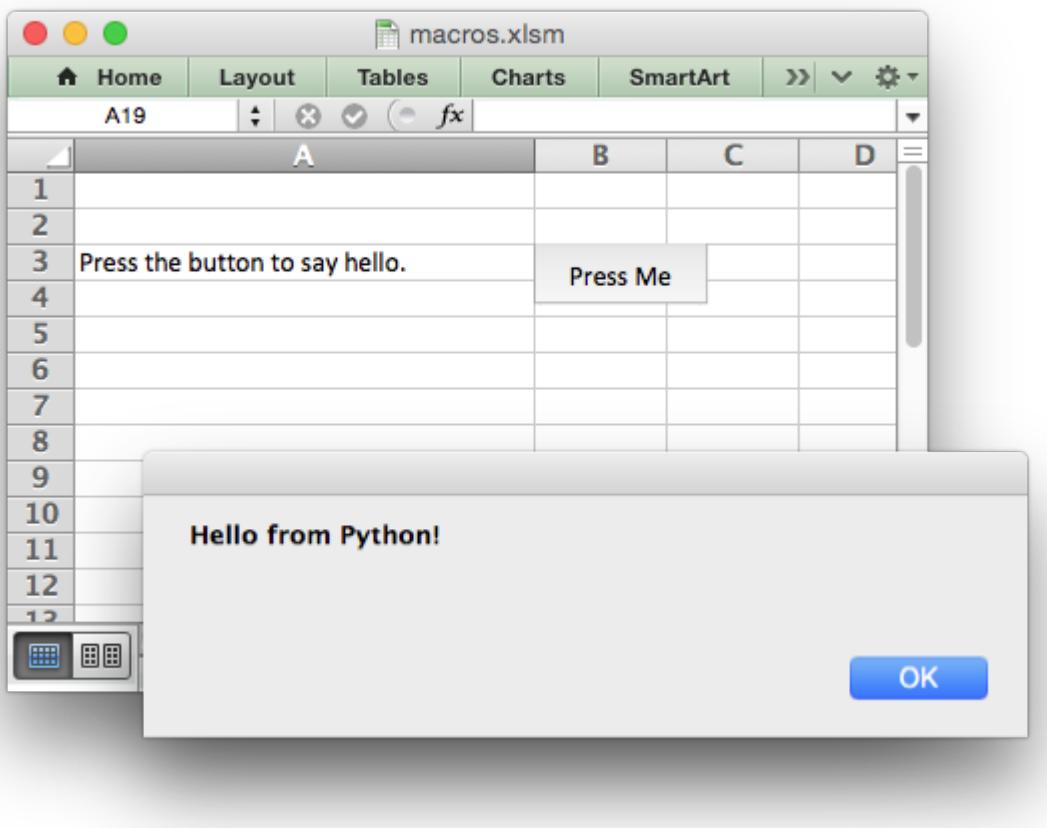
Rows	Columns	Time (s)	Memory (bytes)
200	50	0.37	62208
400	50	0.74	62208
800	50	1.46	62208
1600	50	2.93	62208
3200	50	5.90	62208
6400	50	11.84	62208
12800	50	23.63	62208

In `constant_memory` mode the performance should be approximately the same as normal mode.

These figures were generated using programs in the `dev/performance` directory of the XlsxWriter repo.

WORKING WITH VBA MACROS

This section explains how to add a VBA file containing functions or macros to an XlsxWriter file.

29.1 The Excel XLSM file format

An Excel `xlsm` file is exactly the same as an `xlsx` file except that is contains an additional `vbaProject.bin` file which contains functions and/or macros. Excel uses a different extension to differentiate between the two file formats since files containing macros are usually subject to additional security checks.

29.2 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 a 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.

29.3 The vba_extract.py utility

The `vba_extract.py` 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
```

If the VBA project is signed, `vba_extract.py` also extracts the `vbaProjectSignature.bin` file from the xlsm file.

29.4 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:

```python
import xlsxwriter

# Note the file extension should be .xlsm.
workbook = xlsxwriter.Workbook('macros.xlsm')
worksheet = workbook.add_worksheet()

worksheet.set_column('A:A', 30)
```

```python
    # 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:

```python
    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.

29.5 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:

```python
    # Note: set codename for workbook and any worksheets.
    workbook.set_vba_name('MyWorkbook')
    worksheet1.set_vba_name('MySheet1')
    worksheet2.set_vba_name('MySheet2')
```

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 particularly important for macros created with non-English versions of Excel.

29.6 Adding a VBA macro signature file to an XlsxWriter file

VBA macros can be signed in Excel to allow for blocking execution of unsigned macros in certain environments.

The `vba_extract.py` utility can be used to extract the `vbaProject.bin` and `vbaPro-jectSignature.bin` files from an existing xlsm file with signed macros.

To add these files to the XlsxWriter workbook using the `add_signed_vba_project()` method:

```
workbook.add_signed_vba_project("./vbaProject.bin", "./vbaProjectSignature.bin");
```

29.7 What to do if it doesn't work

The XlsxWriter test suite contains several tests to ensure that this feature works 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. 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, or the macro was extracted from an non-English language version of Excel, then you can specify these using the workbook and worksheet `set_vba_name()` methods:

   ```
   # Note: set codename for workbook and any worksheets.
   workbook.set_vba_name('MyWorkbook')
   worksheet1.set_vba_name('MySheet1')
   worksheet2.set_vba_name('MySheet2')
   ```

3. 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.

WORKING WITH PANDAS AND XLSXWRITER

Python Pandas is a Python data analysis library. It can read, filter and re-arrange small and large data sets and output them in a range of formats including Excel.

Pandas writes Excel xlsx files using either openpyxl or XlsxWriter.

30.1 Using XlsxWriter with Pandas

To use XlsxWriter with Pandas you specify it as the Excel writer *engine*:
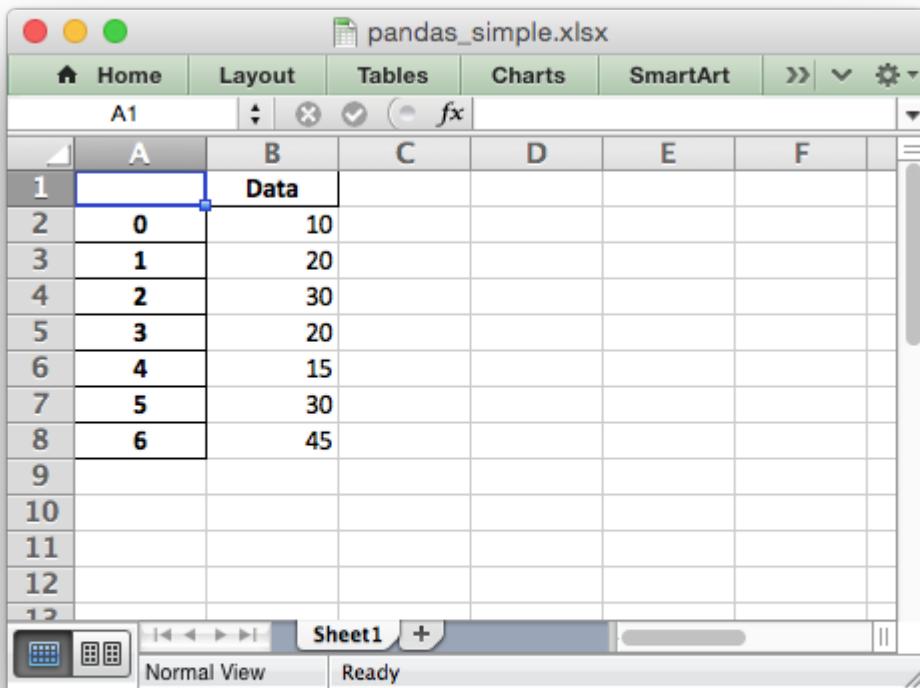
```python
import pandas as pd

# Create a Pandas dataframe from the data.
df = pd.DataFrame({'Data': [10, 20, 30, 20, 15, 30, 45]})

# Create a Pandas Excel writer using XlsxWriter as the engine.
writer = pd.ExcelWriter('pandas_simple.xlsx', engine='xlsxwriter')

# Convert the dataframe to an XlsxWriter Excel object.
df.to_excel(writer, sheet_name='Sheet1')

# Close the Pandas Excel writer and output the Excel file.
writer.close()
```

The output from this would look like the following:

See the full example at *Example: Pandas Excel example*.

30.2 Accessing XlsxWriter from Pandas

In order to apply XlsxWriter features such as Charts, Conditional Formatting and Column Format-
ting to the Pandas output we need to access the underlying *workbook* and *worksheet* objects.
After that we can treat them as normal XlsxWriter objects.

Continuing on from the above example we do that as follows:

```python
import pandas as pd

# Create a Pandas dataframe from the data.
df = pd.DataFrame({'Data': [10, 20, 30, 20, 15, 30, 45]})

# Create a Pandas Excel writer using XlsxWriter as the engine.
writer = pd.ExcelWriter('pandas_simple.xlsx', engine='xlsxwriter')

# Convert the dataframe to an XlsxWriter Excel object.
df.to_excel(writer, sheet_name='Sheet1')

# Get the xlsxwriter objects from the dataframe writer object.
```

```
workbook  = writer.book
worksheet = writer.sheets['Sheet1']
```

This is equivalent to the following code when using XlsxWriter on its own:

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

The Workbook and Worksheet objects can then be used to access other XlsxWriter features, see below.

30.3 Adding Charts to Dataframe output

Once we have the Workbook and Worksheet objects, as shown in the previous section, we we can use them to apply other features such as adding a chart:

```
# Get the xlsxwriter objects from the dataframe writer object.
workbook  = writer.book
worksheet = writer.sheets['Sheet1']

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

# Get the dimensions of the dataframe.
(max_row, max_col) = df.shape

# Configure the series of the chart from the dataframe data.
chart.add_series({'values': ['Sheet1', 1, 1, max_row, 1]})

# Insert the chart into the worksheet.
worksheet.insert_chart(1, 3, chart)
```
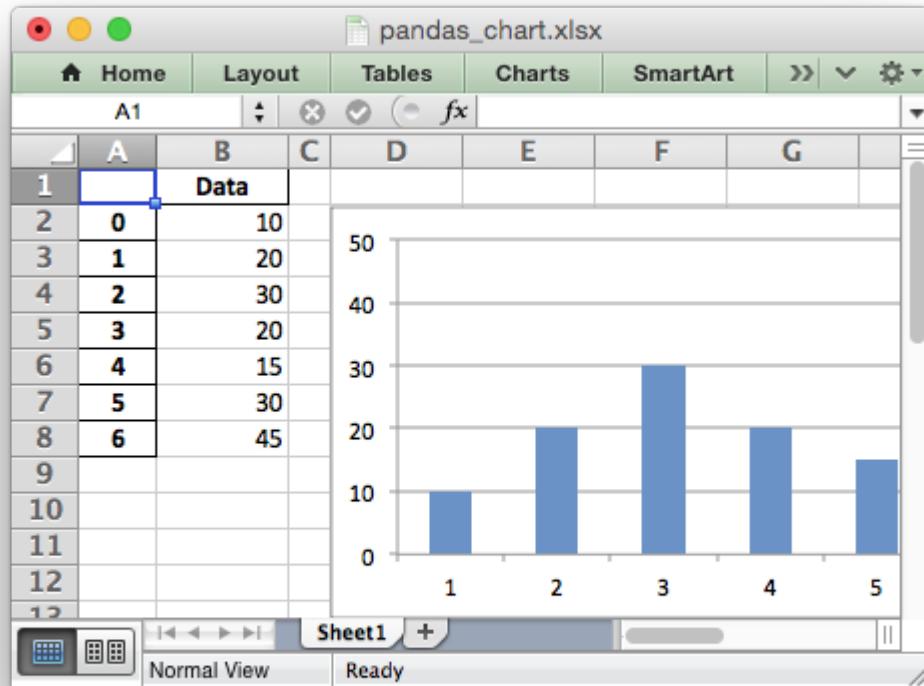
The output would look like this:
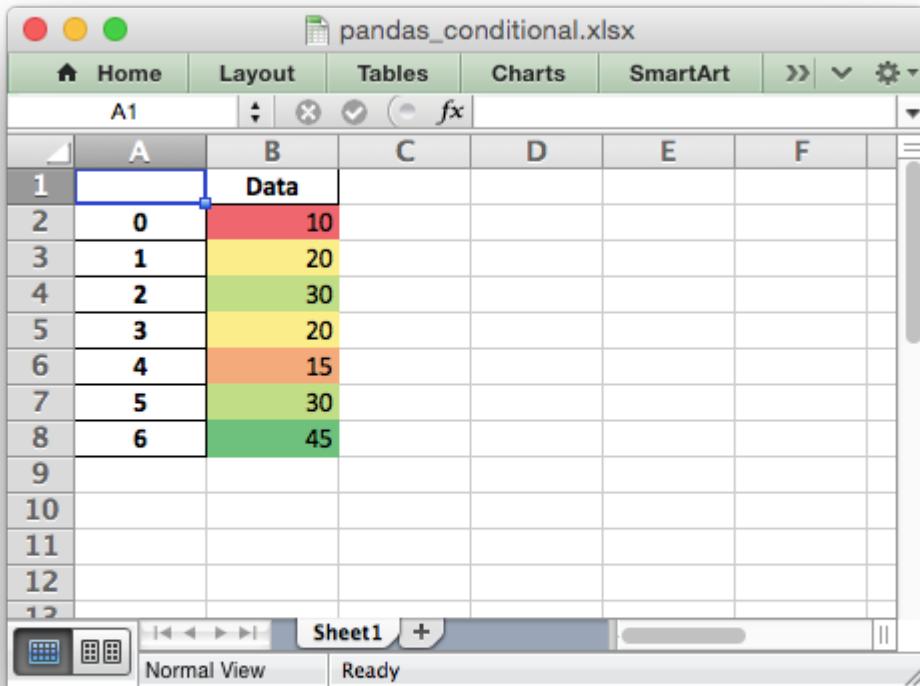
See the full example at *Example: Pandas Excel output with a chart*.

30.4 Adding Conditional Formatting to Dataframe output

Another option is to apply a conditional format like this:

```
# Apply a conditional format to the required cell range.
worksheet.conditional_format(1, max_col, max_row, max_col,
                             {'type': '3_color_scale'})
```

Which would give:

See the full example at *Example: Pandas Excel output with conditional formatting* and the section of the docs on *Working with Conditional Formatting*.

30.5 Formatting of the Dataframe output

XlsxWriter and Pandas provide very little support for formatting the output data from a dataframe apart from default formatting such as the header and index cells and any cells that contain dates or datetimes. In addition it isn't possible to format any cells that already have a default format applied.

If you require very controlled formatting of the dataframe output then you would probably be better off using Xlsxwriter directly with raw data taken from Pandas. However, some formatting options are available.

For example it is possible to set the default date and datetime formats via the Pandas interface:

```
writer = pd.ExcelWriter("pandas_datetime.xlsx",
                        engine='xlsxwriter',
                        datetime_format='mmm d yyyy hh:mm:ss',
                        date_format='mmmm dd yyyy')
```

Which would give:

See the full example at *Example: Pandas Excel output with datetimes*.

It is possible to format any other, non date/datetime column data using `set_column()`:

```
# Add some cell formats.
format1 = workbook.add_format({'num_format': '#,##0.00'})
format2 = workbook.add_format({'num_format': '0%'})

# Set the column width and format.
worksheet.set_column(1, 1, 18, format1)

# Set the format but not the column width.
worksheet.set_column(2, 2, None, format2)
```

See the full example at *Example: Pandas Excel output with column formatting*.

30.6 Formatting of the Dataframe headers

Pandas writes the dataframe header with a default cell format. Since it is a cell format it cannot be overridden using `set_row()`. If you wish to use your own format for the headings then the best approach is to turn off the automatic header from Pandas and write your own. For example:

```
# Turn off the default header and skip one row to allow us to insert a
# user defined header.
df.to_excel(writer, sheet_name='Sheet1', startrow=1, header=False)

# Get the xlsxwriter workbook and worksheet objects.
workbook  = writer.book
worksheet = writer.sheets['Sheet1']

# Add a header format.
header_format = workbook.add_format({
    'bold': True,
    'text_wrap': True,
    'valign': 'top',
    'fg_color': '#D7E4BC',
```

```
        'border': 1})

    # Write the column headers with the defined format.
    for col_num, value in enumerate(df.columns.values):
        worksheet.write(0, col_num + 1, value, header_format)
```


See the full example at *Example: Pandas Excel output with user defined header format*.

30.7 Adding a Dataframe to a Worksheet Table

As explained in *Working with Worksheet Tables*, tables in Excel are a way of grouping a range of cells into a single entity, like this:

The way to do this with a Pandas dataframe is to first write the data without the index or header, and by starting 1 row forward to allow space for the table header:

```
df.to_excel(writer, sheet_name='Sheet1',
            startrow=1, header=False, index=False)
```

We then create a list of headers to use in add_table():

```
column_settings = [{'header': column} for column in df.columns]
```

Finally we add the Excel table structure, based on the dataframe *shape* and with the column headers we generated from the dataframe columns:

```
(max_row, max_col) = df.shape
```

```
worksheet.add_table(0, 0, max_row, max_col - 1, {'columns': column_settings})
```

See the full example at *Example: Pandas Excel output with a worksheet table*.

30.8 Adding an autofilter to a Dataframe output

As explained in *Working with Autofilters*, autofilters in Excel are a way of filtering a 2d range of data to only display rows that match a user defined criteria.

The way to do this with a Pandas dataframe is to first write the data without the index (unless you want to include it in the filtered data):

```
df.to_excel(writer, sheet_name='Sheet1', index=False)
```

We then get the dataframe *shape* and add the autofilter:

```
worksheet.autofilter(0, 0, max_row, max_col - 1)
```


We can also add an optional 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')
```

However, it isn't enough to just apply the criteria. The rows that don't match must also be hidden. We use Pandas to figure our which rows to hide:

```
for row_num in (df.index[(df['Region'] != 'East')].tolist()):
    worksheet.set_row(row_num + 1, options={'hidden': True})
```

This gives us a filtered worksheet like this:

See the full example at *Example: Pandas Excel output with an autofilter*.

30.9 Handling multiple Pandas Dataframes

It is possible to write more than one dataframe to a worksheet or to several worksheets. For example to write multiple dataframes to multiple worksheets:
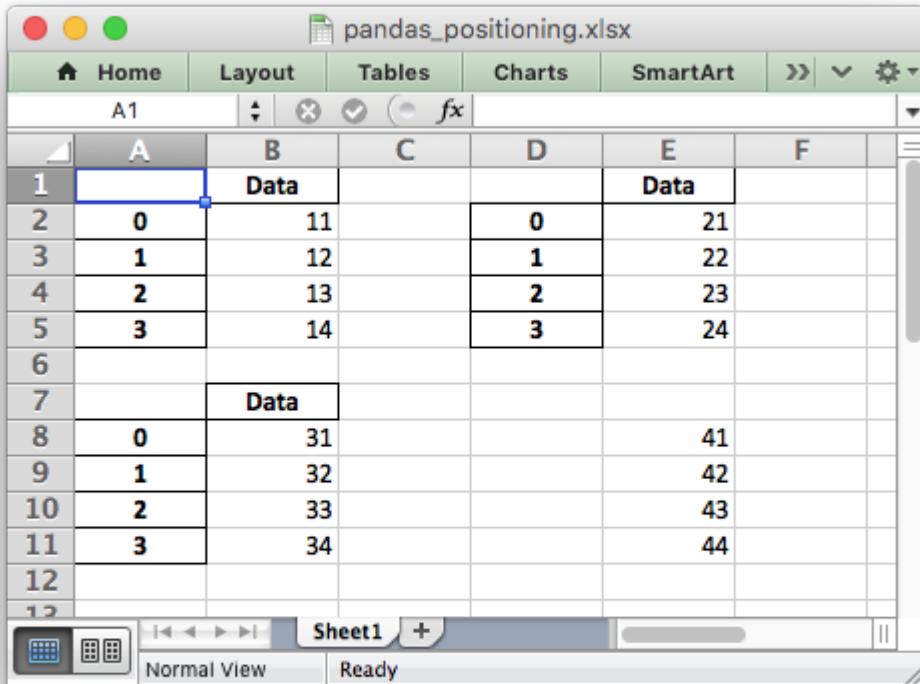
```python
# Write each dataframe to a different worksheet.
df1.to_excel(writer, sheet_name='Sheet1')
df2.to_excel(writer, sheet_name='Sheet2')
df3.to_excel(writer, sheet_name='Sheet3')
```

See the full example at *Example: Pandas Excel with multiple dataframes*.

It is also possible to position multiple dataframes within the same worksheet:

```python
# Position the dataframes in the worksheet.
df1.to_excel(writer, sheet_name='Sheet1')  # Default position, cell A1.
df2.to_excel(writer, sheet_name='Sheet1', startcol=3)
df3.to_excel(writer, sheet_name='Sheet1', startrow=6)

# Write the dataframe without the header and index.
df4.to_excel(writer, sheet_name='Sheet1',
             startrow=7, startcol=4, header=False, index=False)
```

See the full example at *Example: Pandas Excel dataframe positioning*.

30.10 Passing XlsxWriter constructor options to Pandas

XlsxWriter supports several `Workbook()` constructor options such as `strings_to_urls()`. These can also be applied to the `Workbook` object created by Pandas using the `engine_kwargs` keyword:

```
writer = pd.ExcelWriter('pandas_example.xlsx',
                        engine='xlsxwriter',
                        engine_kwargs={'options': {'strings_to_numbers': True}})
```

Note, versions of Pandas prior to 1.3.0 used this syntax:

```
writer = pd.ExcelWriter('pandas_example.xlsx',
                        engine='xlsxwriter',
                        options={'strings_to_numbers': True})
```

30.11 Saving the Dataframe output to a string

It is also possible to write the Pandas XlsxWriter DataFrame output to a byte array:

```python
import pandas as pd
import io

# Create a Pandas dataframe from the data.
df = pd.DataFrame({'Data': [10, 20, 30, 20, 15, 30, 45]})

output = io.BytesIO()

# Use the BytesIO object as the filehandle.
writer = pd.ExcelWriter(output, engine='xlsxwriter')

# Write the data frame to the BytesIO object.
df.to_excel(writer, sheet_name='Sheet1')

writer.close()
xlsx_data = output.getvalue()

# Do something with the data...
```

Note: This feature requires Pandas >= 0.17.

30.12 Additional Pandas and Excel Information

Here are some additional resources in relation to Pandas, Excel and XlsxWriter.

- The XlsxWriter Pandas examples later in the document: *Pandas with XlsxWriter Examples*.

- The Pandas documentation on the pandas.DataFrame.to_excel() method.

- A more detailed tutorial on Using Pandas and XlsxWriter to create Excel charts.

- The series of articles on the "Practical Business Python" website about Using Pandas and Excel.

WORKING WITH POLARS AND XLSXWRITER

Polars is a fast dataframe library for Rust and Python.

Polars provides very tight integration of XlsxWriter and supports a lot of features such as conditional formats, tables, autofilters, autofit and others without having to use the external library directly. At the same time it allows use of native XlsxWriter workbook and worksheets objects to take advantage of any features that it doesn't support.
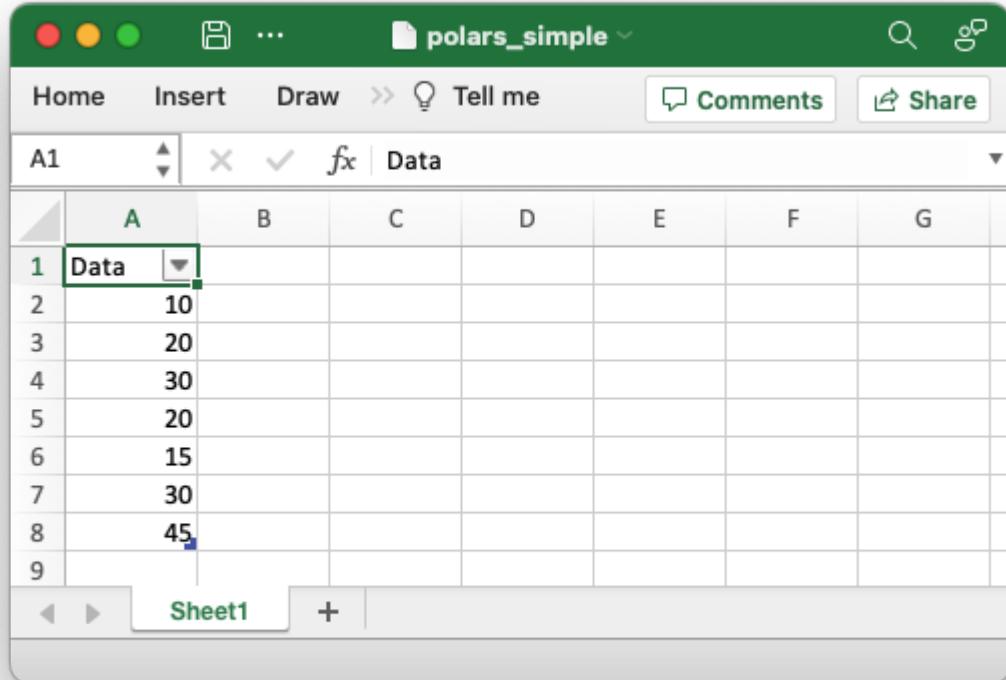
Creating an Excel file from a dataframe with Polars is straightforward and doesn't require any direct invocation of XlsxWriter. All that is required is to use the `write_excel()` method with a Polars dataframe:

```python
import polars as pl

df = pl.DataFrame({"Data": [10, 20, 30, 20, 15, 30, 45]})

df.write_excel(workbook="polars_simple.xlsx")
```

This is a complete example and the output from this would look like the following:

The write_excel() API is explained in detail in the Polars documentation.

One interesting aspect of the Polars output is that it writes the dataframe as an *Excel Data Table*. We will discuss this and other XlsxWriter features that are available from `write_excel()` in the sections below.

31.1 Sharing XlsxWriter workbooks with Polars

In a majority of use cases you will be able to control the output workbook and worksheets via the write_excel() APIs but there may be some situations you may wish to start a normal XlsxWriter workbook and then add Polars data to it.
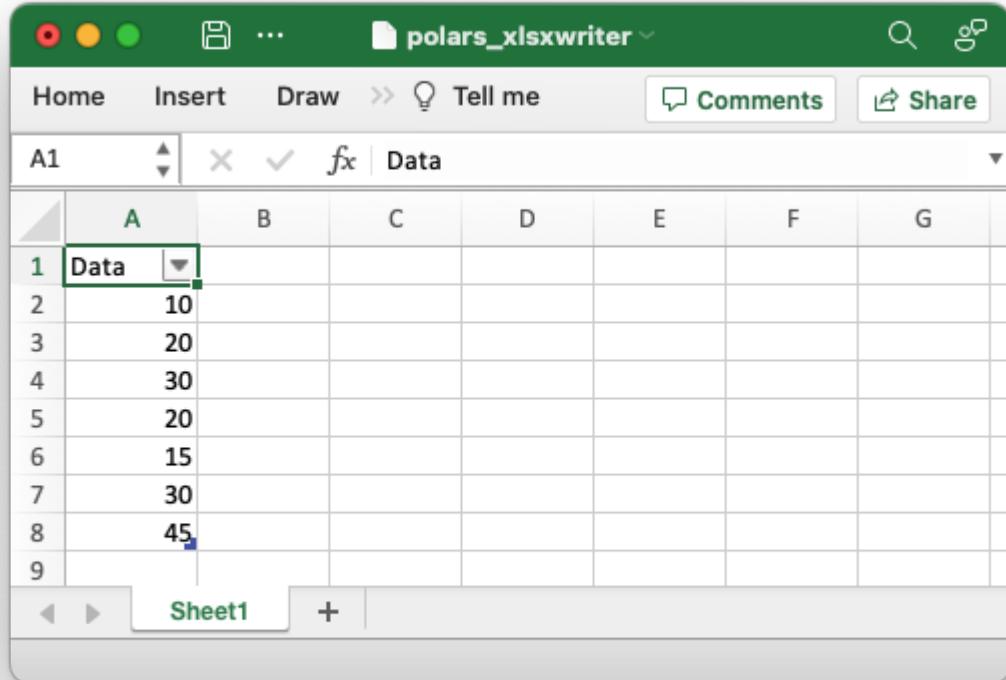
To do this you can create a *workbook* object and pass it to the `workbook` parameter of Polars write_excel():

```python
import xlsxwriter
import polars as pl

df = pl.DataFrame({"Data": [10, 20, 30, 20, 15]})

with xlsxwriter.Workbook("polars_xlsxwriter.xlsx") as workbook:
    df.write_excel(workbook=workbook)
```

Output:

As can be seen from the image, Polars creates a new worksheet and adds the data to it. However, you can also add Polars data to a worksheet created from XlsxWriter:
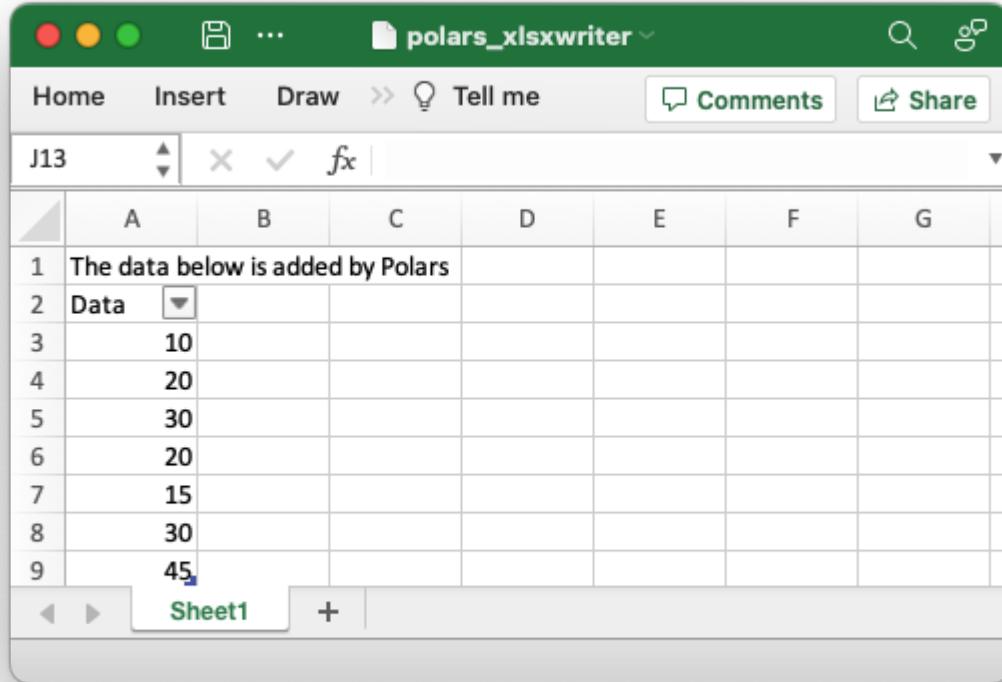
```python
import xlsxwriter
import polars as pl

df = pl.DataFrame({"Data": [10, 20, 30, 20, 15, 30, 45]})

with xlsxwriter.Workbook("polars_xlsxwriter.xlsx") as workbook:
    # Create a new worksheet.
    worksheet = workbook.add_worksheet()

    # Do something with the worksheet.
    worksheet.write("A1", "The data below is added by Polars")

    # Write the Polars data to the worksheet created above, at an offset to
    # avoid overwriting the previous text.
    df.write_excel(workbook=workbook, worksheet="Sheet1", position="A2")
```

Output:

(See the example at *Example: Polars integration with XlsxWriter*.)

31.2 Adding Charts to Dataframe output

With the techniques shown above we can get access to the Workbook and Worksheet objects and
then use them to apply other features such as adding a chart:

```python
import xlsxwriter
import polars as pl

df = pl.DataFrame({"Data": [10, 20, 30, 20, 15, 30, 45]})

with xlsxwriter.Workbook("polars_chart.xlsx") as workbook:
    # Create the worksheet so we can reuse it later.
    worksheet = workbook.add_worksheet()

    # Write the Polars data to the worksheet created above.
    df.write_excel(workbook=workbook, worksheet="Sheet1")

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

    # Get the dimensions of the dataframe.
    (max_row, max_col) = df.shape
```

```
# Configure the series of the chart from the dataframe data.
chart.add_series({"values": ["Sheet1", 1, max_col - 1, max_row, max_col - 1]})

# Insert the chart into the worksheet.
worksheet.insert_chart(1, 3, chart)
```
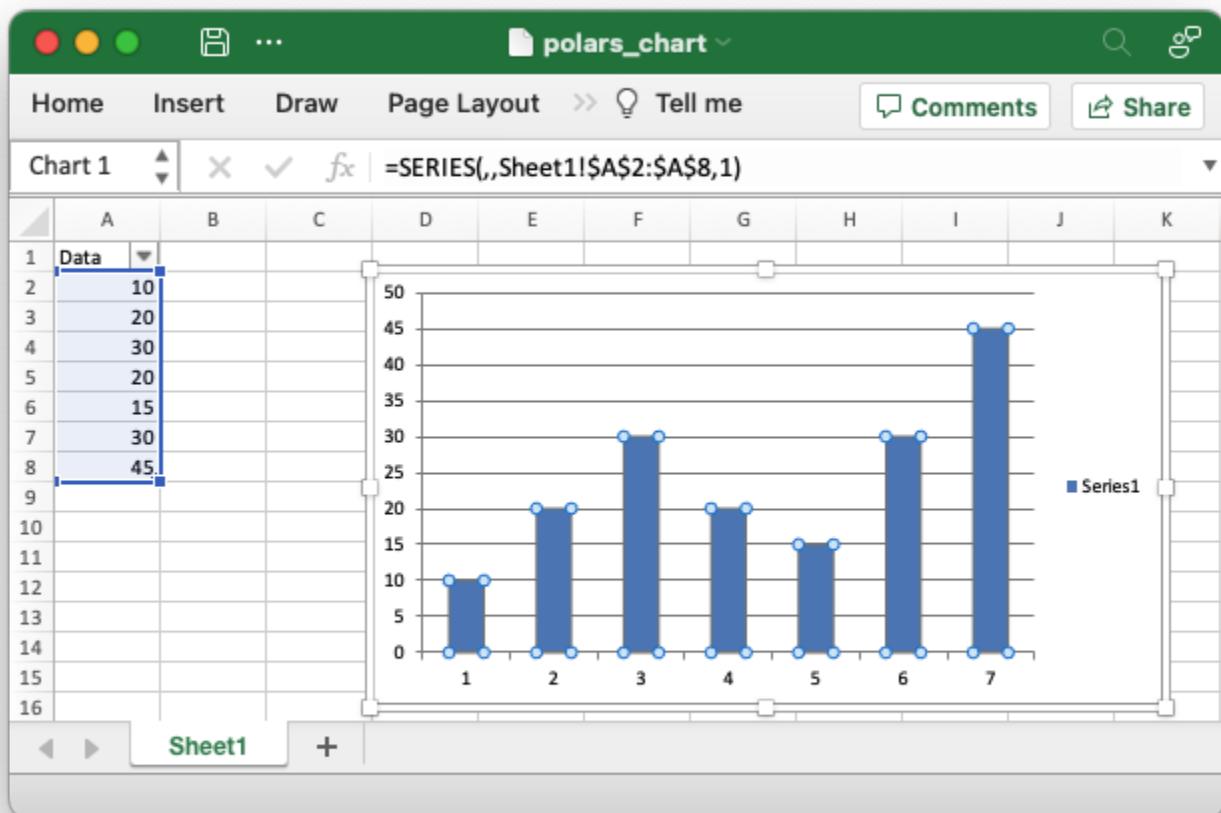
The output would look like this:

(See the example at *Example: Polars Excel output with a chart*.)

31.3 Adding Conditional Formatting to Dataframe output

Following on from the technique shown in the previous sections we could also add a conditional format to the dataframe data like this:

```
import xlsxwriter
import polars as pl
```

```
with xlsxwriter.Workbook("polars_conditional.xlsx") as workbook:
    df = pl.DataFrame({"Data": [10, 20, 30, 20, 15, 30, 45]})

    worksheet = workbook.add_worksheet()

    # Write the Polars data to the worksheet created above.
    df.write_excel(workbook=workbook, worksheet="Sheet1")

    # Get the dimensions of the dataframe.
    (max_row, max_col) = df.shape

    # Apply a conditional format to the required cell range.
    worksheet.conditional_format(1, max_col - 1, max_row, max_col - 1,
                                 {"type": "3_color_scale"})
```
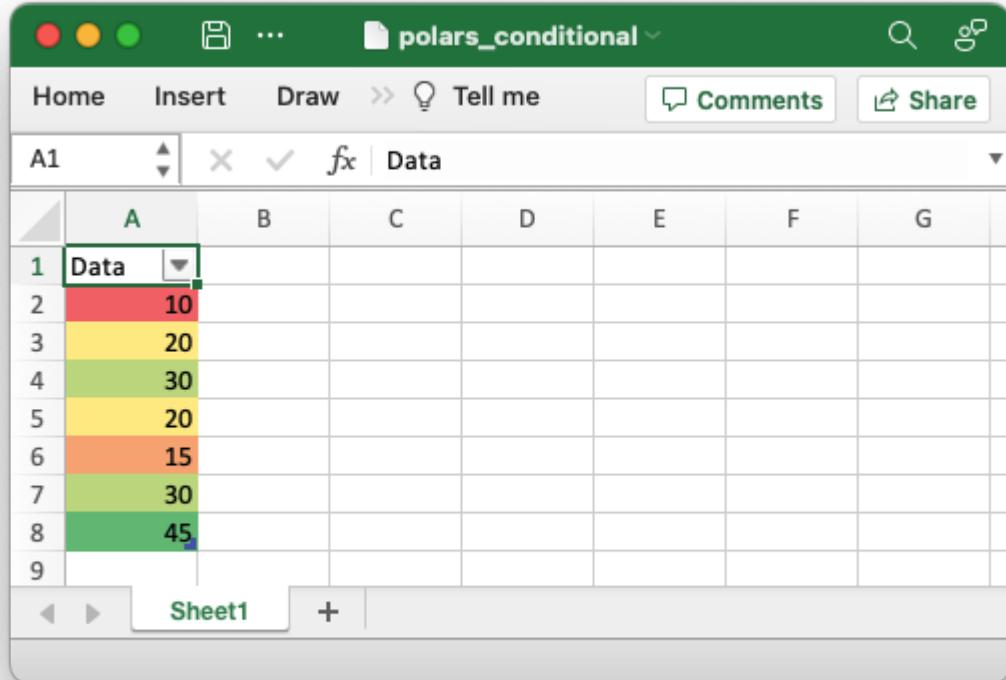
However, this can also be done directly and more succinctly using the Polars `write_excel()` APIs:

```
import polars as pl

df = pl.DataFrame({"Data": [10, 20, 30, 20, 15, 30, 45]})

df.write_excel(
    workbook="pandas_conditional.xlsx",
    conditional_formats={"Data": {"type": "3_color_scale"}},
)
```

Which would give:

See the full example at *Example: Polars Excel output with conditional formatting* and the section of the docs on *Working with Conditional Formatting*.

31.4 Handling multiple Polars Dataframes

It is possible to write more than one dataframe to a worksheet or to several worksheets. For example to write multiple dataframes to multiple worksheets:

```python
with xlsxwriter.Workbook("polars_multiple.xlsx") as workbook:
    df1.write_excel(workbook=workbook)
    df2.write_excel(workbook=workbook)
    df3.write_excel(workbook=workbook)
```

(See the full example at *Example: Polars Excel with multiple dataframes*.)
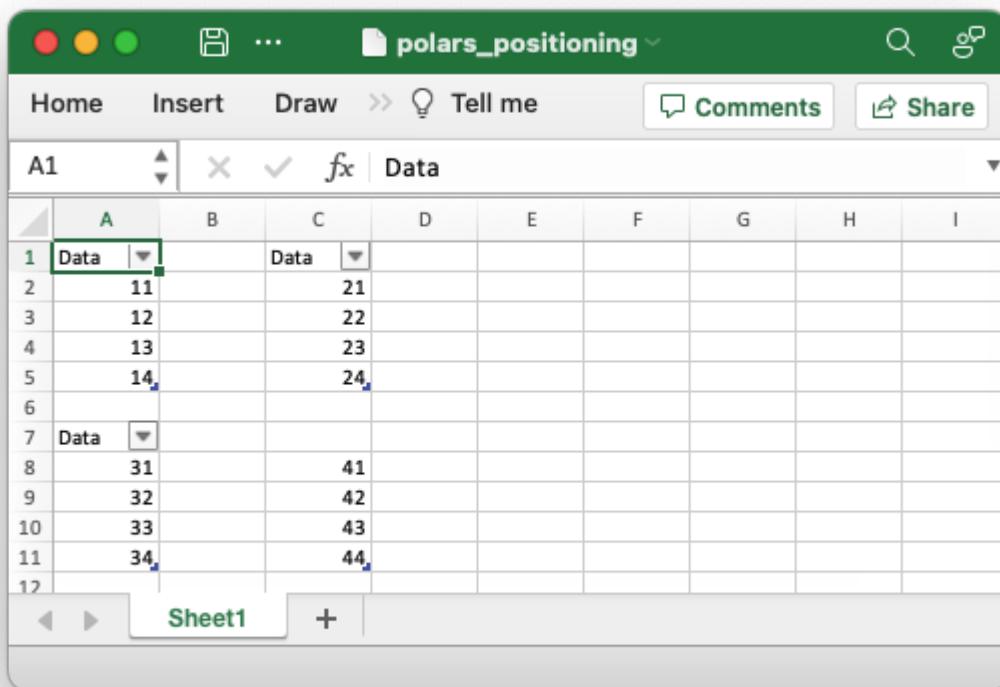
It is also possible to position multiple dataframes within the same worksheet:

```python
with xlsxwriter.Workbook("polars_positioning.xlsx") as workbook:
    # Write the dataframe to the default worksheet and position: Sheet1!A1.
    df1.write_excel(workbook=workbook)

    # Write the dataframe using a cell string position.
    df2.write_excel(workbook=workbook, worksheet="Sheet1", position="C1")
```

```
    # Write the dataframe using a (row, col) tuple position.
    df3.write_excel(workbook=workbook, worksheet="Sheet1", position=(6, 0))

    # Write the dataframe without the header.
    df4.write_excel(
        workbook=workbook,
        worksheet="Sheet1",
        position="C8",
        has_header=False)
```

Output:

(See the full example at *Example: Polars Excel dataframe positioning*.)

31.5 Formatting the dataframe output

Polars uses some sensible default formatting for different data types. For example consider this dataframe comprised of dates, strings and positive and negative numbers:

```
from datetime import date
import polars as pl

df = pl.DataFrame(
```

```
    {
        "Dates": [date(2023, 1, 1), date(2023, 1, 2), date(2023, 1, 3)],
        "Strings": ["Alice", "Bob", "Carol"],
        "Numbers": [0.12345, 100, -99.523],
    }
)

df.write_excel(workbook="polars_format_default.xlsx", autofit=True)
```
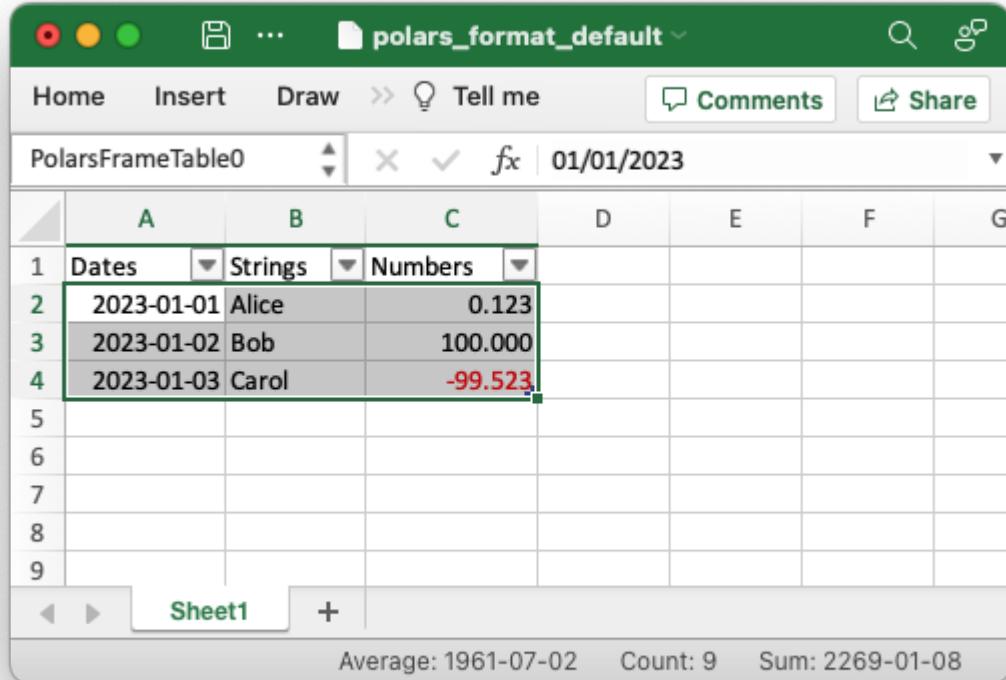

(See the full example at *Example: Polars default format example*).

As can be seen the dates are formatted with a "yyyy-mm-dd" style format and the numbers are formatted to 3 decimal places with negative numbers shown in red (using the number format "#,##0.000;[Red]-#,##0.000"). We also used the autofit parameter in this example to autofit the column widths.

One thing to note from the previous examples is that the Polars dataframes are added to the Excel worksheet as *Excel Data Tables*. This can be seen from the green corner symbol in the bottom right of the dataframe values and from the table view:

Tables are a useful Excel data representation that is analogous to a Python dataframe. We can also use the table properties as well as some of the Polars `write_excel()` options to add some more formatting to the previous example:

```python
from datetime import date
import polars as pl

# Create a Pandas dataframe with some sample data.
df = pl.DataFrame(
    {
        "Dates": [date(2023, 1, 1), date(2023, 1, 2), date(2023, 1, 3)],
        "Strings": ["Alice", "Bob", "Carol"],
        "Numbers": [0.12345, 100, -99.523],
    }
)

# Write the dataframe to a new Excel file with formatting options.
df.write_excel(
    workbook="polars_format_custom.xlsx",

    # Set an alternative table style.
    table_style="Table Style Medium 4",

    # See the floating point precision for reals.
```

```
        float_precision=6,

        # Set an alternative number/date format for Polar Date types.
        dtype_formats={pl.Date: "yyyy mm dd;@"},

        # Add totals to the numeric columns.
        column_totals=True,

        # Autofit the column widths.
        autofit=True,
    )
```


(See the full example at *Example: Polars custom format example*).

31.6 Adding Sparklines to the output dataframe

We can also add *sparklines* to the dataframe output:

```python
import polars as pl
from polars.datatypes import INTEGER_DTYPES

df = pl.DataFrame(
    {
```

```
            "Zone": ["North", "South", "East", "West", "Central"],
            "Q1": [100, 55, -20, 0, 35],
            "Q2": [30, -10, 15, 60, 20],
            "Q3": [-50, 0, 40, 80, 80],
            "Q4": [75, 55, 25, -10, -55],
        }
    )

    # Write the dataframe with sparklines and some additional formatting.
    df.write_excel(
        workbook="polars_sparklines.xlsx",

        # Set an alternative table style.
        table_style="Table Style Light 2",

        # Specify an Excel number format for integer types.
        dtype_formats={INTEGER_DTYPES: "#,##0_);(#,##0)"},

        # Configure sparklines to the dataframe.
        sparklines={
            # We use the default options with just  the source columns.
            "Trend": ["Q1", "Q2", "Q3", "Q4"],

            # We also add a customized sparkline type, with a positioning directive.
            "Change": {
                "columns": ["Q1", "Q2", "Q3", "Q4"],
                "insert_after": "Zone",
                "type": "win_loss",
            },
        },
        column_totals=["Q1", "Q2", "Q3", "Q4"],

        # Hide the default gridlines on the worksheet.
        hide_gridlines=True,
    )
```

Output:

(See the full example at *Example: Polars Excel output with sparklines*).

See also *Working with Sparklines*.
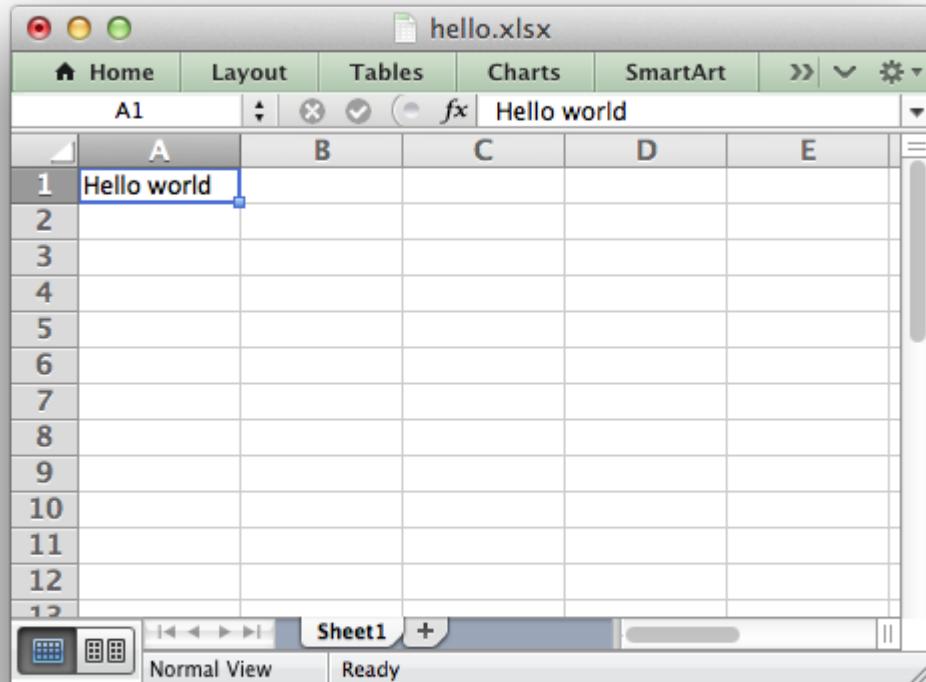
THIRTYTWO

EXAMPLES

The following are some of the examples included in the examples directory of the XlsxWriter distribution.

32.1 Example: Hello World

The simplest possible spreadsheet. This is a good place to start to see if the XlsxWriter module is installed correctly.


```
#####################################################################
#
```
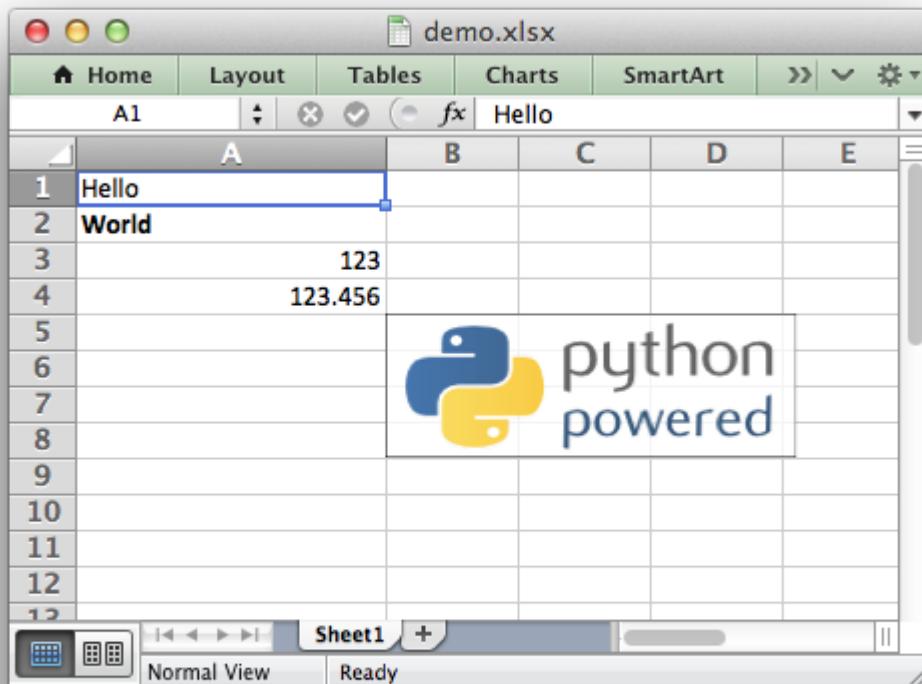
```
# A hello world spreadsheet using the XlsxWriter Python module.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook("hello_world.xlsx")
worksheet = workbook.add_worksheet()

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

workbook.close()
```

32.2 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.
#
# SPDX-License-Identifier: BSD-2-Clause
```

```python
# Copyright 2013-2023, 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*.

32.3 Example: Catch exception on closing

A simple program demonstrating a check for exceptions when closing the file.

We try to `close()` the file in a loop so that if there is an exception, such as if the file is open or locked, we can ask the user to close the file, after which we can try again to overwrite it.

```python
###############################################################################
#
# A simple program demonstrating a check for exceptions when closing the file.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
```

```python
import xlsxwriter

workbook = xlsxwriter.Workbook("check_close.xlsx")
worksheet = workbook.add_worksheet()

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

# Try to close() the file in a loop so that if there is an exception, such as
# if the file is open in Excel, we can ask the user to close the file, and
# try again to overwrite it.
while True:
    try:
        workbook.close()
    except xlsxwriter.exceptions.FileCreateError as e:
        decision = input(
            "Exception caught in workbook.close(): %s\n"
            "Please close the file if it is open in Excel.\n"
            "Try to write file again? [Y/n]: " % e
        )
        if decision != "n":
            continue

    break
```

32.4 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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()
```

32.5 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
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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 a sample alternative link format.
red_format = workbook.add_format(
    {
        "font_color": "red",
        "bold": 1,
        "underline": 1,
        "font_size": 12,
    }
```
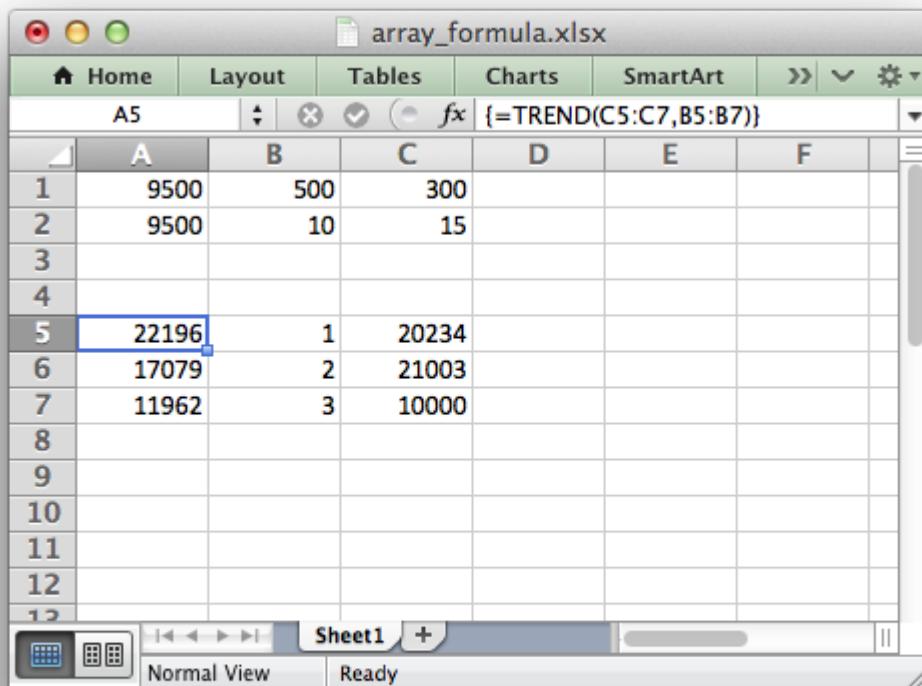
```
    )

    # Write some hyperlinks
    worksheet.write_url("A1", "http://www.python.org/")  # Implicit format.
    worksheet.write_url("A3", "http://www.python.org/", string="Python Home")
    worksheet.write_url("A5", "http://www.python.org/", tip="Click here")
    worksheet.write_url("A7", "http://www.python.org/", red_format)
    worksheet.write_url("A9", "mailto:jmcnamara@cpan.org", string="Mail me")

    # Write a URL that isn't a hyperlink
    worksheet.write_string("A11", "http://www.python.org/")

    workbook.close()
```

32.6 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.
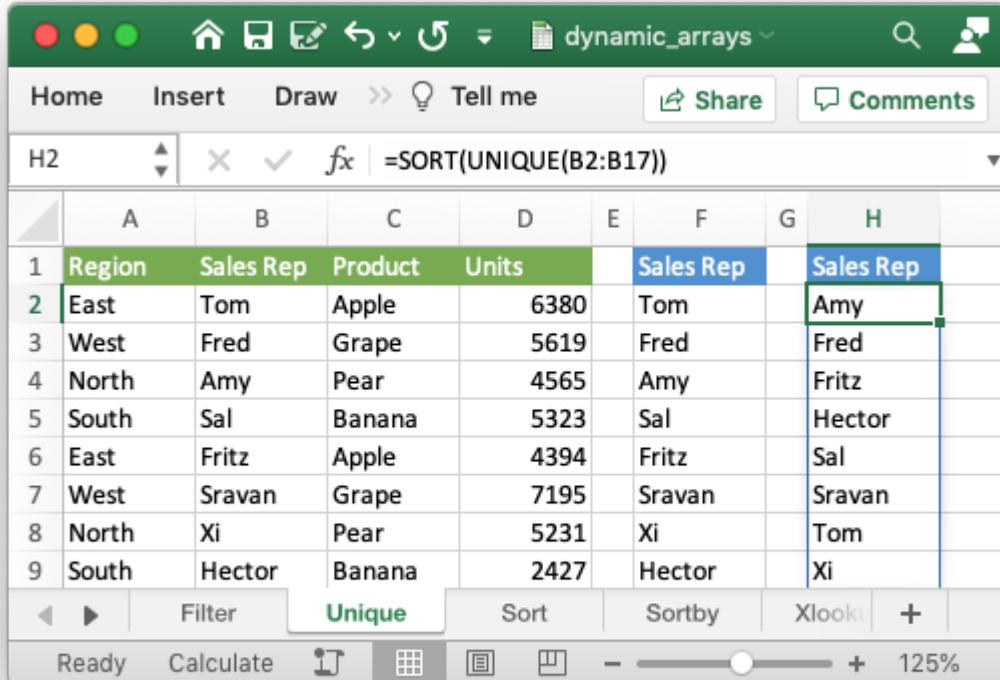

```
    ##########################################################################
    #
    # Example of how to use Python and the XlsxWriter module to write
    # simple array formulas.
```

```python
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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()
```

32.7 Example: Dynamic array formulas

This program is an example of writing formulas that work with dynamic arrays using some of the new functions and functionality introduced in Excel 365. See the write_dynamic_array_formula() method and *Dynamic Array support* for more details.

```
##########################################################################
#
# An example of how to use the XlsxWriter module to write formulas and
# functions that create dynamic arrays. These functions are new to Excel
# 365. The examples mirror the examples in the Excel documentation on these
# functions.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter


def main():
    # Create a new workbook called simple.xls and add some worksheets.
    workbook = xlsxwriter.Workbook("dynamic_arrays.xlsx")

    worksheet1 = workbook.add_worksheet("Filter")
    worksheet2 = workbook.add_worksheet("Unique")
    worksheet3 = workbook.add_worksheet("Sort")
    worksheet4 = workbook.add_worksheet("Sortby")
    worksheet5 = workbook.add_worksheet("Xlookup")
    worksheet6 = workbook.add_worksheet("Xmatch")
    worksheet7 = workbook.add_worksheet("Randarray")
```

```python
worksheet8 = workbook.add_worksheet("Sequence")
worksheet9 = workbook.add_worksheet("Spill ranges")
worksheet10 = workbook.add_worksheet("Older functions")

header1 = workbook.add_format({"fg_color": "#74AC4C", "color": "#FFFFFF"})
header2 = workbook.add_format({"fg_color": "#528FD3", "color": "#FFFFFF"})


#
# Example of using the FILTER() function.
#
worksheet1.write("F2", "=FILTER(A1:D17,C1:C17=K2)")

# Write the data the function will work on.
worksheet1.write("K1", "Product", header2)
worksheet1.write("K2", "Apple")
worksheet1.write("F1", "Region", header2)
worksheet1.write("G1", "Sales Rep", header2)
worksheet1.write("H1", "Product", header2)
worksheet1.write("I1", "Units", header2)

write_worksheet_data(worksheet1, header1)
worksheet1.set_column_pixels("E:E", 20)
worksheet1.set_column_pixels("J:J", 20)


#
# Example of using the UNIQUE() function.
#
worksheet2.write("F2", "=UNIQUE(B2:B17)")

# A more complex example combining SORT and UNIQUE.
worksheet2.write("H2", "=SORT(UNIQUE(B2:B17))")

# Write the data the function will work on.
worksheet2.write("F1", "Sales Rep", header2)
worksheet2.write("H1", "Sales Rep", header2)

write_worksheet_data(worksheet2, header1)
worksheet2.set_column_pixels("E:E", 20)
worksheet2.set_column_pixels("G:G", 20)


#
# Example of using the SORT() function.
#
worksheet3.write("F2", "=SORT(B2:B17)")

# A more complex example combining SORT and FILTER.
worksheet3.write("H2", '=SORT(FILTER(C2:D17,D2:D17>5000,""),2,1)')

# Write the data the function will work on.
worksheet3.write("F1", "Sales Rep", header2)
worksheet3.write("H1", "Product", header2)
worksheet3.write("I1", "Units", header2)
```

```
write_worksheet_data(worksheet3, header1)
worksheet3.set_column_pixels("E:E", 20)
worksheet3.set_column_pixels("G:G", 20)

#
# Example of using the SORTBY() function.
#
worksheet4.write("D2", "=SORTBY(A2:B9,B2:B9)")

# Write the data the function will work on.
worksheet4.write("A1", "Name", header1)
worksheet4.write("B1", "Age", header1)

worksheet4.write("A2", "Tom")
worksheet4.write("A3", "Fred")
worksheet4.write("A4", "Amy")
worksheet4.write("A5", "Sal")
worksheet4.write("A6", "Fritz")
worksheet4.write("A7", "Srivan")
worksheet4.write("A8", "Xi")
worksheet4.write("A9", "Hector")

worksheet4.write("B2", 52)
worksheet4.write("B3", 65)
worksheet4.write("B4", 22)
worksheet4.write("B5", 73)
worksheet4.write("B6", 19)
worksheet4.write("B7", 39)
worksheet4.write("B8", 19)
worksheet4.write("B9", 66)

worksheet4.write("D1", "Name", header2)
worksheet4.write("E1", "Age", header2)

worksheet4.set_column_pixels("C:C", 20)

#
# Example of using the XLOOKUP() function.
#
worksheet5.write("F1", "=XLOOKUP(E1,A2:A9,C2:C9)")

# Write the data the function will work on.
worksheet5.write("A1", "Country", header1)
worksheet5.write("B1", "Abr", header1)
worksheet5.write("C1", "Prefix", header1)

worksheet5.write("A2", "China")
worksheet5.write("A3", "India")
worksheet5.write("A4", "United States")
worksheet5.write("A5", "Indonesia")
worksheet5.write("A6", "Brazil")
worksheet5.write("A7", "Pakistan")
worksheet5.write("A8", "Nigeria")
```

```
worksheet5.write("A9", "Bangladesh")

worksheet5.write("B2", "CN")
worksheet5.write("B3", "IN")
worksheet5.write("B4", "US")
worksheet5.write("B5", "ID")
worksheet5.write("B6", "BR")
worksheet5.write("B7", "PK")
worksheet5.write("B8", "NG")
worksheet5.write("B9", "BD")

worksheet5.write("C2", 86)
worksheet5.write("C3", 91)
worksheet5.write("C4", 1)
worksheet5.write("C5", 62)
worksheet5.write("C6", 55)
worksheet5.write("C7", 92)
worksheet5.write("C8", 234)
worksheet5.write("C9", 880)

worksheet5.write("E1", "Brazil", header2)

worksheet5.set_column_pixels("A:A", 100)
worksheet5.set_column_pixels("D:D", 20)

#
# Example of using the XMATCH() function.
#
worksheet6.write("D2", "=XMATCH(C2,A2:A6)")

# Write the data the function will work on.
worksheet6.write("A1", "Product", header1)

worksheet6.write("A2", "Apple")
worksheet6.write("A3", "Grape")
worksheet6.write("A4", "Pear")
worksheet6.write("A5", "Banana")
worksheet6.write("A6", "Cherry")

worksheet6.write("C1", "Product", header2)
worksheet6.write("D1", "Position", header2)
worksheet6.write("C2", "Grape")

worksheet6.set_column_pixels("B:B", 20)

#
# Example of using the RANDARRAY() function.
#
worksheet7.write("A1", "=RANDARRAY(5,3,1,100, TRUE)")

#
# Example of using the SEQUENCE() function.
#
```

```python
        worksheet8.write("A1", "=SEQUENCE(4,5)")

        #
        # Example of using the Spill range operator.
        #
        worksheet9.write("H2", "=ANCHORARRAY(F2)")

        worksheet9.write("J2", "=COUNTA(ANCHORARRAY(F2))")

        # Write the data the to work on.
        worksheet9.write("F2", "=UNIQUE(B2:B17)")
        worksheet9.write("F1", "Unique", header2)
        worksheet9.write("H1", "Spill", header2)
        worksheet9.write("J1", "Spill", header2)

        write_worksheet_data(worksheet9, header1)
        worksheet9.set_column_pixels("E:E", 20)
        worksheet9.set_column_pixels("G:G", 20)
        worksheet9.set_column_pixels("I:I", 20)

        #
        # Example of using dynamic ranges with older Excel functions.
        #
        worksheet10.write_dynamic_array_formula("B1:B3", "=LEN(A1:A3)")

        # Write the data the to work on.
        worksheet10.write("A1", "Foo")
        worksheet10.write("A2", "Food")
        worksheet10.write("A3", "Frood")

        # Close the workbook.
        workbook.close()


# Utility function to write the data some of the functions work on.
def write_worksheet_data(worksheet, header):
    worksheet.write("A1", "Region", header)
    worksheet.write("B1", "Sales Rep", header)
    worksheet.write("C1", "Product", header)
    worksheet.write("D1", "Units", header)

    data = (
        ["East", "Tom", "Apple", 6380],
        ["West", "Fred", "Grape", 5619],
        ["North", "Amy", "Pear", 4565],
        ["South", "Sal", "Banana", 5323],
        ["East", "Fritz", "Apple", 4394],
        ["West", "Sravan", "Grape", 7195],
        ["North", "Xi", "Pear", 5231],
        ["South", "Hector", "Banana", 2427],
        ["East", "Tom", "Banana", 4213],
        ["West", "Fred", "Pear", 3239],
        ["North", "Amy", "Grape", 6520],
```

```
        ["South", "Sal", "Apple", 1310],
        ["East", "Fritz", "Banana", 6274],
        ["West", "Sravan", "Pear", 4894],
        ["North", "Xi", "Grape", 7580],
        ["South", "Hector", "Apple", 9814],
    )

    row_num = 1
    for row_data in data:
        worksheet.write_row(row_num, 0, row_data)
        row_num += 1


if __name__ == "__main__":
    main()
```
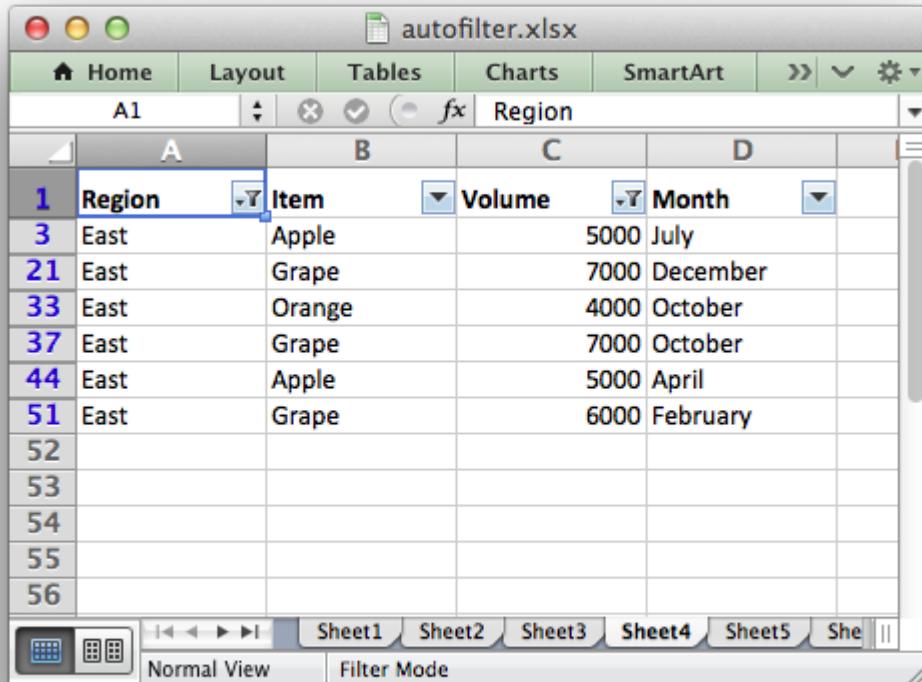
32.8 Example: Applying Autofilters

This program is an example of using autofilters in a worksheet. See *Working with Autofilters* for more details.


```
################################################################################
#
```

```python
# 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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()
worksheet7 = 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.
```

```python
    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
```

```python
###############################################################################
#
#
# 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 a filter list condition in one of the columns.
#

# Set the autofilter.
worksheet5.autofilter("A1:D51")

# Add filter criteria.
worksheet5.filter_column_list("A", ["East", "North", "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 == "North" or region == "South":
        # 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 blanks.
#
# Create a blank cell in our test data.

# Set the autofilter.
worksheet6.autofilter("A1:D51")

# Add filter criteria.
```

```python
    worksheet6.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.
            worksheet6.set_row(row, options={"hidden": True})

        worksheet6.write_row(row, 0, row_data)

        # Move on to the next worksheet row.
        row += 1


    ############################################################################
    #
    #
    # Example 7. Autofilter with filter for non-blanks.
    #

    # Set the autofilter.
    worksheet7.autofilter("A1:D51")

    # Add filter criteria.
    worksheet7.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.
            worksheet7.set_row(row, options={"hidden": True})

        worksheet7.write_row(row, 0, row_data)

        # Move on to the next worksheet row.
        row += 1
```
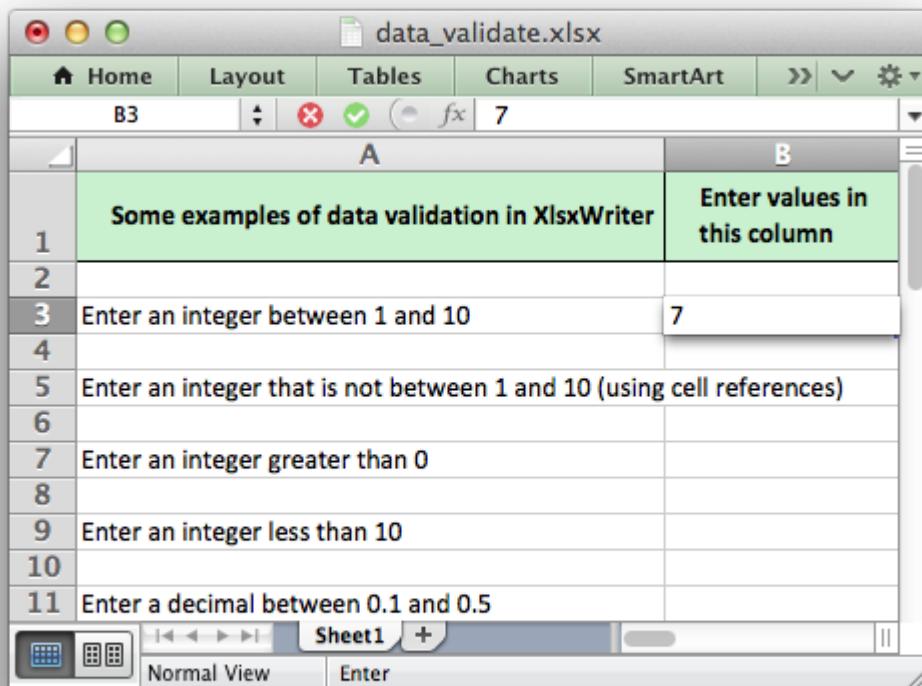
```
workbook.close()
```

32.9 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
from datetime import date, time
```

```python
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)
```

```python
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("B15", {"validate": "list", "source": "=$E$4:$G$4"})


    # Example 8. Limiting input to a date in a fixed range.
    #
    txt = "Enter a date between 1/1/2013 and 12/12/2013"

    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.
    #
```

```python
    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",
```
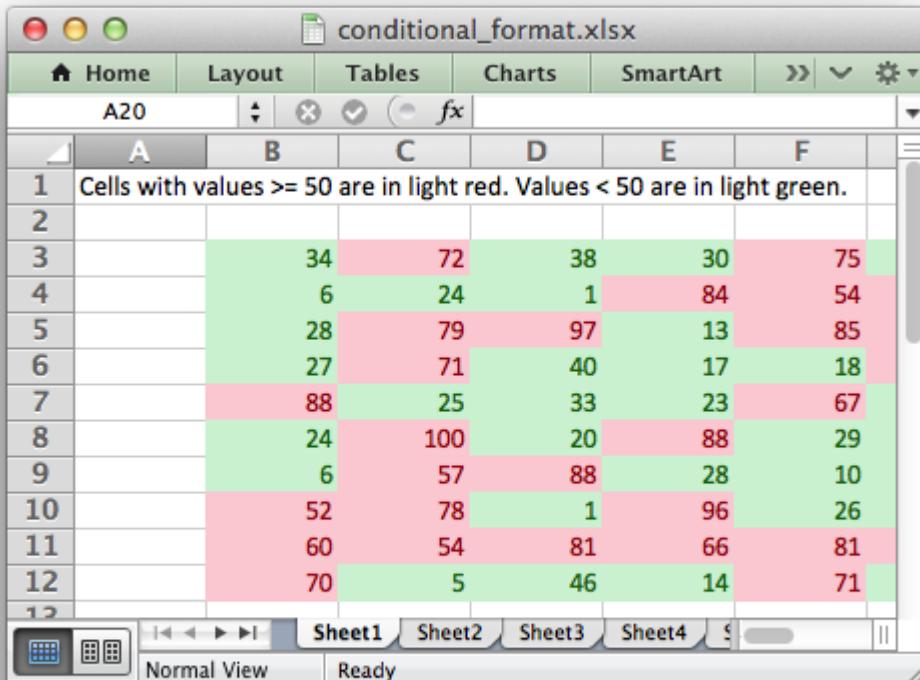
```
        },
    )

    workbook.close()
```

32.10 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook("conditional_format.xlsx")
```

```python
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()

# 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}
)
```

```python
###############################################################################
#
# 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})
```

```python
##############################################################################
#
# 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)
```

```python
    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 with default and user colors."

    data = range(1, 13)

    worksheet7.write("A1", caption)

    worksheet7.write("B2", "2 Color Scale")
    worksheet7.write("D2", "2 Color Scale + user colors")

    worksheet7.write("G2", "3 Color Scale")
    worksheet7.write("I2", "3 Color Scale + user colors")

    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, 6, row_data)
        worksheet7.write(row + 2, 8, row_data)

    worksheet7.conditional_format("B3:B14", {"type": "2_color_scale"})

    worksheet7.conditional_format(
```

```
    "D3:D14", {"type": "2_color_scale", "min_color": "#FF0000", "max_color": "#00FF00"
)

worksheet7.conditional_format("G3:G14", {"type": "3_color_scale"})

worksheet7.conditional_format(
    "I3:I14",
    {
        "type": "3_color_scale",
        "min_color": "#C5D9F1",
        "mid_color": "#8DB4E3",
        "max_color": "#538ED5",
    },
)


###############################################################################
#
# Example 8.
#
caption = "Examples of data bars."

worksheet8.write("A1", caption)

worksheet8.write("B2", "Default data bars")
worksheet8.write("D2", "Bars only")
worksheet8.write("F2", "With user color")
worksheet8.write("H2", "Solid bars")
worksheet8.write("J2", "Right to left")
worksheet8.write("L2", "Excel 2010 style")
worksheet8.write("N2", "Negative same as positive")

data = range(1, 13)
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.write(row + 2, 7, row_data)
    worksheet8.write(row + 2, 9, row_data)

data = [-1, -2, -3, -2, -1, 0, 1, 2, 3, 2, 1, 0]
for row, row_data in enumerate(data):
    worksheet8.write(row + 2, 11, row_data)
    worksheet8.write(row + 2, 13, row_data)

worksheet8.conditional_format("B3:B14", {"type": "data_bar"})

worksheet8.conditional_format("D3:D14", {"type": "data_bar", "bar_only": True})

worksheet8.conditional_format("F3:F14", {"type": "data_bar", "bar_color": "#63C384"})

worksheet8.conditional_format("H3:H14", {"type": "data_bar", "bar_solid": True})
```

```
worksheet8.conditional_format("J3:J14", {"type": "data_bar", "bar_direction": "right"})

worksheet8.conditional_format("L3:L14", {"type": "data_bar", "data_bar_2010": True})

worksheet8.conditional_format(
    "N3:N14",
    {
        "type": "data_bar",
        "bar_negative_color_same": True,
        "bar_negative_border_color_same": True,
    },
)


###############################################################################
#
# Example 9.
#
caption = "Examples of conditional formats with icon sets."

data = [
    [1, 2, 3],
    [1, 2, 3],
    [1, 2, 3],
    [1, 2, 3],
    [1, 2, 3, 4],
    [1, 2, 3, 4, 5],
    [1, 2, 3, 4, 5],
]

worksheet9.write("A1", caption)

for row, row_data in enumerate(data):
    worksheet9.write_row(row + 2, 1, row_data)

worksheet9.conditional_format(
    "B3:D3", {"type": "icon_set", "icon_style": "3_traffic_lights"}
)

worksheet9.conditional_format(
    "B4:D4",
    {"type": "icon_set", "icon_style": "3_traffic_lights", "reverse_icons": True},
)

worksheet9.conditional_format(
    "B5:D5", {"type": "icon_set", "icon_style": "3_traffic_lights", "icons_only": True}
)

worksheet9.conditional_format("B6:D6", {"type": "icon_set", "icon_style": "3_arrows"})

worksheet9.conditional_format("B7:E7", {"type": "icon_set", "icon_style": "4_arrows"})

worksheet9.conditional_format("B8:F8", {"type": "icon_set", "icon_style": "5_arrows"})
```
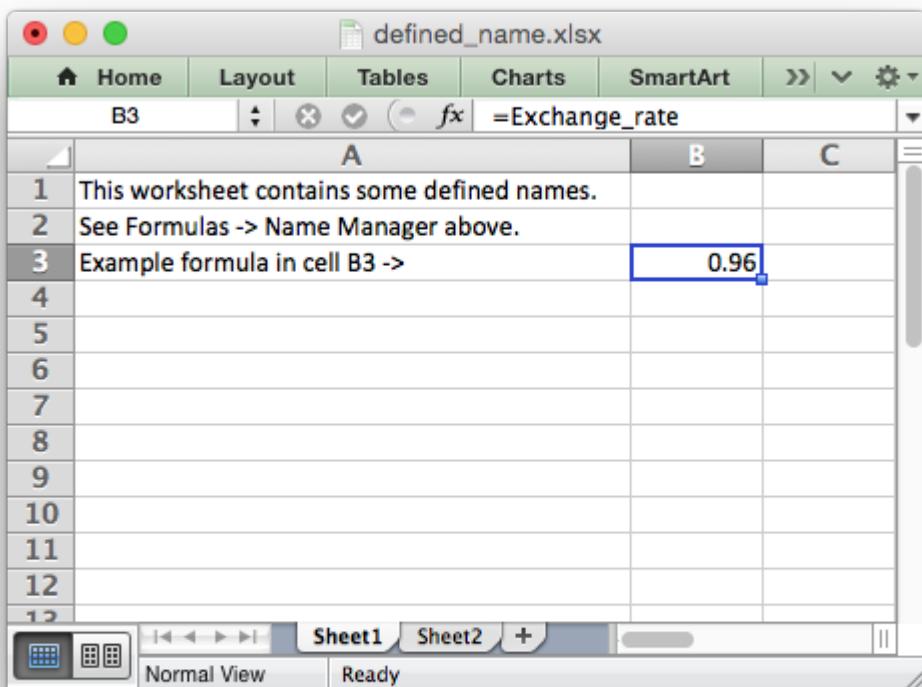
```
worksheet9.conditional_format("B9:F9", {"type": "icon_set", "icon_style": "5_ratings"}

workbook.close()
```

32.11 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
```

```python
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()
```
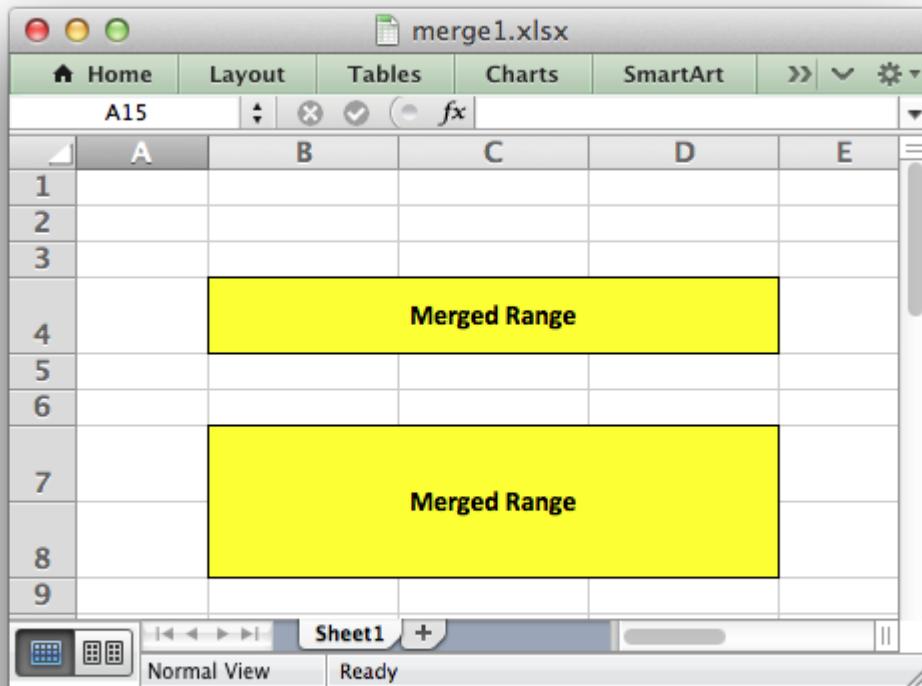
32.12 Example: Merging Cells

This program is an example of merging cells in a worksheet. See the `merge_range()` method for more details.

```python
#######################################################################
#
# A simple example of merging cells with the XlsxWriter Python module.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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()
```
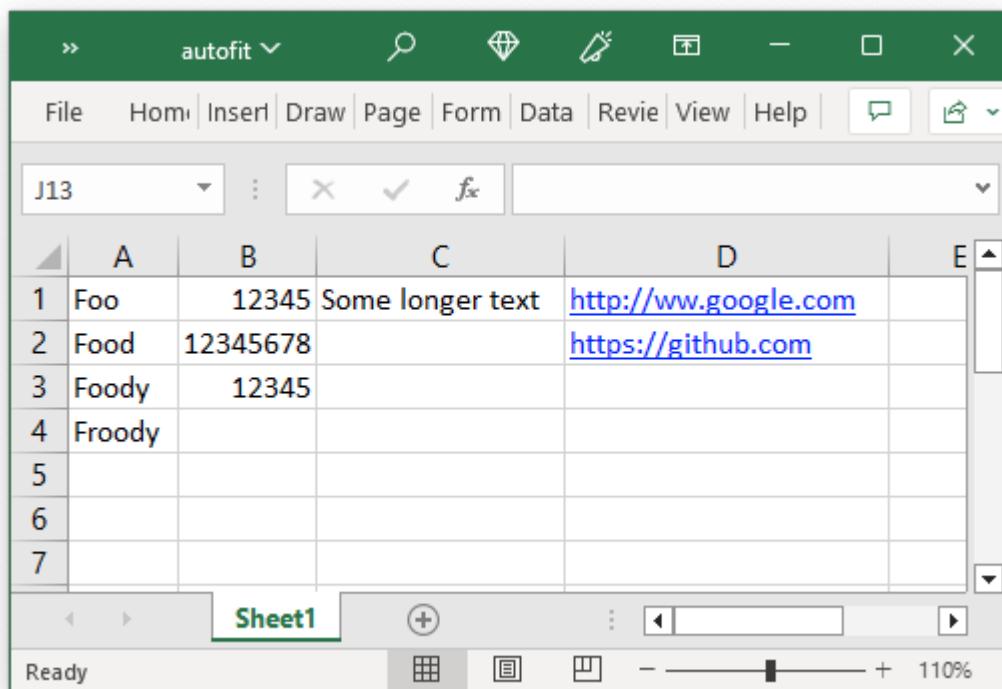
32.13 Example: Autofitting columns

An example of simulating autofitting column widths using the `autofit()` method:


```
###############################################################
#
# An example of using simulated autofit to automatically adjust the width of
# worksheet columns based on the data in the cells.
#
```

```python
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

workbook = Workbook("autofit.xlsx")
worksheet = workbook.add_worksheet()

# Write some worksheet data to demonstrate autofitting.
worksheet.write(0, 0, "Foo")
worksheet.write(1, 0, "Food")
worksheet.write(2, 0, "Foody")
worksheet.write(3, 0, "Froody")

worksheet.write(0, 1, 12345)
worksheet.write(1, 1, 12345678)
worksheet.write(2, 1, 12345)

worksheet.write(0, 2, "Some longer text")

worksheet.write(0, 3, "http://ww.google.com")
worksheet.write(1, 3, "https://github.com")

# Autofit the worksheet.
worksheet.autofit()

workbook.close()
```
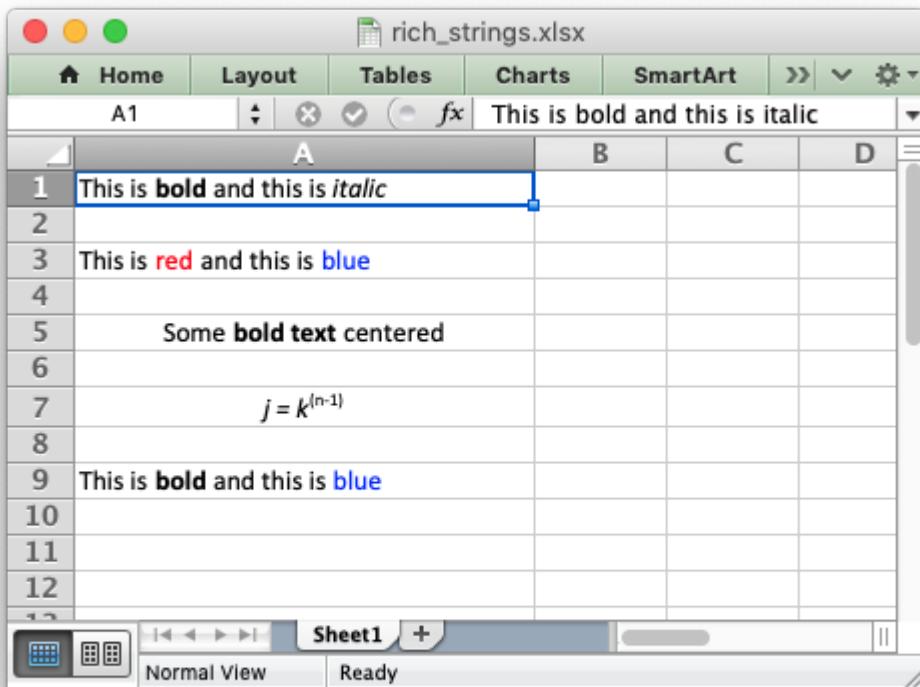
32.14 Example: Writing "Rich" strings with multiple formats

This program is an example of writing rich strings with multiple format to a cell in a worksheet. See the `write_rich_string()` method for more details.

```
########################################################################
#
# An example of using Python and XlsxWriter to write some "rich strings",
# i.e., strings with multiple formats.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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)

# If you have formats and segments in a list you can add them like this:
segments = ["This is ", bold, "bold", " and this is ", blue, "blue"]
worksheet.write_rich_string("A9", *segments)

workbook.close()
```
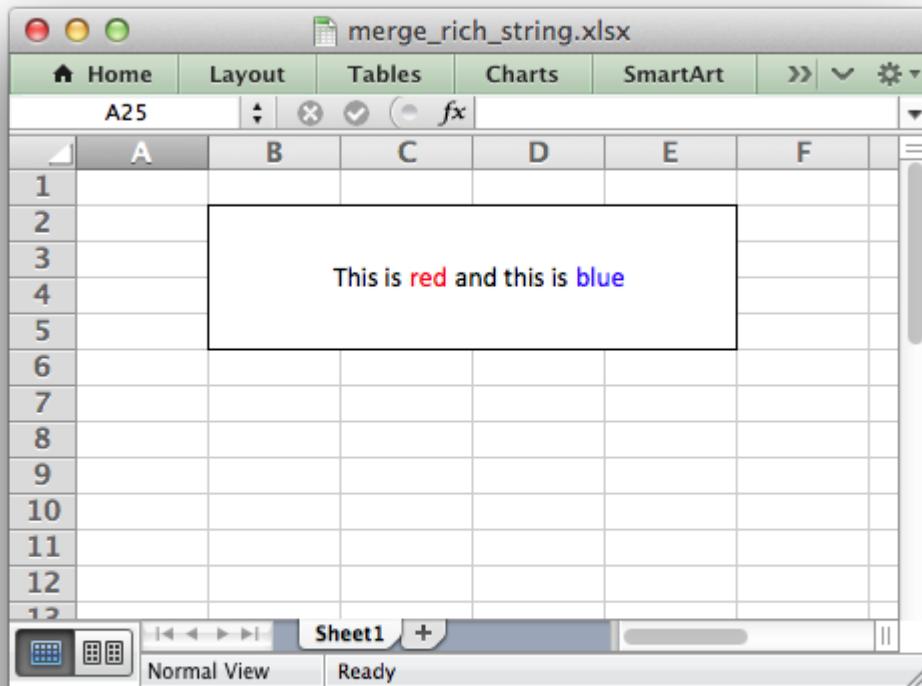
32.15 Example: Merging Cells with a Rich String

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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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
```
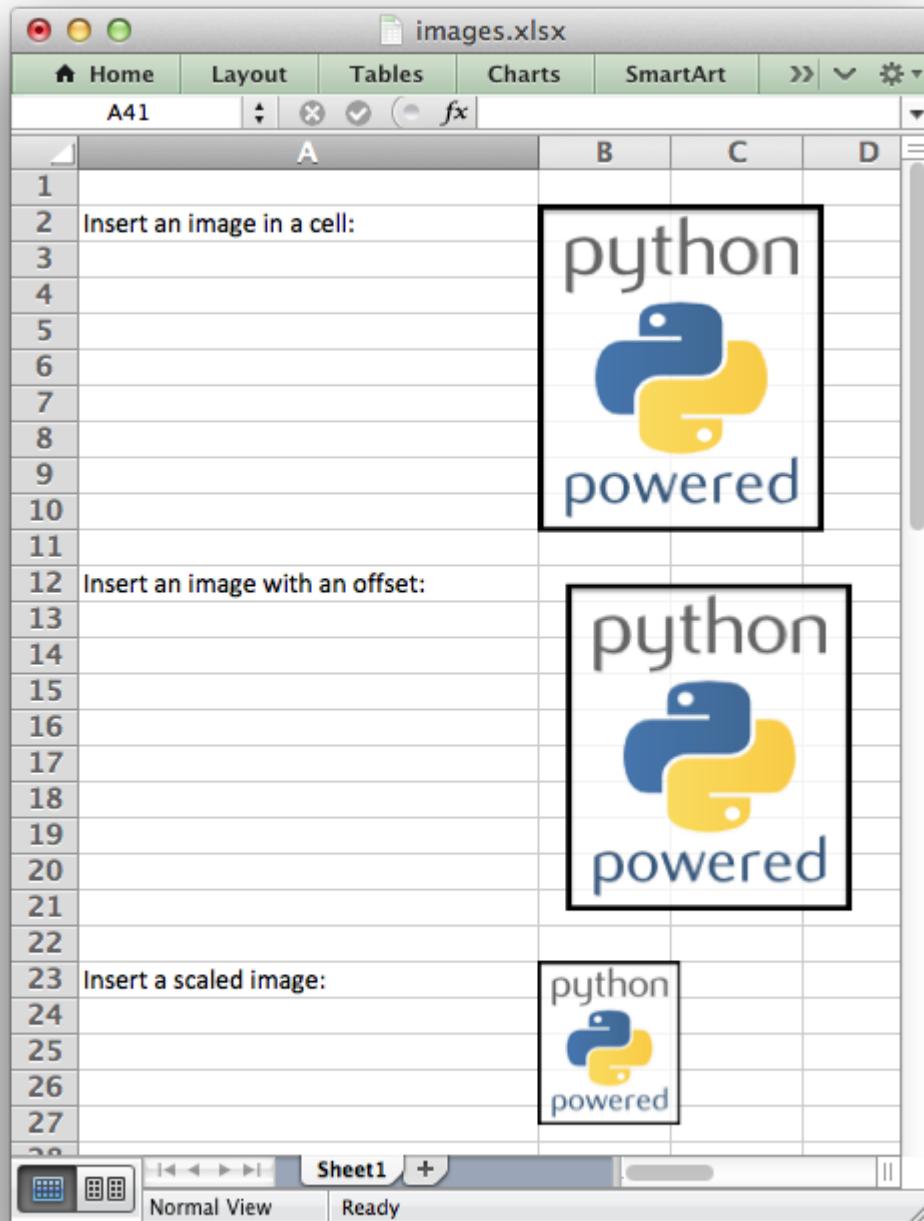
```
    # must also pass the cell format used in the merged cells format at the end.
    worksheet.write_rich_string(
        "B2", "This is ", red, "red", " and this is ", blue, "blue", cell_format
    )

    workbook.close()
```

32.16 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
```
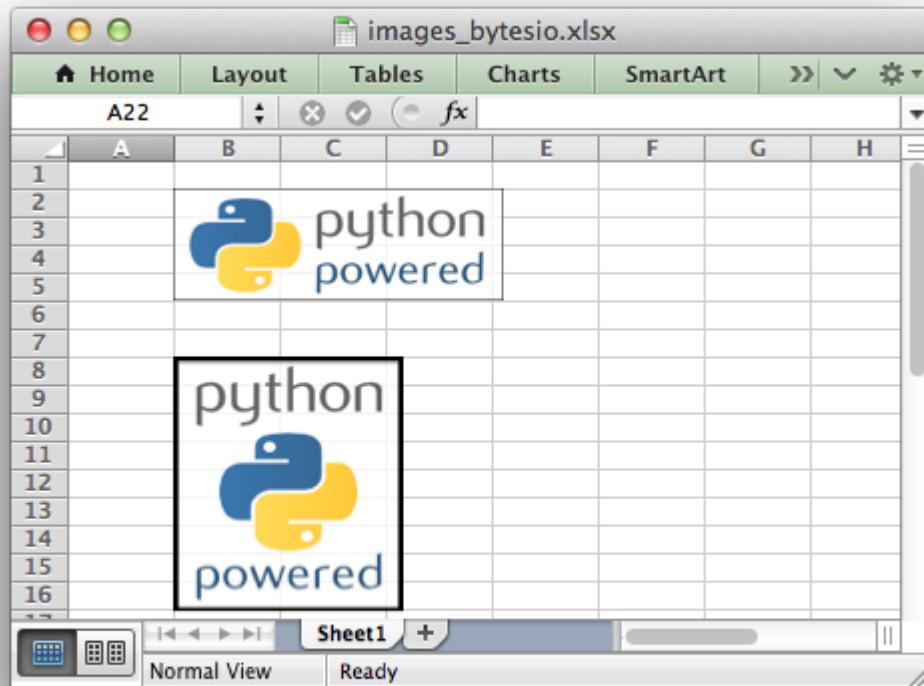
```python
#
import xlsxwriter


# 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()
```

32.17 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

from io import BytesIO
from urllib.request 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"
```
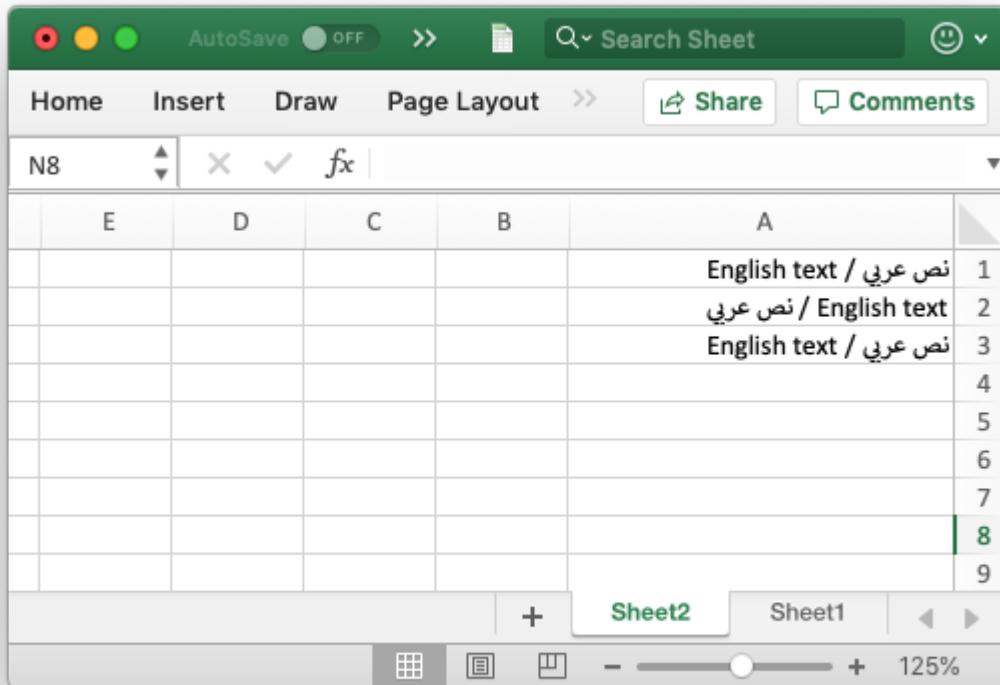
```
    )

    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()
```

32.18 Example: Left to Right worksheets and text

Example of how to use Python and the XlsxWriter module to change the default worksheet and cell text direction from left-to-right to right-to-left as required by some middle eastern versions of Excel.

32.19 Example: Simple Django class

A simple Django View class to write an Excel file using the XlsxWriter module.

```python
##############################################################################
#
# A simple Django view class to write an Excel file using the XlsxWriter
# module.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import io
from django.http import HttpResponse
from django.views.generic import View
import xlsxwriter


def get_simple_table_data():
    # Simulate a more complex table read.
    return [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```python
class MyView(View):
    def get(self, request):
        # Create an in-memory output file for the new workbook.
        output = io.BytesIO()

        # 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' Workbook() constructor option as shown in the docs.
        workbook = xlsxwriter.Workbook(output)
        worksheet = workbook.add_worksheet()

        # Get some data to write to the spreadsheet.
        data = get_simple_table_data()

        # Write some test data.
        for row_num, columns in enumerate(data):
            for col_num, cell_data in enumerate(columns):
                worksheet.write(row_num, col_num, cell_data)

        # Close the workbook before sending the data.
        workbook.close()

        # Rewind the buffer.
        output.seek(0)

        # Set up the Http response.
        filename = "django_simple.xlsx"
        response = HttpResponse(
            output,
            content_type="application/vnd.openxmlformats-officedocument.spreadsheetml.
        )
        response["Content-Disposition"] = "attachment; filename=%s" % filename

        return response
```

32.20 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 SimpleHTTPRequestHandler or Django or with the Google App Engine.

Even though the final file will be in memory, via the BytesIO object, the XlsxWriter module uses temp files during assembly for efficiency. To avoid this on servers that don't allow temp files set the in_memory constructor option to True.

The Python 3 Runtime Environment in Google App Engine supports a filesystem with read/write access to /tmp which means that the in_memory option isn't required there.

```
###########################################################################
#
# Example of using Python and XlsxWriter to create an Excel XLSX file in an in
# memory string suitable for serving via SimpleHTTPRequestHandler or Django or
```

```python
    # with the Google App Engine.
    #
    # SPDX-License-Identifier: BSD-2-Clause
    # Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
    #

    import http.server
    import socketserver
    import io

    import xlsxwriter


    class Handler(http.server.SimpleHTTPRequestHandler):
        def do_GET(self):
            # Create an in-memory output file for the new workbook.
            output = io.BytesIO()

            # 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 set the 'in_memory' constructor option to True.
            #
            # Note: The Python 3 Runtime Environment in Google App Engine supports
            # a filesystem with read/write access to /tmp which means that the
            # 'in_memory' option isn't required there and can be omitted. See:
            #
            # https://cloud.google.com/appengine/docs/standard/python3/runtime#filesystem
            #
            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-officedocument.spreadsheetml.sheet",
            )
            self.end_headers()
            self.wfile.write(output.read())
            return


    print("Server listening on port 8000...")
```
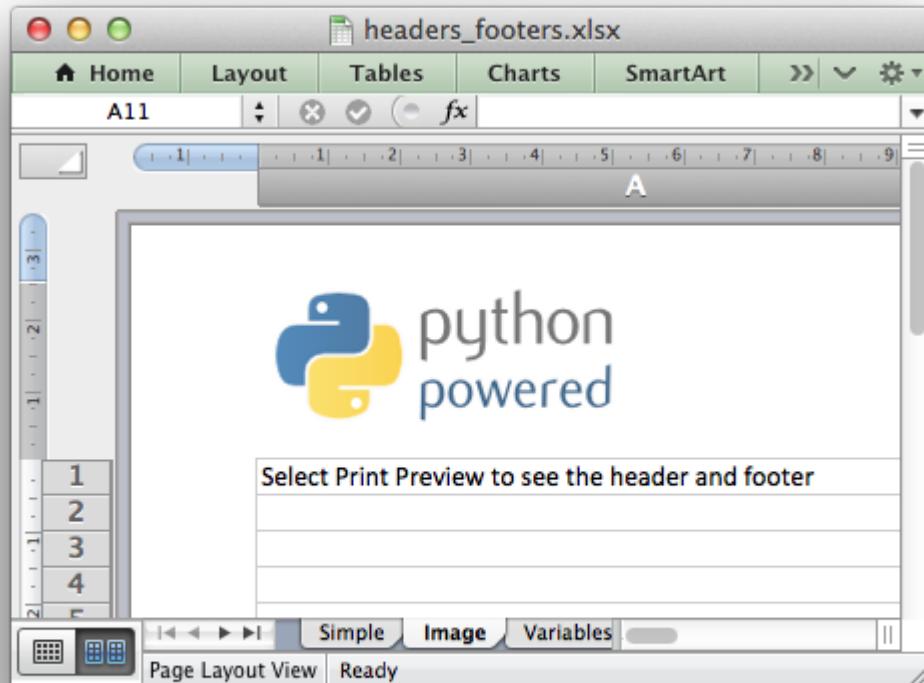
```
httpd = socketserver.TCPServer(("", 8000), Handler)
httpd.serve_forever()
```

32.21 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.
    #
    # SPDX-License-Identifier: BSD-2-Clause
    # Copyright 2013-2023, 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 centered 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()
```

32.22 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter
```

```python
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)
```

```python
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()
```

32.23 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*.


```python
###############################################################################
#
# 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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()
worksheet13 = 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})
```

```python
###############################################################################
#
# 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)
```

32.23. Example: Worksheet Tables

```python
    # 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": [
```

```python
                {"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)
```

```python
    # 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 Excel style removed."

# 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,
    "style": None,
```

```python
        "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(Table12[@[Quarter 1]:[Quarter 4]])",
                "total_function": "sum",
            },
        ],
    }


    # Add a table to the worksheet.
    worksheet12.add_table("B3:G8", options)


    ###########################################################################
    #
    # Example 13.
    #
    caption = "Table with column formats."

    # Set the columns widths.
    worksheet13.set_column("B:G", 12)

    # Write the caption.
    worksheet13.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,
            },
```
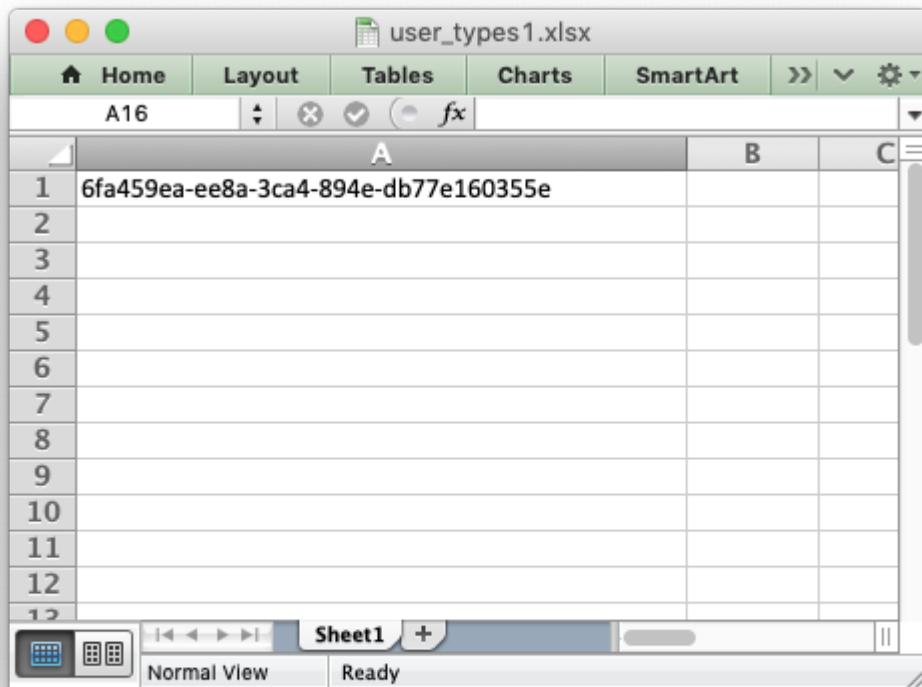
```
        {
            "header": "Quarter 4",
            "total_function": "sum",
            "format": currency_format,
        },
        {
            "header": "Year",
            "formula": "=SUM(Table13[@[Quarter 1]:[Quarter 4]])",
            "total_function": "sum",
            "format": currency_format,
        },
    ],
}

# Add a table to the worksheet.
worksheet13.add_table("B3:G8", options)

workbook.close()
```

32.24 Example: Writing User Defined Types (1)

An example of adding support for user defined types to the XlsxWriter `write()` method using the `add_write_handler()` method.

This example takes UUID data and writes it as a string by adding a callback handler to the `write()` method. A UUID data type would normally raise a `TypeError` in XlsxWriter since it isn't a type that is supported by Excel.

See the *Writing user defined types* section for more details on how this functionality works.

```
########################################################################
#
# An example of adding support for user defined types to the XlsxWriter write()
# method.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter
import uuid


# Create a function that will behave like a worksheet write() method.
#
# This function takes a UUID and writes it as as string. It should take the
# parameters shown below and return the return value from the called worksheet
# write_*() method. In this case it changes the UUID to a string and calls
# write_string() to write it.
#
def write_uuid(worksheet, row, col, token, format=None):
    return worksheet.write_string(row, col, str(token), format)


# Set up the workbook as usual.
workbook = xlsxwriter.Workbook("user_types1.xlsx")
```

```
worksheet = workbook.add_worksheet()

# Make the first column wider for clarity.
worksheet.set_column("A:A", 40)

# Add the write() handler/callback to the worksheet.
worksheet.add_write_handler(uuid.UUID, write_uuid)

# Create a UUID.
my_uuid = uuid.uuid3(uuid.NAMESPACE_DNS, "python.org")

# Write the UUID. This would raise a TypeError without the handler.
worksheet.write("A1", my_uuid)

workbook.close()
```
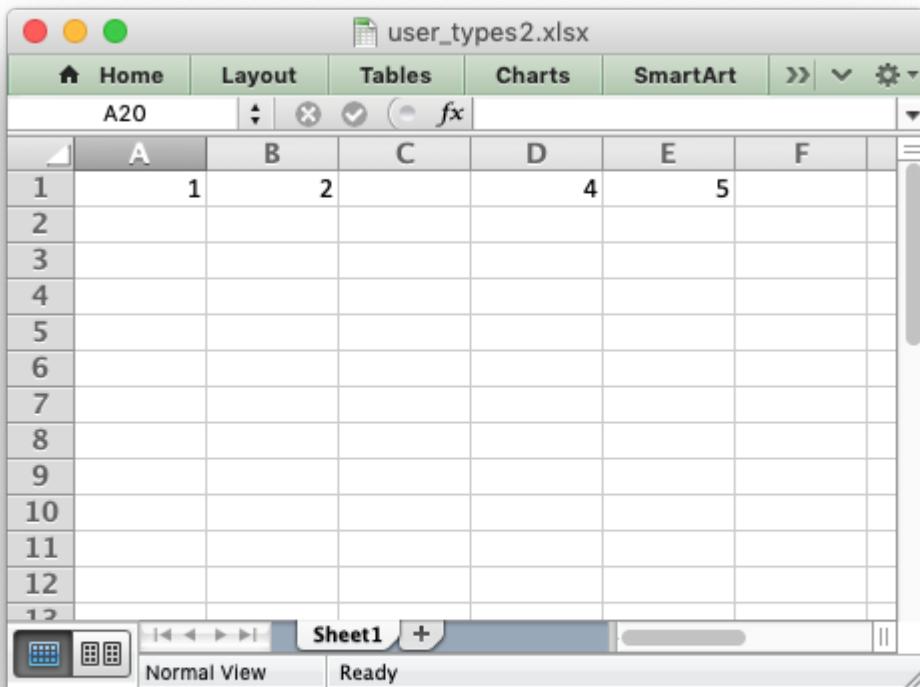
32.25 Example: Writing User Defined Types (2)

An example of adding support for user defined types to the XlsxWriter `write()` method using the `add_write_handler()` method.

This example removes NaN (Not a Number) values from numeric data and writes a blank cell instead. Note, another way to handle this is with the `nan_inf_to_errors` option in the `Workbook()` constructor.

See the *Writing user defined types* section for more details on how this functionality works.

```
#############################################################################
#
# An example of adding support for user defined types to the XlsxWriter write()
# method.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter
import math


# Create a function that will behave like a worksheet write() method.
#
# This function takes a float and if it is NaN then it writes a blank cell
# instead. It should take the parameters shown below and return the return
# value from the called worksheet write_*() method.
#
def ignore_nan(worksheet, row, col, number, format=None):
    if math.isnan(number):
        return worksheet.write_blank(row, col, None, format)
    else:
        # Return control to the calling write() method for any other number.
        return None
```

```
# Set up the workbook as usual.
workbook = xlsxwriter.Workbook("user_types2.xlsx")
worksheet = workbook.add_worksheet()


# Add the write() handler/callback to the worksheet.
worksheet.add_write_handler(float, ignore_nan)

# Create some data to write.
my_data = [1, 2, float("nan"), 4, 5]

# Write the data. Note that write_row() calls write() so this will work as
# expected. Writing NaN values would raise a TypeError without the handler.
worksheet.write_row("A1", my_data)

workbook.close()
```

32.26 Example: Writing User Defined types (3)

An example of adding support for user defined types to the XlsxWriter `write()` method using the `add_write_handler()` method.

This, somewhat artificial, example shows how to use the `row` and `col` parameters to control the logic of the callback function. It changes the worksheet `write()` method so that it hides/replaces user passwords when writing string values based on their position in the worksheet.

See the *Writing user defined types* section for more details on how this functionality works.
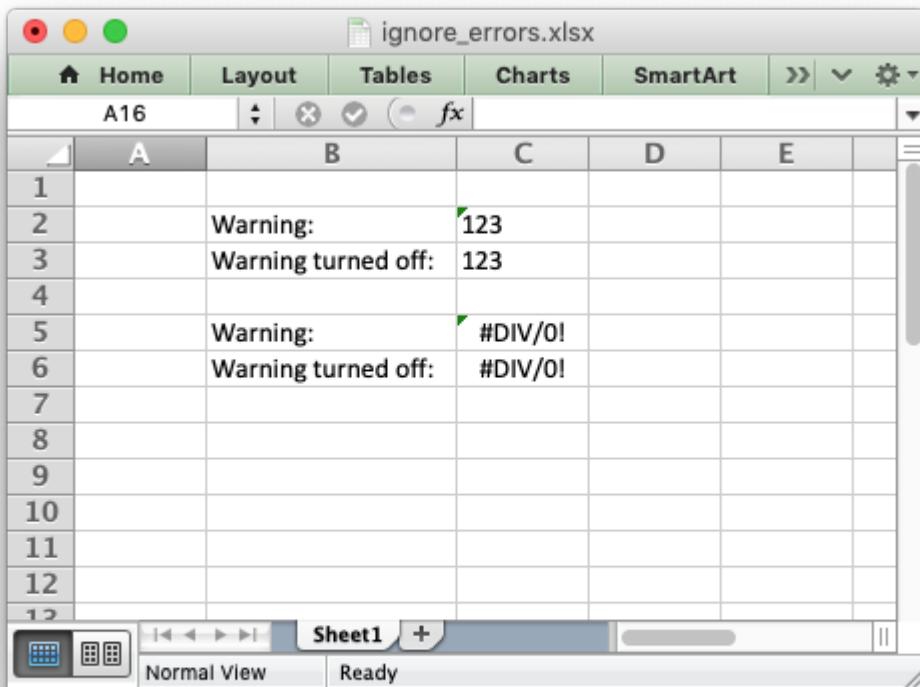
```
###############################################################################
#
# An example of adding support for user defined types to the XlsxWriter write()
# method.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter


# Create a function that changes the worksheet write() method so that it
# hides/replaces user passwords when writing string data. The password data,
# based on the sample data structure, will be data in the second column, apart
# from the header row.
def hide_password(worksheet, row, col, string, format=None):
    if col == 1 and row > 0:
        return worksheet.write_string(row, col, "****", format)
    else:
        return worksheet.write_string(row, col, string, format)


# Set up the workbook as usual.
workbook = xlsxwriter.Workbook("user_types3.xlsx")
worksheet = workbook.add_worksheet()
```

```python
    # Make the headings in the first row bold.
    bold = workbook.add_format({"bold": True})
    worksheet.set_row(0, None, bold)

    # Add the write() handler/callback to the worksheet.
    worksheet.add_write_handler(str, hide_password)

    # Create some data to write.
    my_data = [
        ["Name", "Password", "City"],
        ["Sara", "$5%^6&", "Rome"],
        ["Michele", "123abc", "Milano"],
        ["Maria", "juvexme", "Torino"],
        ["Paolo", "qwerty", "Fano"],
    ]

    # Write the data. Note that write_row() calls write() so this will work as
    # expected.
    for row_num, row_data in enumerate(my_data):
        worksheet.write_row(row_num, 0, row_data)

    workbook.close()
```

32.27 Example: Ignoring Worksheet errors and warnings

An example of ignoring Excel worksheet errors/warnings using the worksheet `ignore_errors()` method.

```
############################################################################
#
# An example of turning off worksheet cells errors/warnings using the
# XlsxWriter Python module.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook("ignore_errors.xlsx")
worksheet = workbook.add_worksheet()

# Write strings that looks like numbers. This will cause an Excel warning.
worksheet.write_string("C2", "123")
worksheet.write_string("C3", "123")

# Write a divide by zero formula. This will also cause an Excel warning.
worksheet.write_formula("C5", "=1/0")
worksheet.write_formula("C6", "=1/0")

# Turn off some of the warnings:
worksheet.ignore_errors({"number_stored_as_text": "C3", "eval_error": "C6"})

# Write some descriptions for the cells and make the column wider for clarity.
```

```
worksheet.set_column("B:B", 16, None)
worksheet.write("B2", "Warning:")
worksheet.write("B3", "Warning turned off:")
worksheet.write("B5", "Warning:")
worksheet.write("B6", "Warning turned off:")

workbook.close()
```
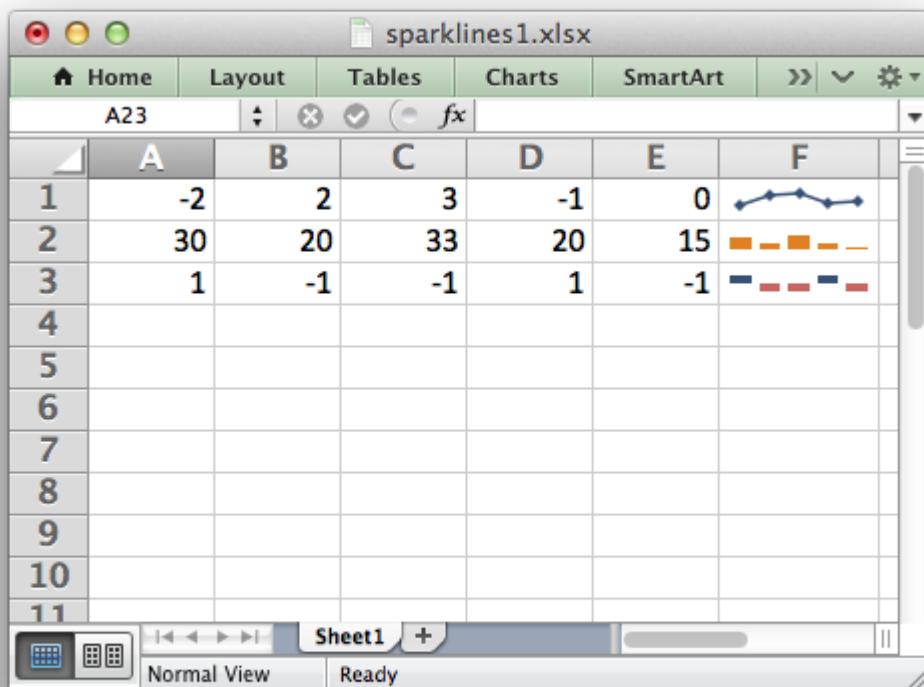
32.28 Example: Sparklines (Simple)

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.
#
```

```python
# See sparklines2.py for examples of more complex sparkline formatting.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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()
```

32.29 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.

	A	B
1	**Sparkline**	**Description**
2	⟍‿⟋	A default "line" sparkline.
3	■▄■▄▄▂▂▄▄	A default "column" sparkline.
4	▀▀▄▄▀▄▀▀▀▄	A default "win/loss" sparkline.
5		
6	⟍‿⟋	Line with markers.
7	⟍‿⟋	Line with high and low points.
8	⟍‿⟋	Line with first and last point markers.
9	⟍‿⟋	Line with negative point markers.
10	⟍‿⟋	Line with axis.
11		
12	■▄■▄▄▂▂▄▄	Column with default style (1).
13	■▄■▄▄▂▂▄▄	Column with style 2.
14	■▄■▄▄▂▂▄▄	Column with style 3.
15	■▄■▄▄▂▂▄▄	Column with style 4.
16	■▄■▄▄▂▂▄▄	Column with style 5.
17	■▄■▄▄▂▂▄▄	Column with style 6.
18	■▄■▄▄▂▂▄▄	Column with a user defined colour.
19		
20	▀▀▄▄▀▄▀▀▀▄	A win/loss sparkline.
21	▀▀▄▄▀▄▀▀▀▄	A win/loss sparkline with negative point
22		
23	▂▂▂▂▂▄▄■	A left to right column (the default).
24	■▄▄▂▂▂▂▂▂	A right to left column.
25	Growth▂▂▄▄■	Sparkline and text in one cell.
26		
27	⟍‿⟋	A grouped sparkline. Changes are applie
28	⟍‿⟋	
29	⟍‿⟋	

Sheet1 | Sheet2 | +

Normal View   Ready

```python
################################################################################
#
# 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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.'
```

```python
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 color."

    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}
```

```python
)

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)
```
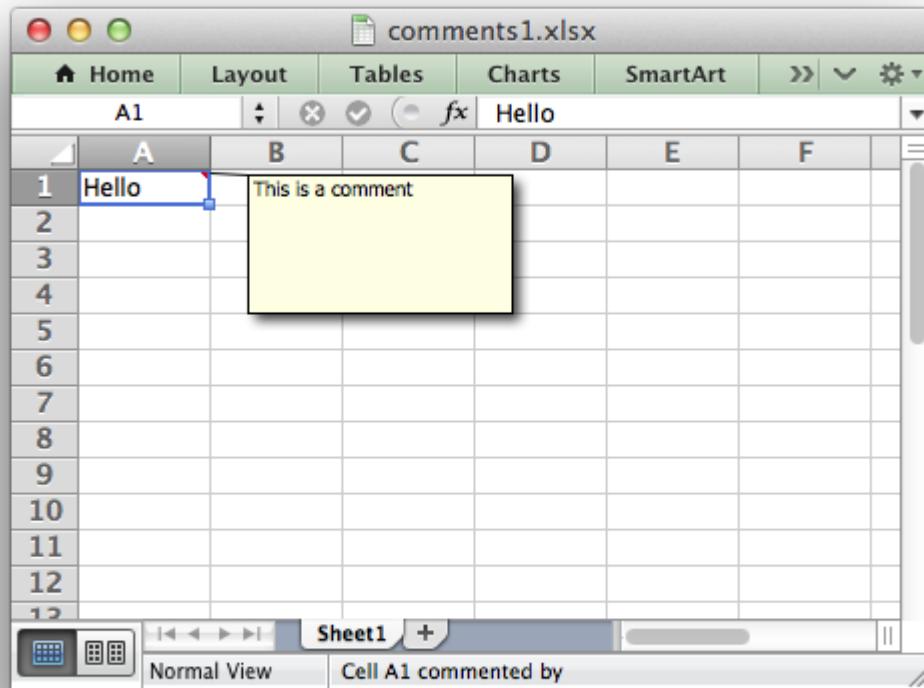
```
data = [
    # Simple line data.
    [-2, 2, 3, -1, 0, -2, 3, 2, 1, 0],
    # Simple column data.
    [30, 20, 33, 20, 15, 5, 5, 15, 10, 15],
    # Simple win/loss data.
    [1, 1, -1, -1, 1, -1, 1, 1, 1, -1],
    # Unbalanced histogram.
    [5, 6, 7, 10, 15, 20, 30, 50, 70, 100],
    # Data for 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()
```

32.30 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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()
```
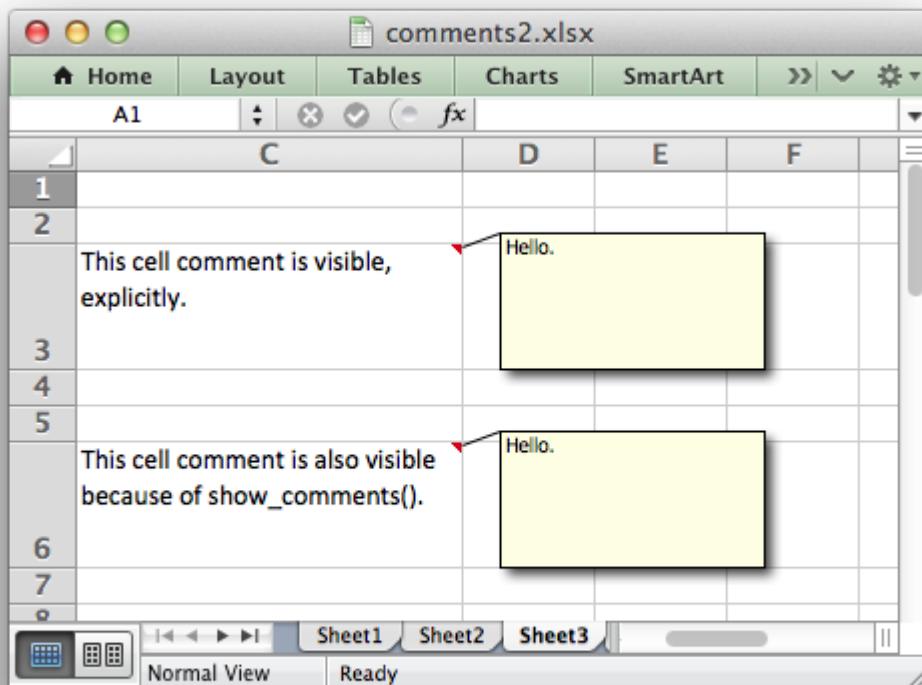
32.31 Example: Adding Cell Comments to Worksheets (Advanced)

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*.


```python
###############################################################################
#
# An example of writing cell comments to a worksheet using Python and
# XlsxWriter.
#
# Each of the worksheets demonstrates different features of cell comments.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook("comments.xlsx")

worksheet1 = workbook.add_worksheet()
worksheet2 = workbook.add_worksheet()
worksheet3 = workbook.add_worksheet()
worksheet4 = workbook.add_worksheet()
worksheet5 = workbook.add_worksheet()
```

```python
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.
#

# Set up some formatting.
worksheet1.set_column("C:C", 25)
worksheet1.set_row(2, 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.
#
```

```python
    # 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": True})

    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.set_row(18, 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."
```

```python
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": 0.5})

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": 50})


###############################################################################
#
# 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 color.
#
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 color."
    comment = "Hello."

    worksheet6.write("C3", cell_text, text_wrap)
    worksheet6.write_comment("C3", comment, {"color": "green"})

    cell_text = "This cell comment has the default color."

    worksheet6.write("C6", cell_text, text_wrap)
    worksheet6.write_comment("C6", comment)

    cell_text = "This cell comment has a different color."

    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)

    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()
```
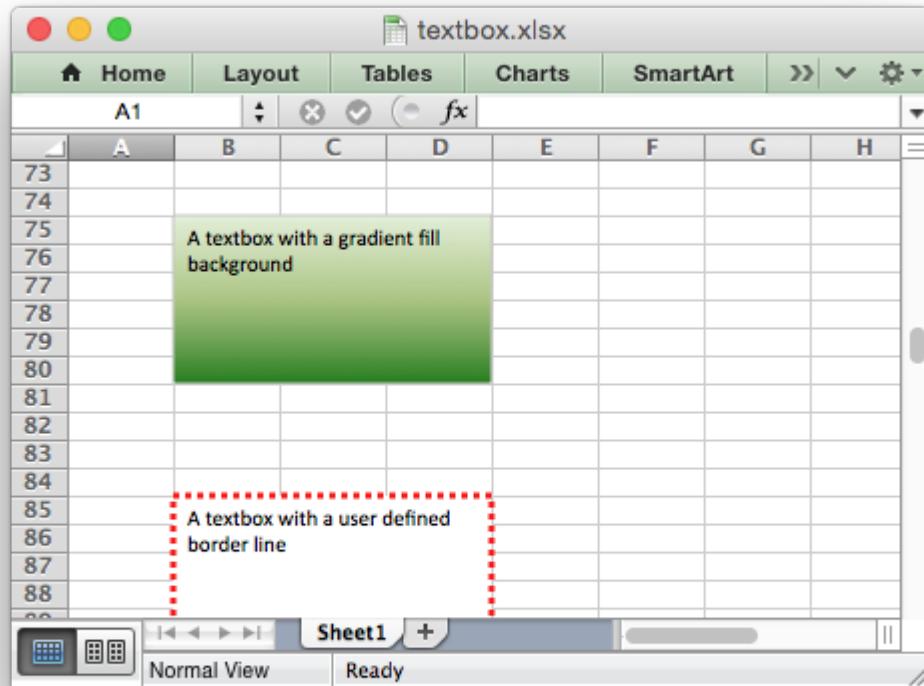
32.32 Example: Insert Textboxes into a Worksheet

The following is an example of how to insert and format textboxes in a worksheet, see `insert_textbox()` and *Working with Textboxes* for more details.

```python
#######################################################################
#
# An example of inserting textboxes into an Excel worksheet using
# Python and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook("textbox.xlsx")
worksheet = workbook.add_worksheet()
row = 4
col = 1

# The examples below show different textbox options and formatting. In each
# example the text describes the formatting.


# Example
text = "A simple textbox with some text"
worksheet.insert_textbox(row, col, text)
row += 10
```

```python
# Example
text = "A textbox with changed dimensions"
options = {
    "width": 256,
    "height": 100,
}
worksheet.insert_textbox(row, col, text, options)
row += 10

# Example
text = "A textbox with an offset in the cell"
options = {
    "x_offset": 10,
    "y_offset": 10,
}
worksheet.insert_textbox(row, col, text, options)
row += 10

# Example
text = "A textbox with scaling"
options = {
    "x_scale": 1.5,
    "y_scale": 0.8,
}
worksheet.insert_textbox(row, col, text, options)
row += 10

# Example
text = "A textbox with some long text that wraps around onto several lines"
worksheet.insert_textbox(row, col, text)
row += 10

# Example
text = "A textbox\nwith some\nnewlines\n\nand paragraphs"
worksheet.insert_textbox(row, col, text)
row += 10

# Example
text = "A textbox with a solid fill background"
options = {
    "fill": {"color": "red"},
}
worksheet.insert_textbox(row, col, text, options)
row += 10

# Example
text = "A textbox with a no fill background"
options = {
    "fill": {"none": True},
}
worksheet.insert_textbox(row, col, text, options)
row += 10
```

```python
# Example
text = "A textbox with a gradient fill background"
options = {
    "gradient": {"colors": ["#DDEBCF", "#9CB86E", "#156B13"]},
}
worksheet.insert_textbox(row, col, text, options)
row += 10

# Example
text = "A textbox with a user defined border line"
options = {
    "border": {"color": "red", "width": 3, "dash_type": "round_dot"},
}
worksheet.insert_textbox(row, col, text, options)
row += 10

# Example
text = "A textbox with no border line"
options = {
    "border": {"none": True},
}
worksheet.insert_textbox(row, col, text, options)
row += 10

# Example
text = "Default alignment: top - left"
worksheet.insert_textbox(row, col, text)
row += 10

# Example
text = "Alignment: top - center"
options = {
    "align": {"horizontal": "center"},
}
worksheet.insert_textbox(row, col, text, options)
row += 10

# Example
text = "Alignment: middle - center"
options = {
    "align": {"vertical": "middle", "horizontal": "center"},
}
worksheet.insert_textbox(row, col, text, options)
row += 10

# Example
text = "Alignment: long text line that wraps and is centered"
options = {
    "align": {"vertical": "middle", "horizontal": "center", "text": "center"},
}
worksheet.insert_textbox(row, col, text, options)
row += 10
```

```python
# Example
text = "Font properties: bold"
options = {
    "font": {"bold": True},
}
worksheet.insert_textbox(row, col, text, options)
row += 10

# Example
text = "Font properties: various"
options = {
    "font": {"bold": True},
}
worksheet.insert_textbox(row, col, text, options)
row += 10

# Example
text = "Font properties: various"
options = {
    "font": {
        "bold": True,
        "italic": True,
        "underline": True,
        "name": "Arial",
        "color": "red",
        "size": 12,
    }
}
worksheet.insert_textbox(row, col, text, options)
row += 10

# Example
text = "Some text in a textbox with formatting"
options = {
    "font": {"color": "white"},
    "align": {"vertical": "middle", "horizontal": "center"},
    "gradient": {"colors": ["red", "blue"]},
}
worksheet.insert_textbox(row, col, text, options)
row += 10

# Example
text = ""
options = {
    "textlink": "=$F$185",
}
worksheet.write("F185", "Text in a cell")
worksheet.insert_textbox(row, col, text, options)
row += 10

# Example
text = "Text rotated up"
options = {"text_rotation": 90}
```

```
    worksheet.insert_textbox(row, col, text, options)
    row += 10

    # Example
    text = "Text rotated down"
    options = {"text_rotation": -90}
    worksheet.insert_textbox(row, col, text, options)
    row += 10

    # Example
    text = "Text rotated vertically"
    options = {"text_rotation": 270}
    worksheet.insert_textbox(row, col, text, options)
    row += 10

    # Example
    text = "Textbox with hyperlink"
    options = {"url": "https://github.com/jmcnamara", "tip": "GitHub"}
    worksheet.insert_textbox(row, col, text, options)
    row += 10

    workbook.close()
```
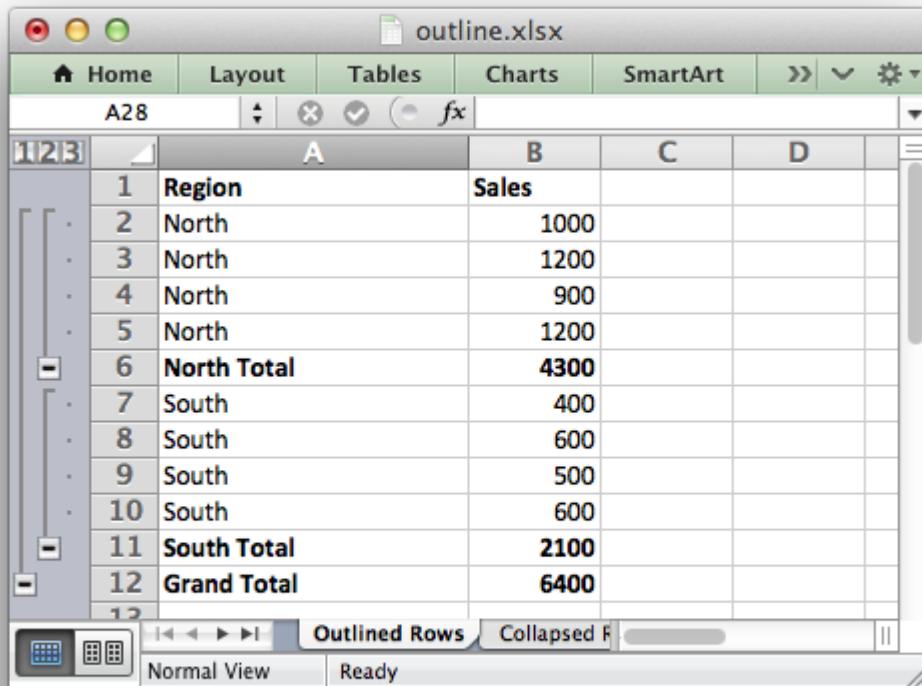
32.33 Example: Outline and Grouping

Examples of how use XlsxWriter to generate Excel outlines and grouping. See also *Working with Outlines and Grouping*.

```python
###############################################################################
#
# 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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
```

```python
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)
```

```python
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",
```
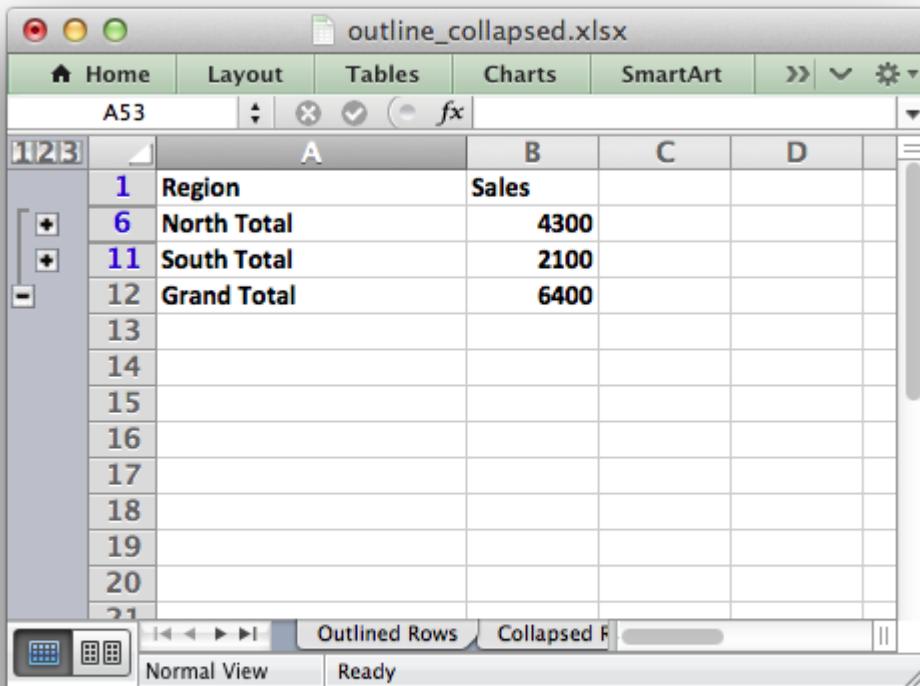
```
        "Level 1",
    ]

    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()
```

32.34 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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})
```

```python
# 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})
```

```python
    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.
```
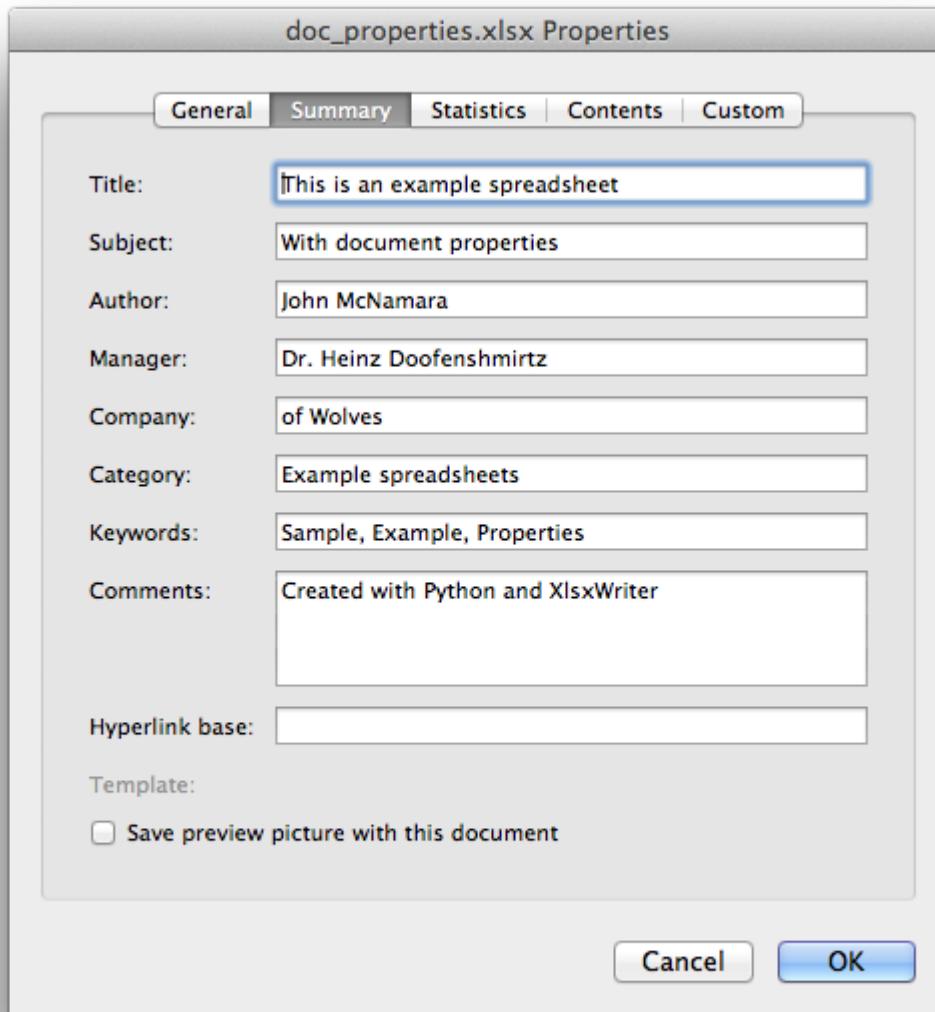
```python
# This is the same as the previous example except with collapsed columns.
#

# 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()
```

32.35 Example: Setting Document Properties

This program is an example setting document properties. See the `set_properties()` workbook method for more details.

```python
###############################################################################
#
# An example of adding document properties to a XlsxWriter file.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook("doc_properties.xlsx")
worksheet = workbook.add_worksheet()

workbook.set_properties(
    {
```
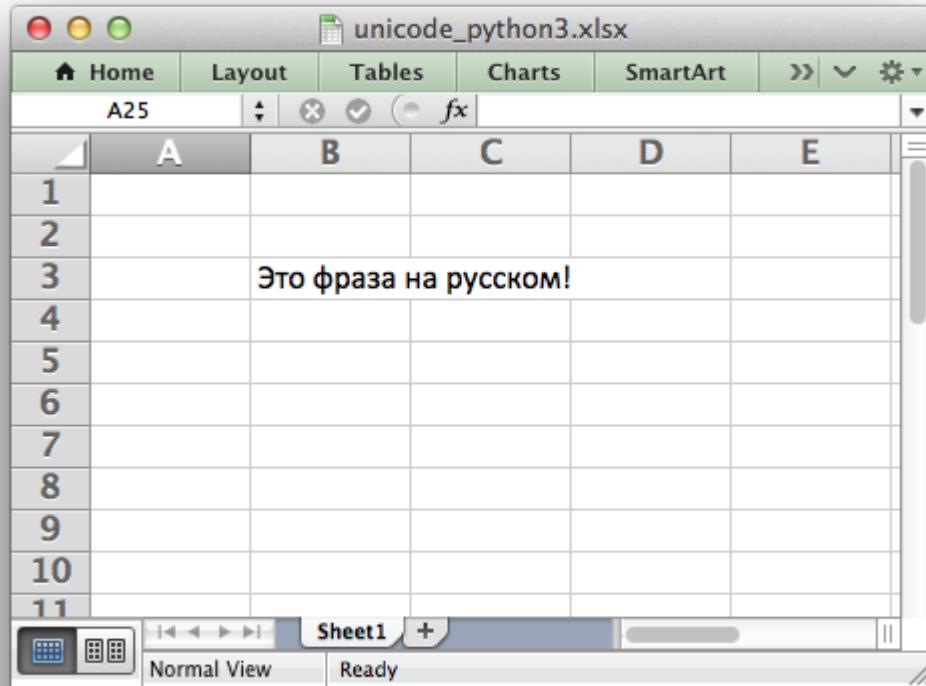
```
            "title": "This is an example spreadsheet",
            "subject": "With document properties",
            "author": "John McNamara",
            "manager": "Dr. Heinz Doofenshmirtz",
            "company": "of Wolves",
            "category": "Example spreadsheets",
            "keywords": "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()
```

32.36 Example: Simple Unicode with Python 3
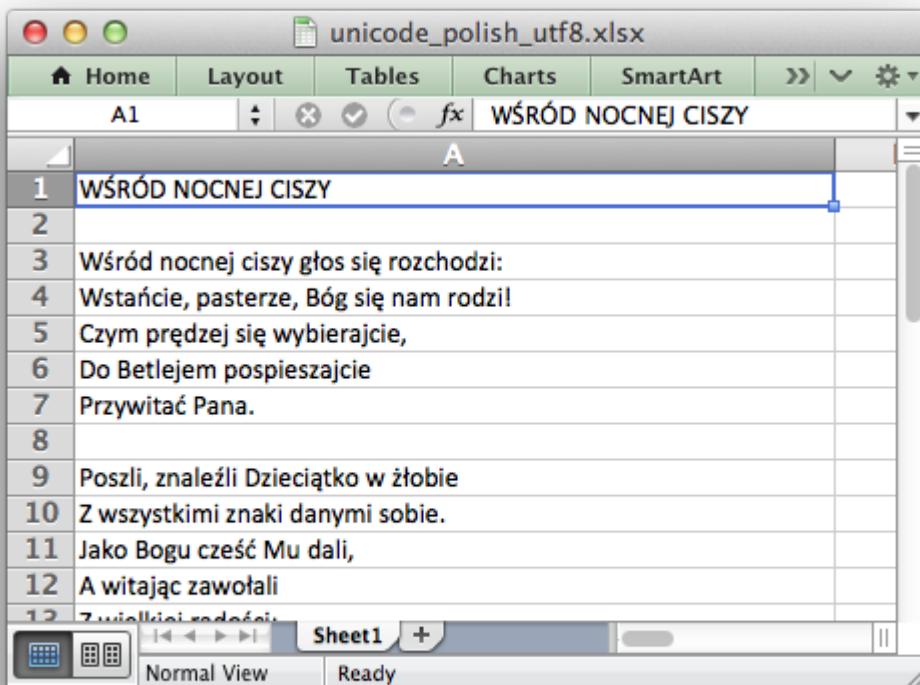
To write Unicode text in UTF-8 to a xlsxwriter file in Python 3 you just need to encode the file as UTF-8.

32.37 Example: Unicode - Polish in UTF-8

This program is an example of reading in data from a UTF-8 encoded text file and converting it to a worksheet.

The main trick is to ensure that the data read in is converted to UTF-8 within the Python program. The XlsxWriter module will then take care of writing the encoding to the Excel file.


```
###############################################################################
#
# 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

# Open the input file with the correct encoding.
textfile = open("unicode_polish_utf8.txt", mode="r", encoding="utf-8")
```

```python
# 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()
```
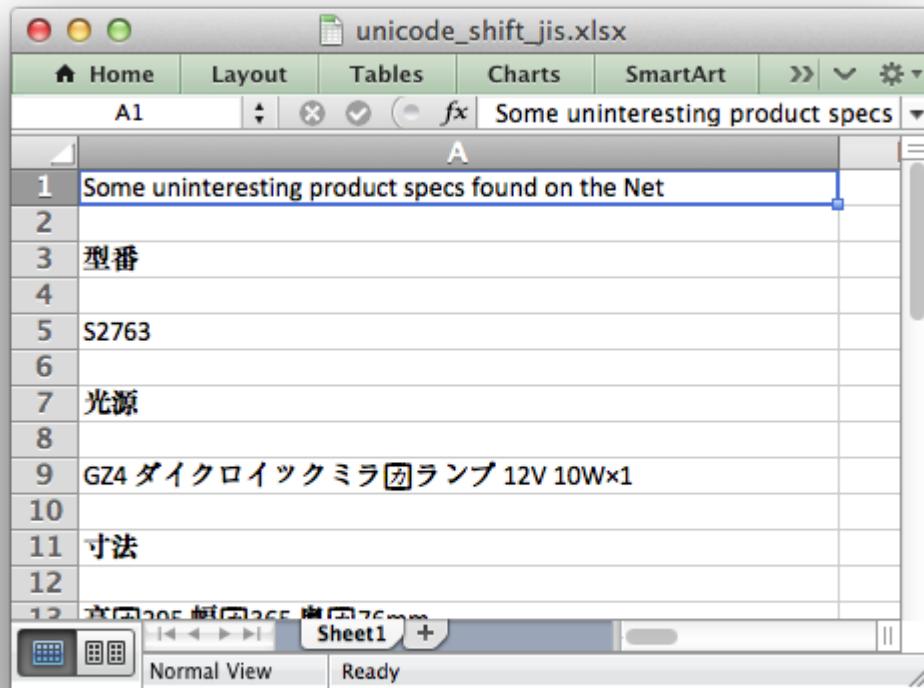
32.38 Example: Unicode - Shift JIS

This program is an example of reading in data from a Shift JIS encoded text file and converting it to a worksheet.

The main trick is to ensure that the data read in is converted to UTF-8 within the Python program. The XlsxWriter module will then take care of writing the encoding to the Excel file.

The encoding of the input data shouldn't matter once it can be converted to UTF-8 via the `codecs` module.

```
##############################################################################
#
# 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

# Open the input file with the correct encoding.
textfile = open("unicode_shift_jis.txt", mode="r", encoding="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.
```
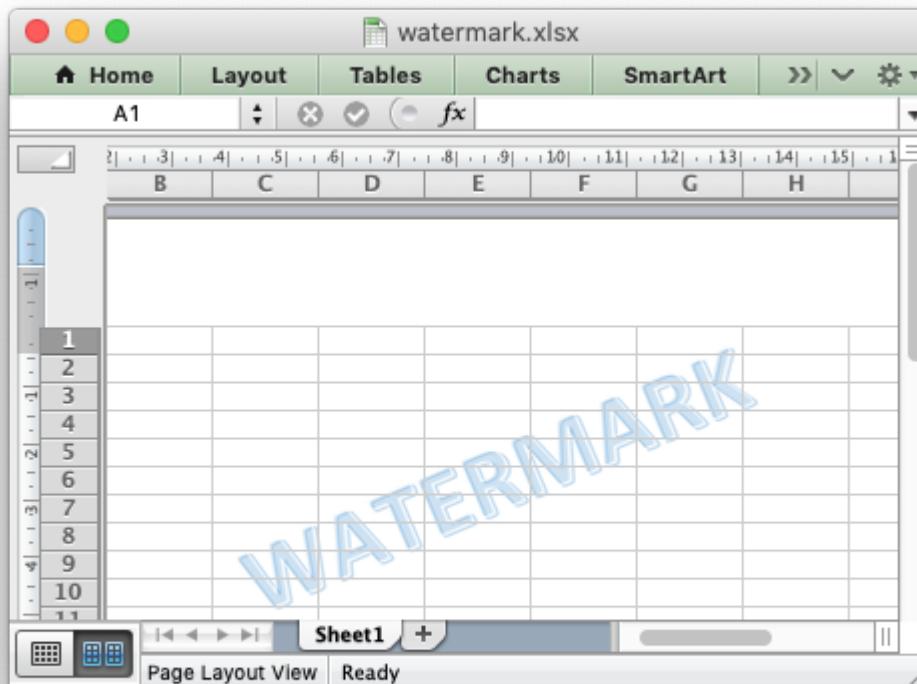
```python
    row = 0
    col = 0

    # 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()
```

32.39 Example: Setting a Worksheet Watermark

This program is an example of adding a worksheet watermark image using the method recommended in the Microsoft documentation: Add a watermark in Excel.


```python
##############################################################################
#
# An example of adding a worksheet watermark image using the XlsxWriter Python
```

```
    # module. This is based on the method of putting an image in the worksheet
    # header as suggested in the Microsoft documentation:
    # https://support.microsoft.com/en-us/office/add-a-watermark-in-excel-a372182a-d733-48
    #
    # SPDX-License-Identifier: BSD-2-Clause
    # Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
    #
    import xlsxwriter

    workbook = xlsxwriter.Workbook("watermark.xlsx")
    worksheet = workbook.add_worksheet()

    # Set a worksheet header with the watermark image.
    worksheet.set_header("&C&[Picture]", {"image_center": "watermark.png"})

    workbook.close()
```
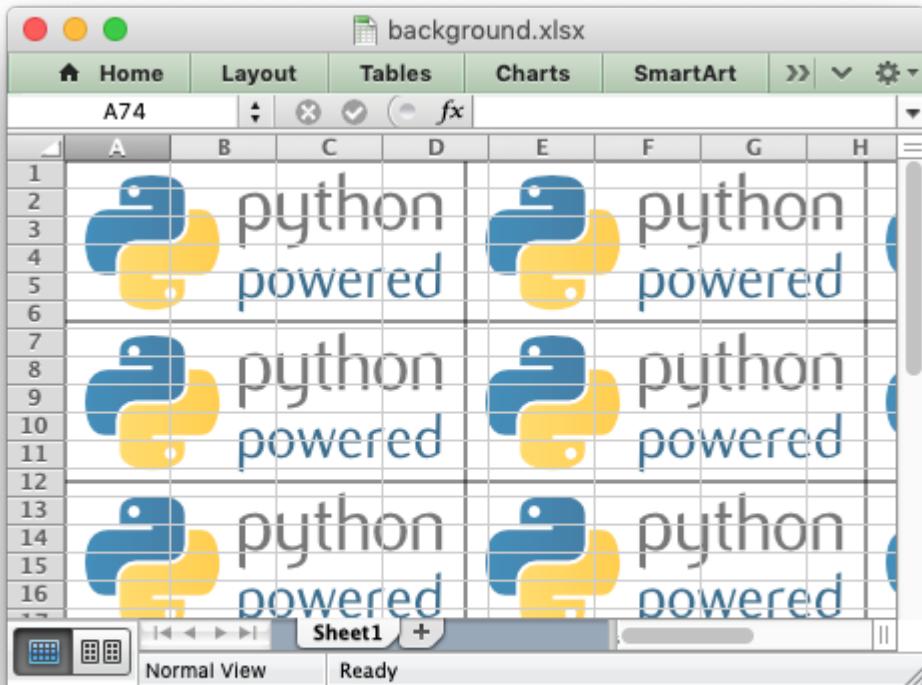
32.40 Example: Setting the Worksheet Background

This program is an example of setting a worksheet background image.   See the
set_background() method for more details.

```
###########################################################################
#
# An example of setting a worksheet background image with the XlsxWriter
# Python module.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter


# Create an new Excel file and add a worksheet.
workbook = xlsxwriter.Workbook("background.xlsx")
worksheet = workbook.add_worksheet()

# Set the background image.
worksheet.set_background("logo.png")

workbook.close()
```
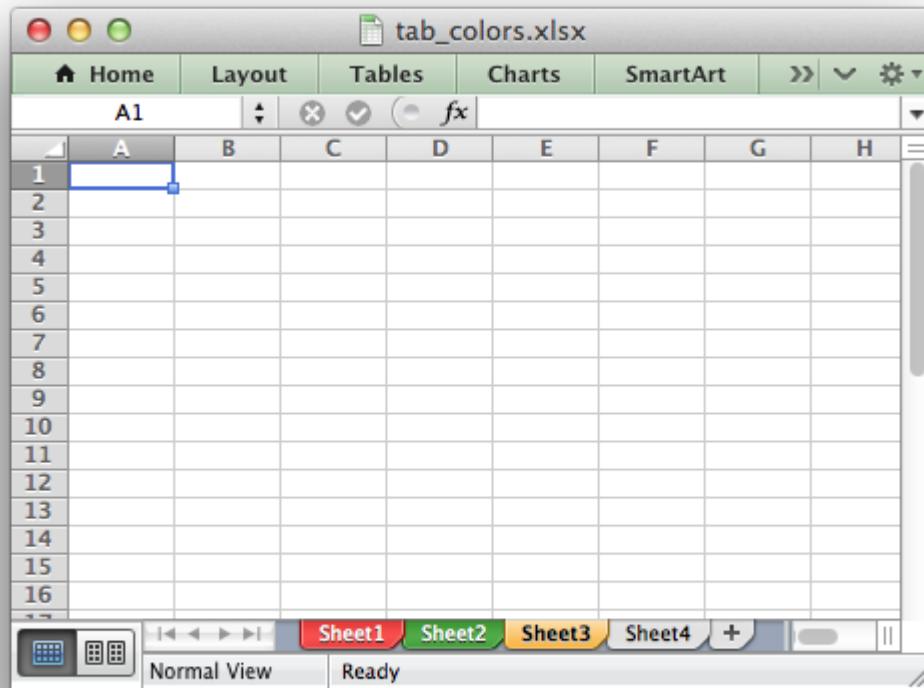
32.41 Example: Setting Worksheet Tab Colors

This program is an example of setting worksheet tab colors. See the set_tab_color() method
for more details.

```
########################################################################
#
# Example of how to set Excel worksheet tab colors using Python
# and the XlsxWriter module.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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 colors
worksheet1.set_tab_color("red")
worksheet2.set_tab_color("green")
worksheet3.set_tab_color("#FF9900")  # Orange

# worksheet4 will have the default color.
```
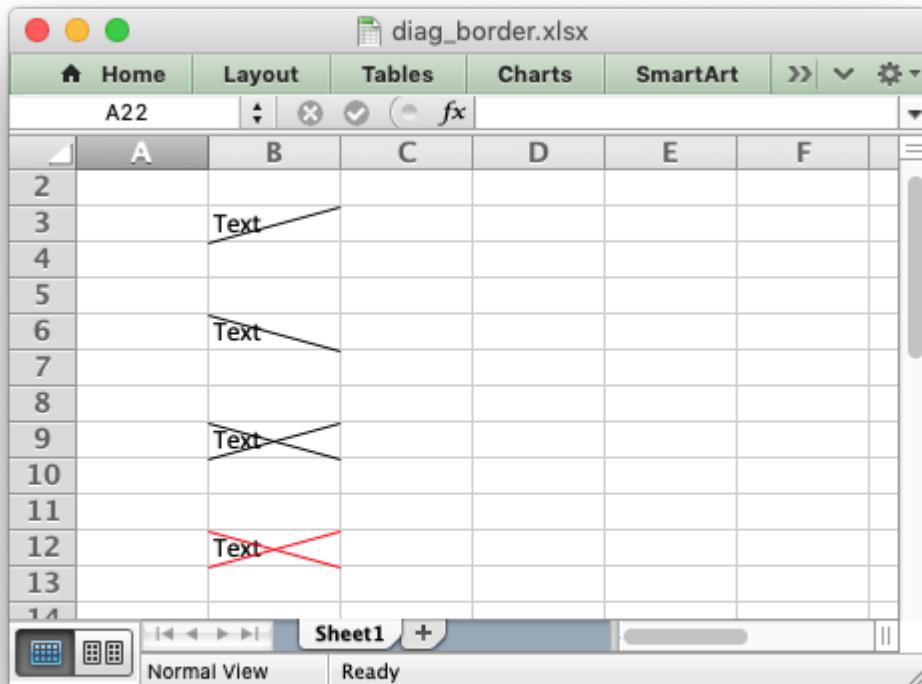
```
workbook.close()
```

32.42 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.

```python
###############################################################################
#
# A simple formatting example that demonstrates how to add diagonal cell
# borders with XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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})
```
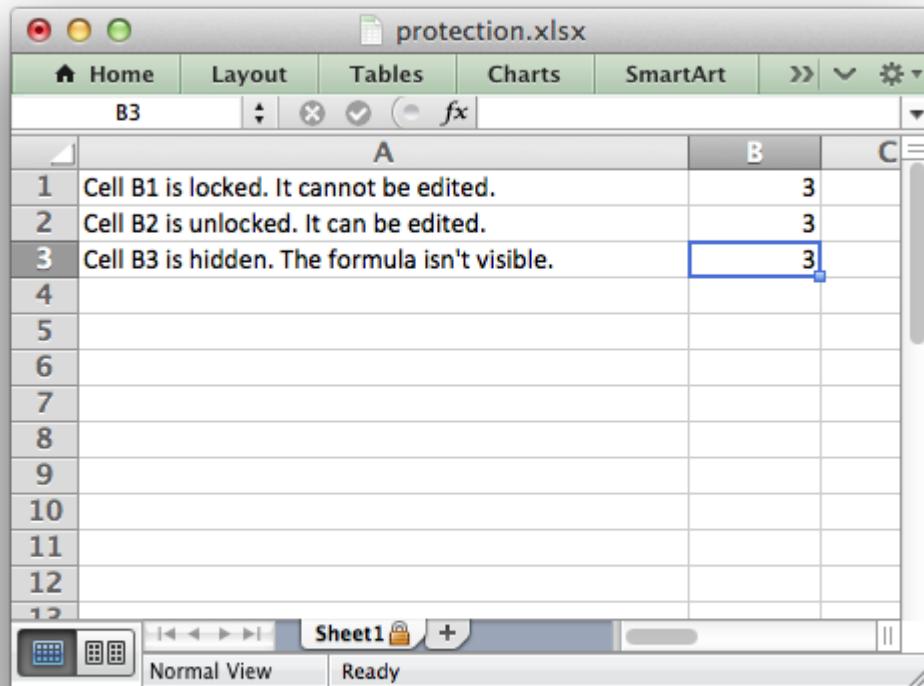
```
    format3 = workbook.add_format({"diag_type": 3})

    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()
```

32.43 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 and the Format `set_locked()` property.

Note, that Excel's behavior is that all cells are locked once you set the default protection. Therefore you need to explicitly unlock cells rather than explicitly lock them.

```
###############################################################################
#
# Example of cell locking and formula hiding in an Excel worksheet
# using Python and the XlsxWriter module.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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": False})
hidden = workbook.add_format({"hidden": True})

# 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.
```
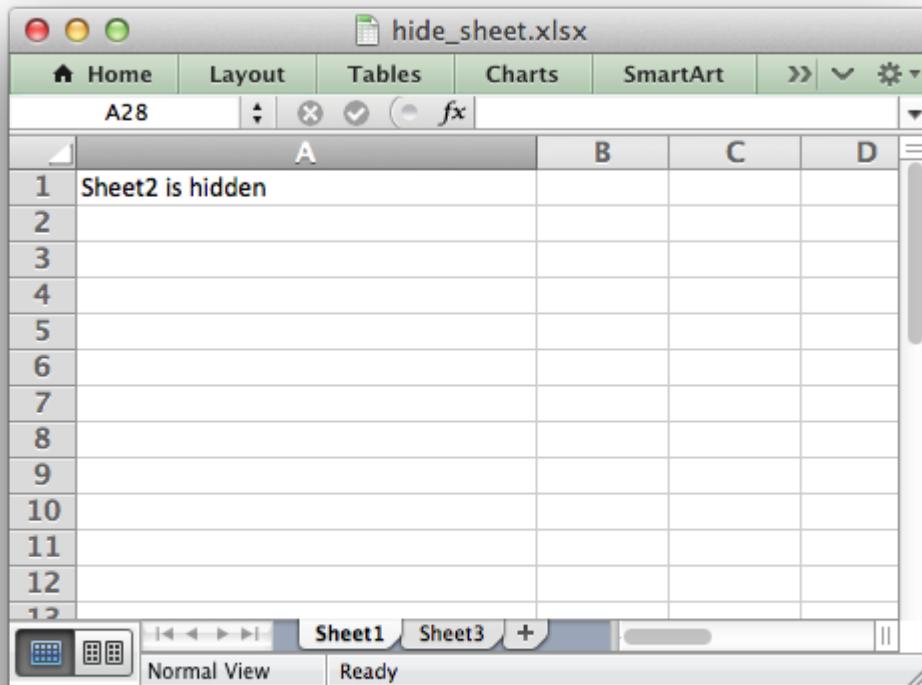
```
worksheet.write("A1", "Cell B1 is locked. It cannot be edited.")
worksheet.write("A2", "Cell B2 is unlocked. It can be edited.")
worksheet.write("A3", "Cell B3 is hidden. The formula isn't visible.")

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()
```

32.44 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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")

# Note, you can't hide the "active" worksheet, which generally is the
# first worksheet, since this would cause an Excel error. So, in order to hide
# the first sheet you will need to activate another worksheet:
#
#     worksheet2.activate()
#     worksheet1.hide()

workbook.close()
```

32.45 Example: Hiding Rows and Columns
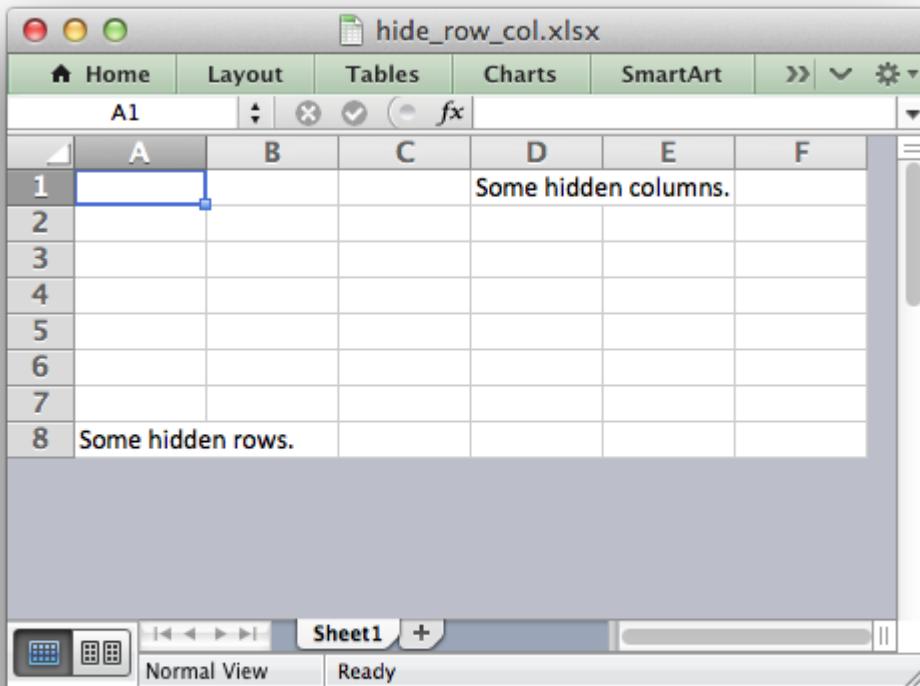
This program is an example of how to hide rows and columns in XlsxWriter.

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 optimization 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 optimization 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 optimizations to hide all rows that don't have data.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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):
```

```python
    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()
```
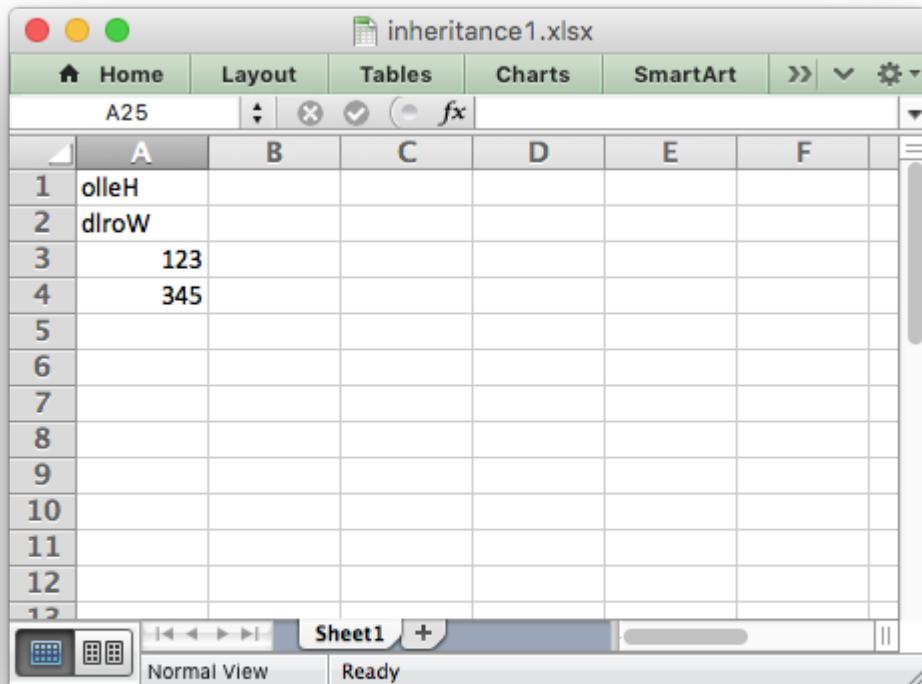
32.46 Example: Example of subclassing the Workbook and Worksheet classes

Example of how to subclass the Workbook and Worksheet objects.

We also override the default `worksheet.write()` method to show how that is done.


```python
###############################################################################
#
# Example of how to subclass the Workbook and Worksheet objects. We also
# override the default worksheet.write() method to show how that is done.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook
```

```python
from xlsxwriter.worksheet import Worksheet
from xlsxwriter.worksheet import convert_cell_args


class MyWorksheet(Worksheet):
    """
    Subclass of the XlsxWriter Worksheet class to override the default
    write() method.

    """

    @convert_cell_args
    def write(self, row, col, *args):
        data = args[0]

        # Reverse strings to demonstrate the overridden method.
        if isinstance(data, str):
            data = data[::-1]
            return self.write_string(row, col, data)
        else:
            # Call the parent version of write() as usual for other data.
            return super(MyWorksheet, self).write(row, col, *args)


class MyWorkbook(Workbook):
    """
    Subclass of the XlsxWriter Workbook class to override the default
    Worksheet class with our custom class.

    """

    def add_worksheet(self, name=None):
        # Overwrite add_worksheet() to create a MyWorksheet object.
        worksheet = super(MyWorkbook, self).add_worksheet(name, MyWorksheet)

        return worksheet


# Create a new MyWorkbook object.
workbook = MyWorkbook("inheritance1.xlsx")

# The code from now on will be the same as a normal "Workbook" program.
worksheet = workbook.add_worksheet()

# Write some data to test the subclassing.
worksheet.write("A1", "Hello")
worksheet.write("A2", "World")
worksheet.write("A3", 123)
worksheet.write("A4", 345)

workbook.close()
```
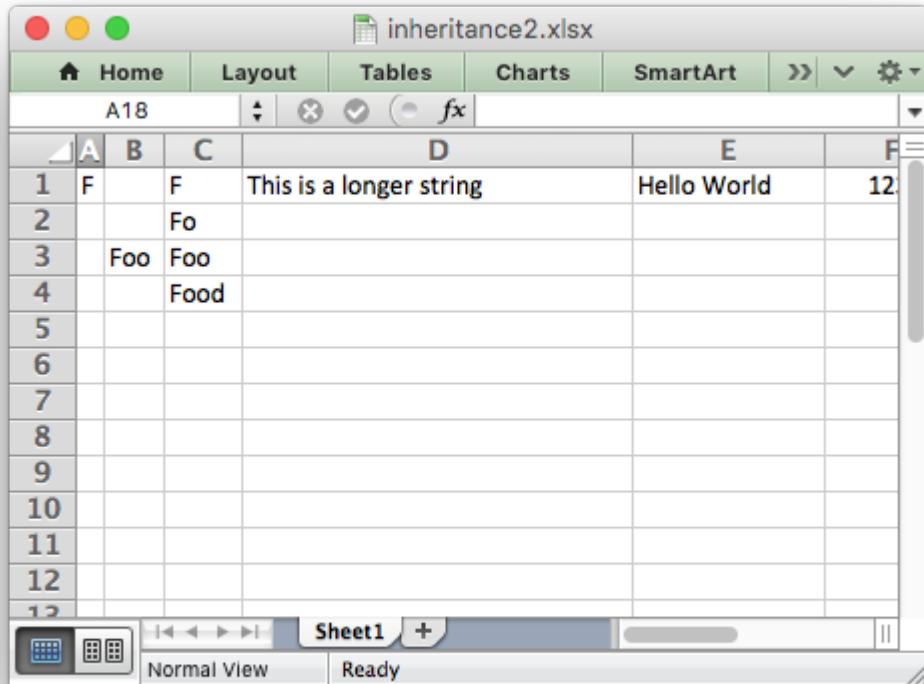
32.47 Example: Advanced example of subclassing

Example of how to subclass the Workbook and Worksheet objects. See also the simpler *inheritance1.py* example.

In this example we see an approach to implementing a simulated autofit in a user application. This works by overriding the `write_string()` method to track the maximum width string in each column and then set the column widths when closing the workbook.

Some notes on this:

- **This isn't a fully functional autofit example** (as shown by the longer strings in the screen shot). It is only a proof or concept or a framework to try out solutions. See the the worksheet `autofit()` method instead.

- The hard part is coming up with an accurate (or mainly accurate) `excel_string_width()` function. One possibility is to use the PIL ImageFont() method and convert the pixel width back to a character width.

- A more rigorous approach would have to consider font sizes, bold, italic, etc.

- The `set_column()` calls in `close()` will override any others set by the user. They also don't set any column formats.

- It doesn't work for horizontal merge ranges.

- There are probably some other corner cases hiding here.

```
###########################################################################
#
# Example of how to subclass the Workbook and Worksheet objects. See also the
# simpler inheritance1.py example.
#
# In this example we see an approach to implementing a simulated autofit in a
# user application. This works by overriding the write_string() method to
# track the maximum width string in each column and then set the column
# widths.
#
# Note: THIS ISN'T A FULLY FUNCTIONAL AUTOFIT EXAMPLE. It is only a proof or
# concept or a framework to try out solutions.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook
from xlsxwriter.worksheet import Worksheet
from xlsxwriter.worksheet import convert_cell_args


def excel_string_width(str):
    """
    Calculate the length of the string in Excel character units. This is only
```

```
    an example and won't give accurate results. It will need to be replaced
    by something more rigorous.

    """
    string_width = len(str)

    if string_width == 0:
        return 0
    else:
        return string_width * 1.1


class MyWorksheet(Worksheet):
    """
    Subclass of the XlsxWriter Worksheet class to override the default
    write_string() method.

    """

    @convert_cell_args
    def write_string(self, row, col, string, cell_format=None):
        # Overridden write_string() method to store the maximum string width
        # seen in each column.

        # Check that row and col are valid and store max and min values.
        if self._check_dimensions(row, col):
            return -1

        # Set the min width for the cell. In some cases this might be the
        # default width of 8.43. In this case we use 0 and adjust for all
        # string widths.
        min_width = 0

        # Check if it the string is the largest we have seen for this column.
        string_width = excel_string_width(string)
        if string_width > min_width:
            max_width = self.max_column_widths.get(col, min_width)
            if string_width > max_width:
                self.max_column_widths[col] = string_width

        # Now call the parent version of write_string() as usual.
        return super(MyWorksheet, self).write_string(row, col, string, cell_format)


class MyWorkbook(Workbook):
    """
    Subclass of the XlsxWriter Workbook class to override the default
    Worksheet class with our custom class.

    """

    def add_worksheet(self, name=None):
        # Overwrite add_worksheet() to create a MyWorksheet object.
```

```
            # Also add an Worksheet attribute to store the column widths.
            worksheet = super(MyWorkbook, self).add_worksheet(name, MyWorksheet)
            worksheet.max_column_widths = {}

            return worksheet

        def close(self):
            # We apply the stored column widths for each worksheet when we close
            # the workbook. This will override any other set_column() values that
            # may have been applied. This could be handled in the application code
            # below, instead.
            for worksheet in self.worksheets():
                for column, width in worksheet.max_column_widths.items():
                    worksheet.set_column(column, column, width)

            return super(MyWorkbook, self).close()


    # Create a new MyWorkbook object.
    workbook = MyWorkbook("inheritance2.xlsx")

    # The code from now on will be the same as a normal "Workbook" program.
    worksheet = workbook.add_worksheet()

    # Write some data to test column fitting.
    worksheet.write("A1", "F")

    worksheet.write("B3", "Foo")

    worksheet.write("C1", "F")
    worksheet.write("C2", "Fo")
    worksheet.write("C3", "Foo")
    worksheet.write("C4", "Food")

    worksheet.write("D1", "This is a longer string")

    # Write a string in row-col notation.
    worksheet.write(0, 4, "Hello World")

    # Write a number.
    worksheet.write(0, 5, 123456)

    workbook.close()
```
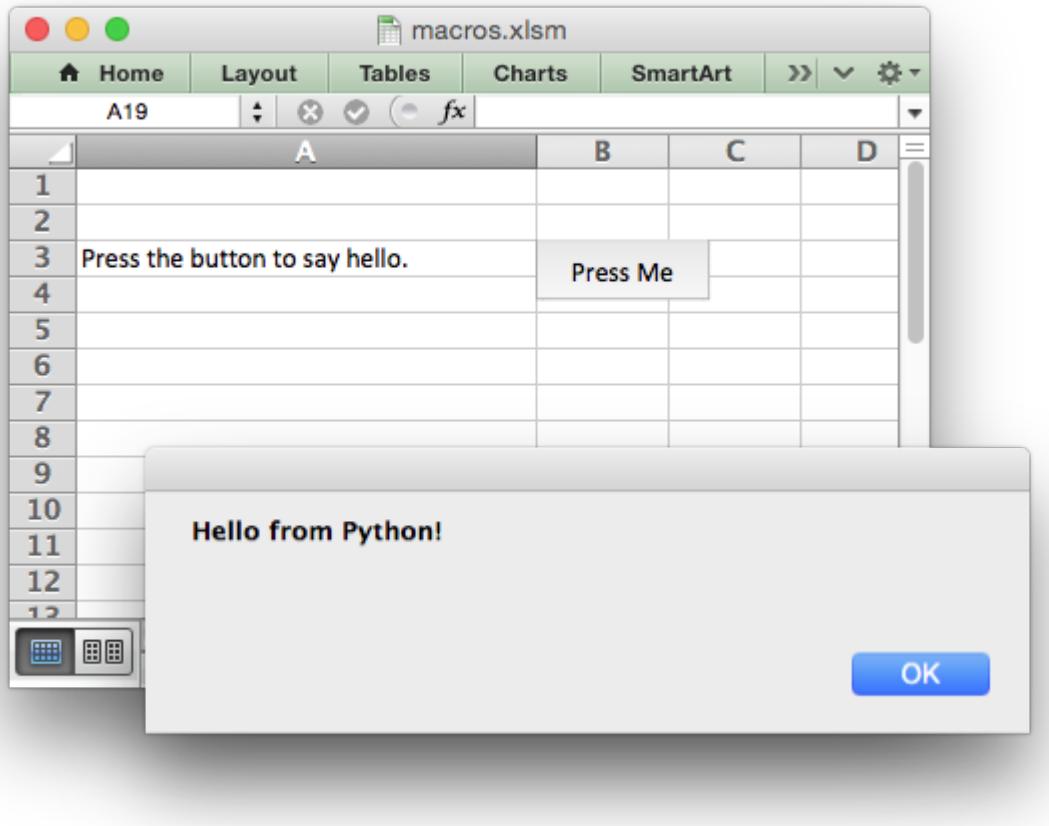
32.48 Example: Adding a VBA macro to a Workbook

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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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")
```
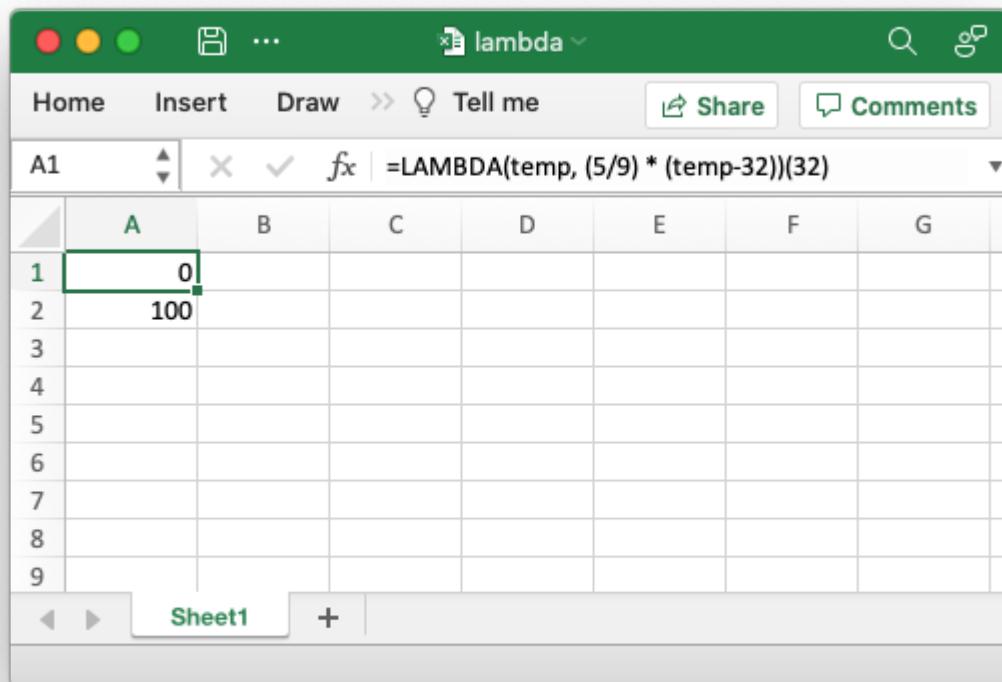
```
    # Show text for the end user.
    worksheet.write("A3", "Press the button to say hello.")

    # 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()
```

32.49 Example: Excel 365 LAMBDA() function

This program is an example of using the new Excel LAMBDA() function. It demonstrates how to create a lambda function in Excel and also how to assign a name to it so that it can be called as a user defined function. This particular example converts from Fahrenheit to Celsius.

See the *The Excel 365 LAMBDA() function* section of the documentation for more details.


```
########################################################################
#
# An example of using the new Excel LAMBDA() function with the XlsxWriter
# module.
#
```

```python
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

workbook = Workbook("lambda.xlsx")
worksheet = workbook.add_worksheet()


# Write a Lambda function to convert Fahrenheit to Celsius to a cell.
#
# Note that the lambda function parameters must be prefixed with
# "_xlpm.". These prefixes won't show up in Excel.
worksheet.write("A1", "=LAMBDA(_xlpm.temp, (5/9) * (_xlpm.temp-32))(32)")

# Create the same formula (without an argument) as a defined name and use that
# to calculate a value.
#
# Note that the formula name is prefixed with "_xlfn." (this is normally
# converted automatically by write_formula() but isn't for defined names)
# and note that the lambda function parameters are prefixed with
# "_xlpm.". These prefixes won't show up in Excel.
workbook.define_name("ToCelsius", "=_xlfn.LAMBDA(_xlpm.temp, (5/9) * (_xlpm.temp-32))'

# The user defined name needs to be written explicitly as a dynamic array
# formula.
worksheet.write_dynamic_array_formula("A2", "=ToCelsius(212)")

workbook.close()
```
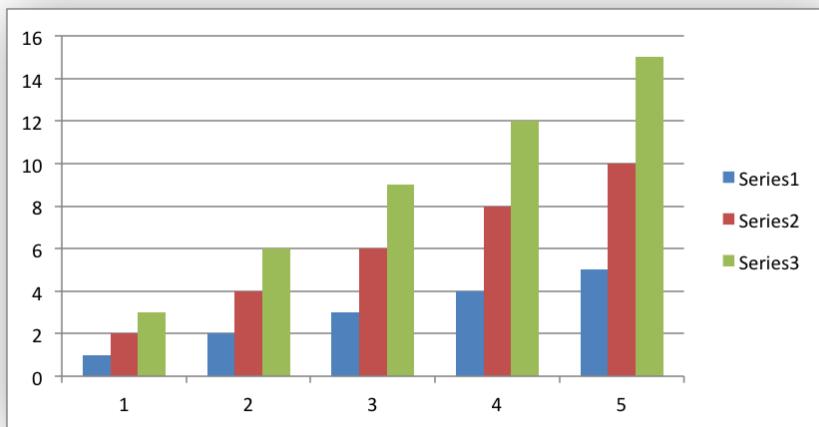
CHART EXAMPLES

The following are some of the examples included in the examples directory of the XlsxWriter distribution.

33.1 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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()
```

33.2 Example: Area Chart

Example of creating Excel Area charts.

Chart 1 in the following example is a default area chart:
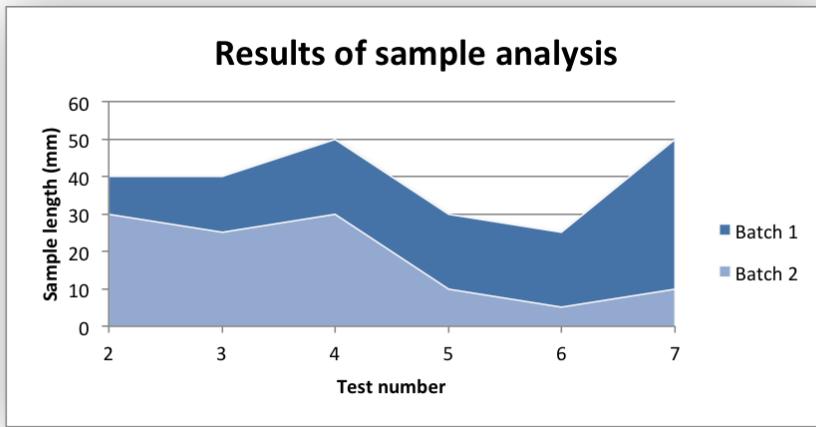
Chart 2 is a stacked area chart:
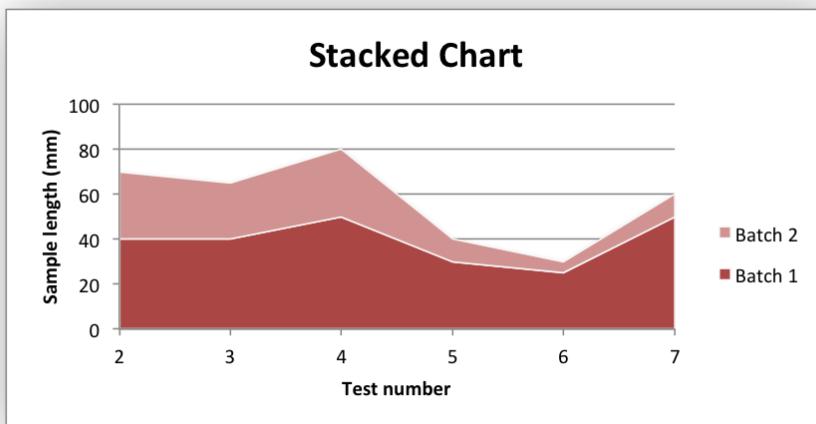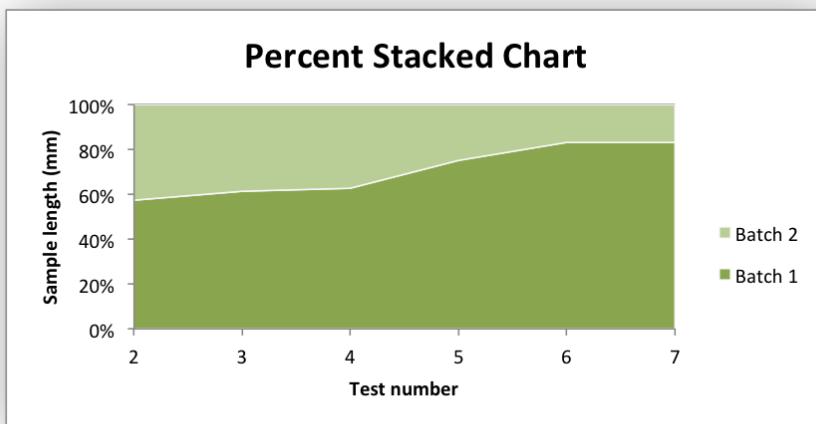
Chart 3 is a percentage stacked area chart:


```
########################################################################
#
# An example of creating Excel Area charts with Python and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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(
        {
```

```python
            "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()
```

33.3 Example: Bar Chart

Example of creating Excel Bar charts.

Chart 1 in the following example is a default bar chart:
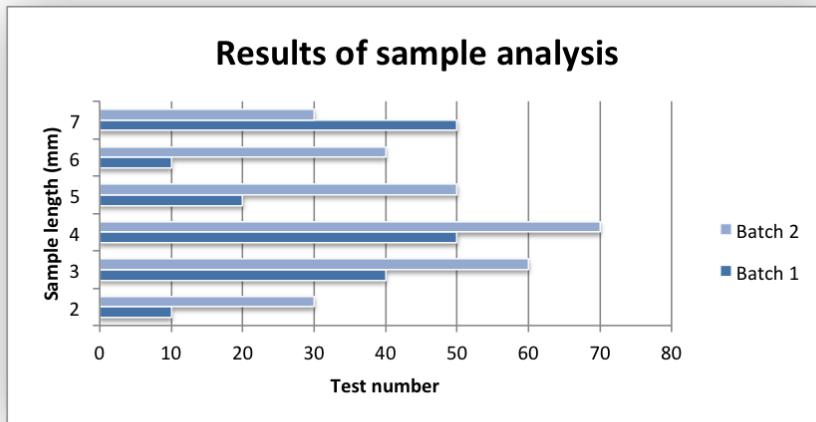

Chart 2 is a stacked bar chart:
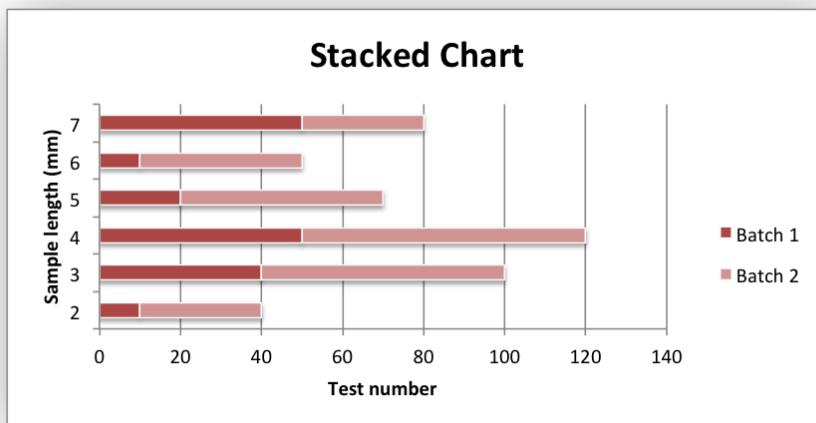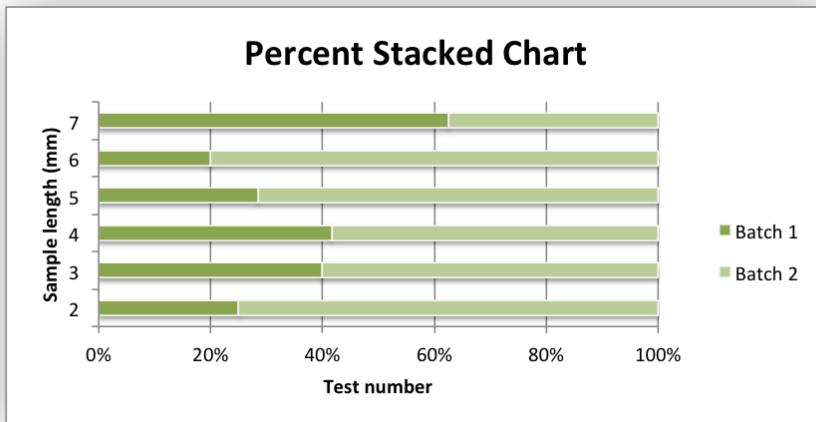
Chart 3 is a percentage stacked bar chart:


```python
###########################################################################
#
# An example of creating Excel Bar charts with Python and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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()
```

33.4 Example: Column Chart

Example of creating Excel Column charts.

Chart 1 in the following example is a default column chart:
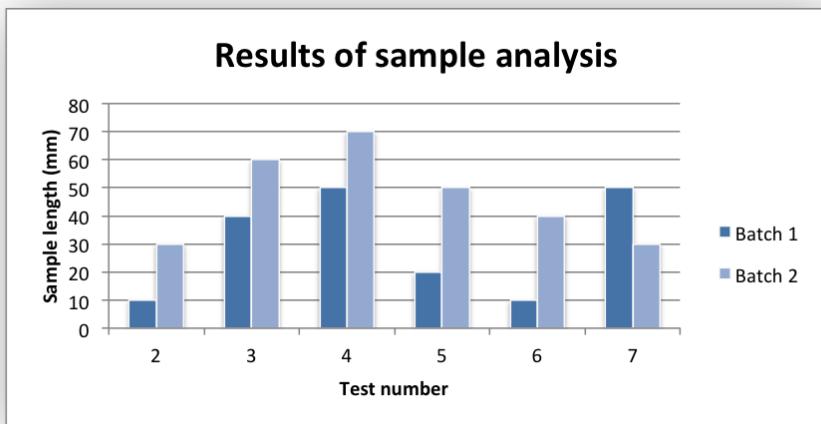
Chart 2 is a stacked column chart:
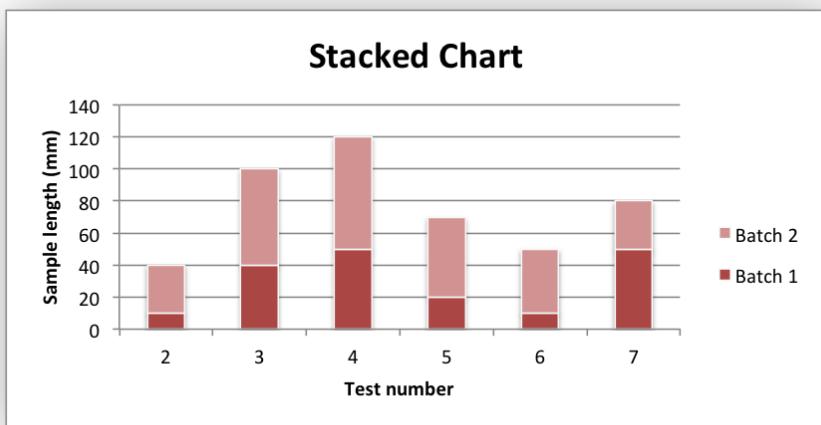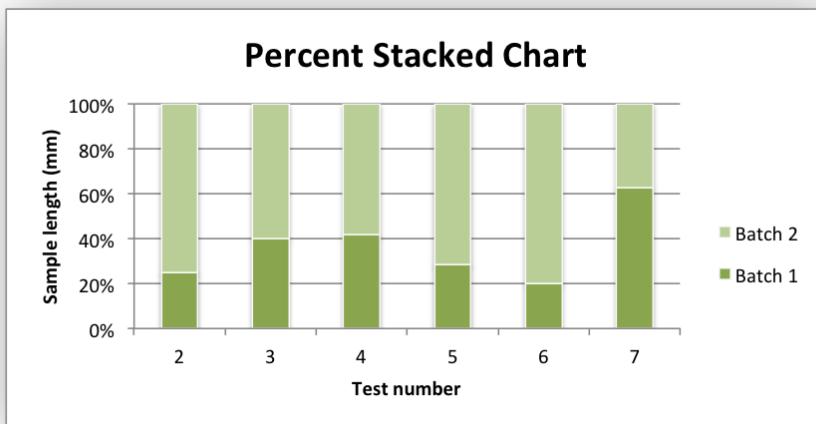
Chart 3 is a percentage stacked column chart:

```python
##############################################################################
#
# An example of creating Excel Column charts with Python and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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",
```

```python
        "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)"})
```

```python
    # 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()
```

33.5 Example: Line Chart

Example of creating an Excel line charts. 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 a default line chart:
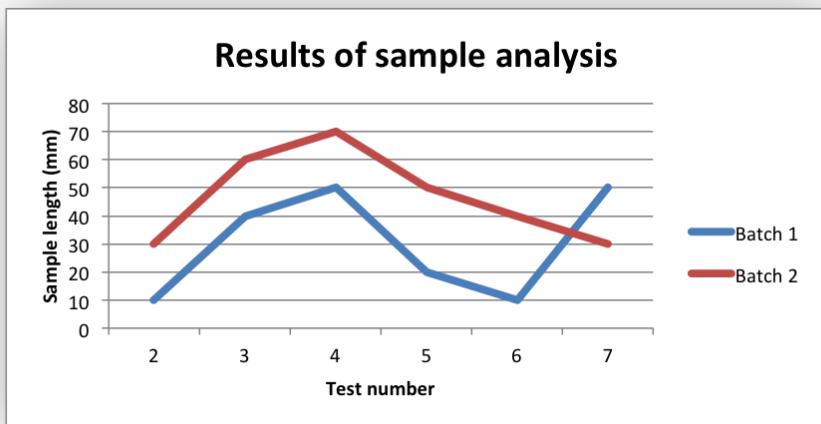
Chart 2 is a stacked line chart:
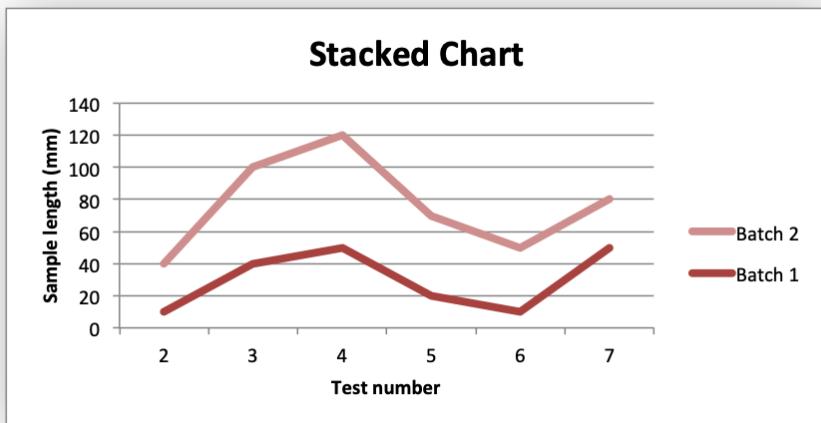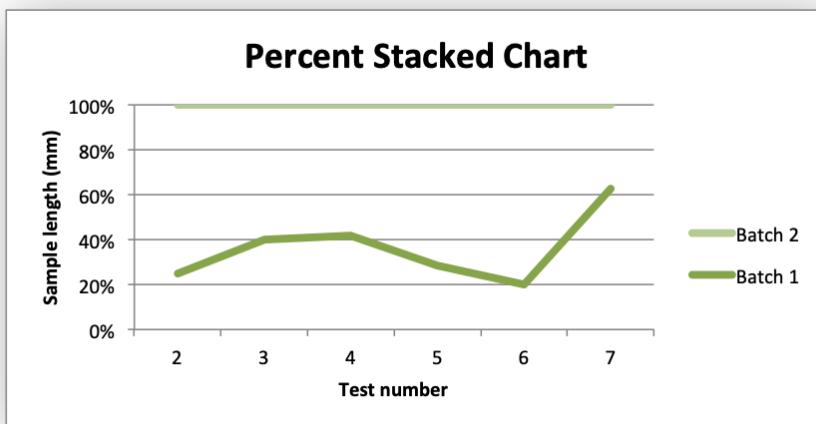
Chart 3 is a percentage stacked line chart:

```
#############################################################################
#
# An example of creating Excel Line charts with Python and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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 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. 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})

    #######################################################################
    #
    # Create a stacked line chart sub-type.
    #
    chart2 = workbook.add_chart({"type": "line", "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 line chart sub-type.
    #
    chart3 = workbook.add_chart({"type": "line", "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()
```

33.6 Example: Pie Chart

Example of creating Excel Pie charts. Chart 1 in the following example is:
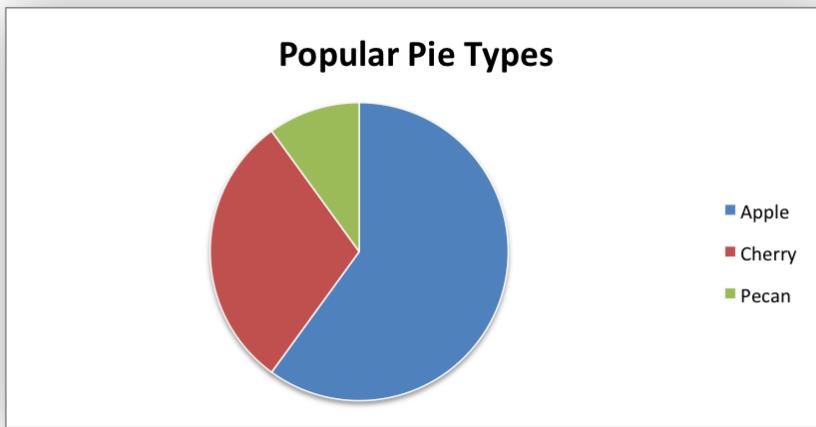
Chart 2 shows how to set segment colors.

It is possible to define chart colors 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.
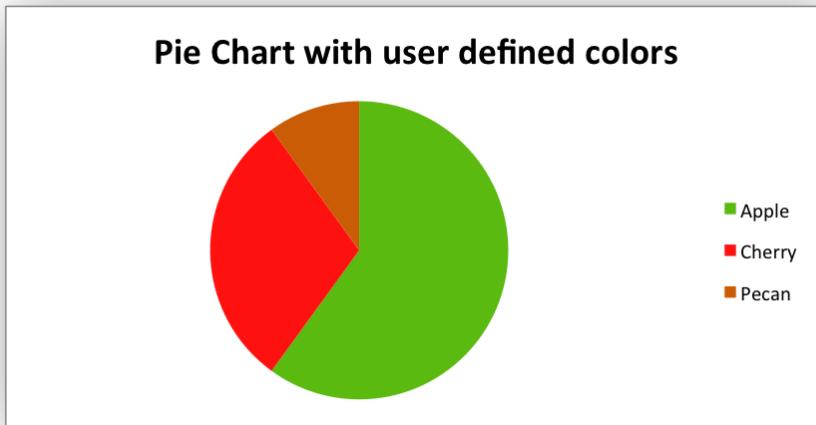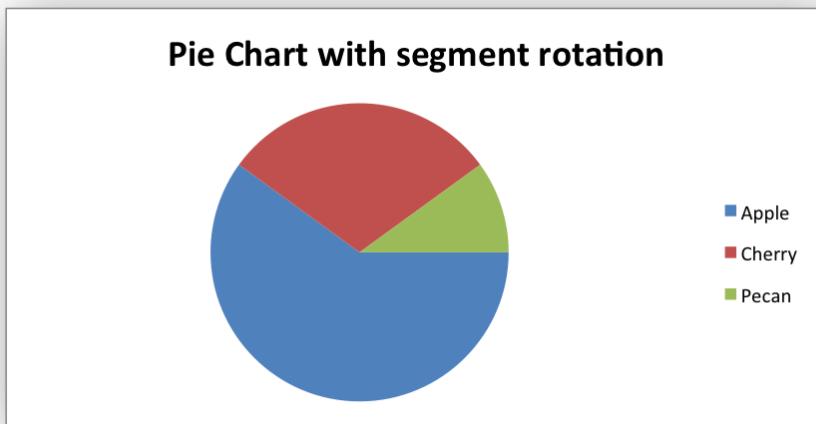
Chart 3 shows how to rotate the segments of the chart:

```
#######################################################################
#
# An example of creating Excel Pie charts with Python and XlsxWriter.
#
# The demo also shows how to set segment colors. 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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"})
```

```python
    # 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 colors.
    chart2.add_series(
        {
            "name": "Pie sales data",
            "categories": "=Sheet1!$A$2:$A$4",
            "values": "=Sheet1!$B$2:$B$4",
            "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!$A$2:$A$4",
        "values": "=Sheet1!$B$2:$B$4",
    }
)

# 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()
```

33.7 Example: Doughnut Chart

Example of creating Excel Doughnut charts. Chart 1 in the following example is:
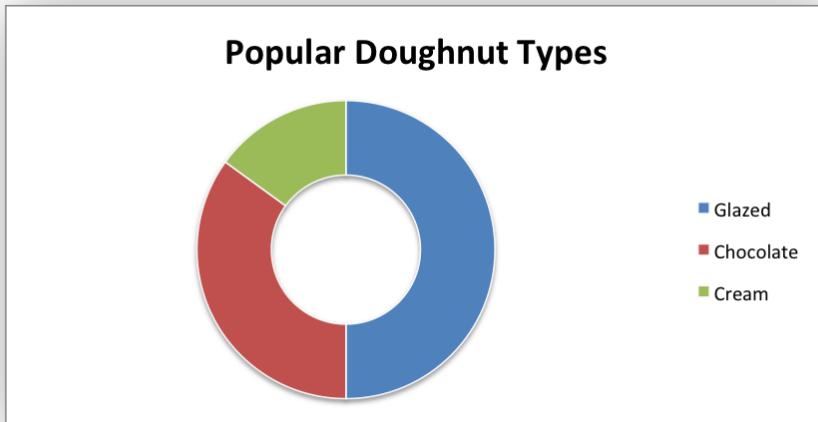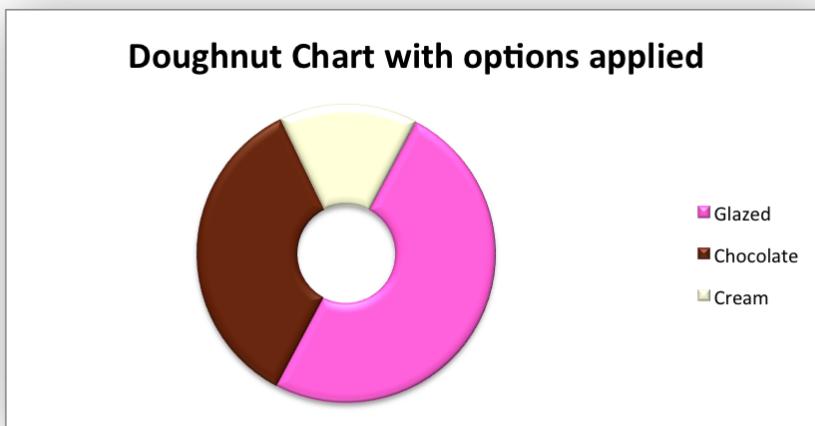

Chart 4 shows how to set segment colors and other options.

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 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 colors. 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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"})
```

```python
# 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 colors.
chart2.add_series(
    {
        "name": "Doughnut sales data",
        "categories": "=Sheet1!$A$2:$A$4",
        "values": "=Sheet1!$B$2:$B$4",
        "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"})
```

```python
    # Configure the series.
    chart3.add_series(
        {
            "name": "Doughnut sales data",
            "categories": "=Sheet1!$A$2:$A$4",
            "values": "=Sheet1!$B$2:$B$4",
        }
    )

    # 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!$A$2:$A$4",
            "values": "=Sheet1!$B$2:$B$4",
            "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()
```

33.8 Example: Scatter Chart

Example of creating Excel Scatter charts.

Chart 1 in the following example is a default scatter chart:
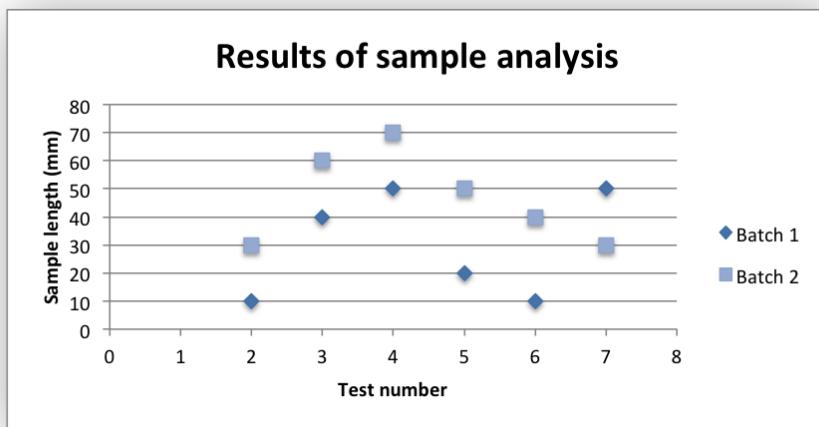

Chart 2 is a scatter chart with straight lines and markers:
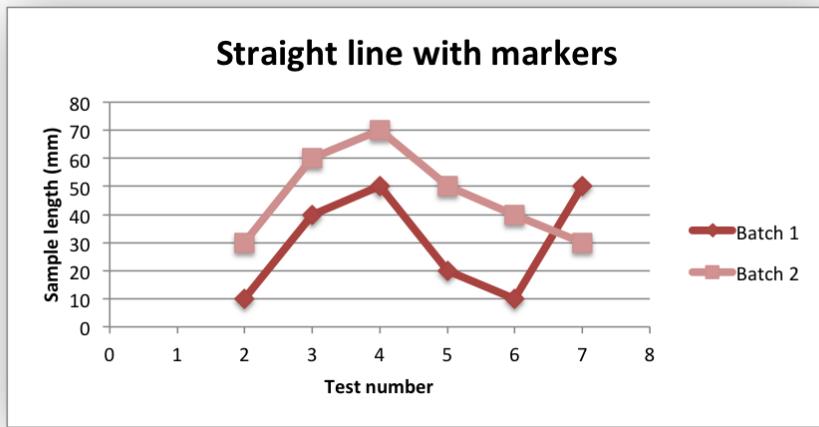

Chart 3 is a scatter chart with straight lines and no markers:
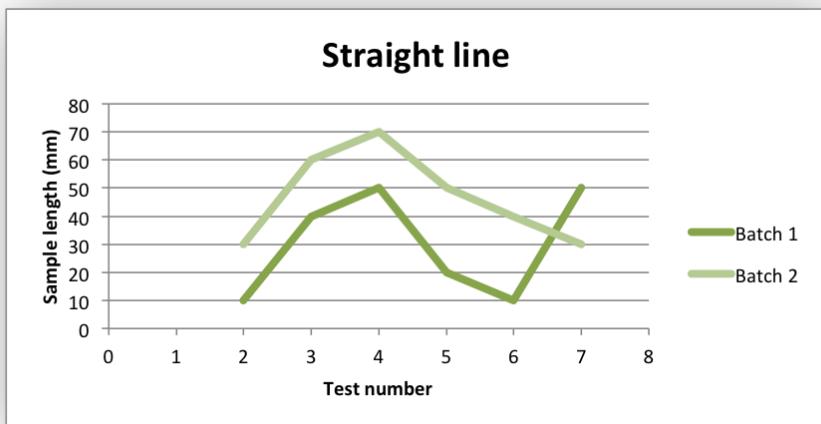
Chart 4 is a scatter chart with smooth lines and markers:
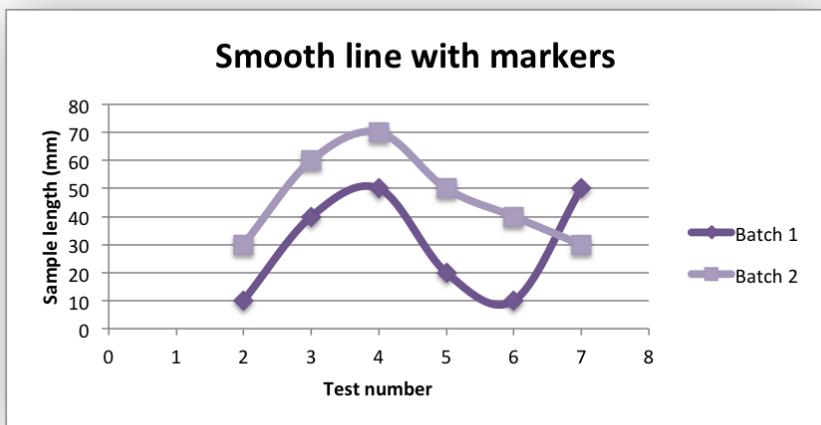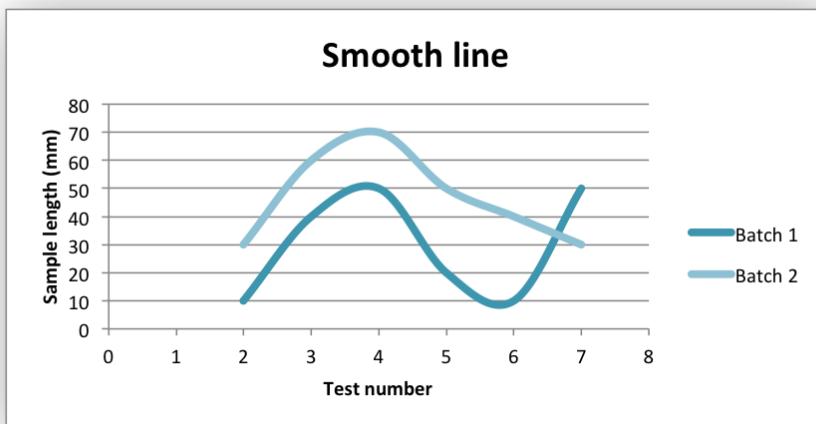
Chart 5 is a scatter chart with smooth lines and no markers:

```
########################################################################
#
# An example of creating Excel Scatter charts with Python and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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(
    {
```

```python
        "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"})
```

```python
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("D50", 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()
```

33.9 Example: Radar Chart

Example of creating Excel Column charts.

Chart 1 in the following example is a default radar chart:
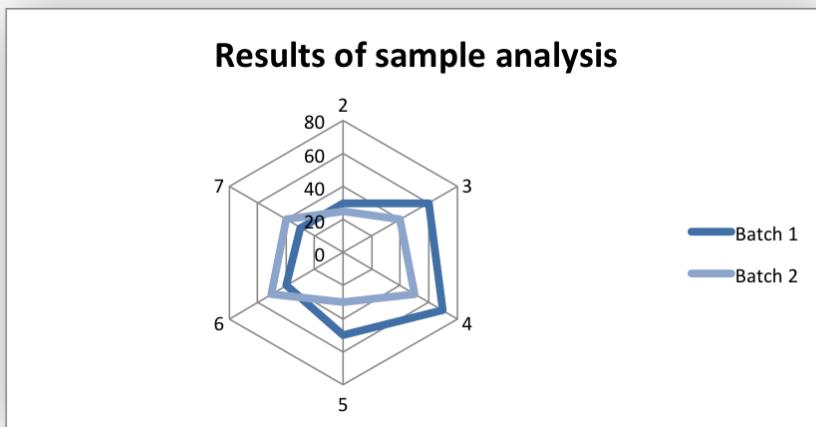
Chart 2 in the following example is a radar chart with markers:
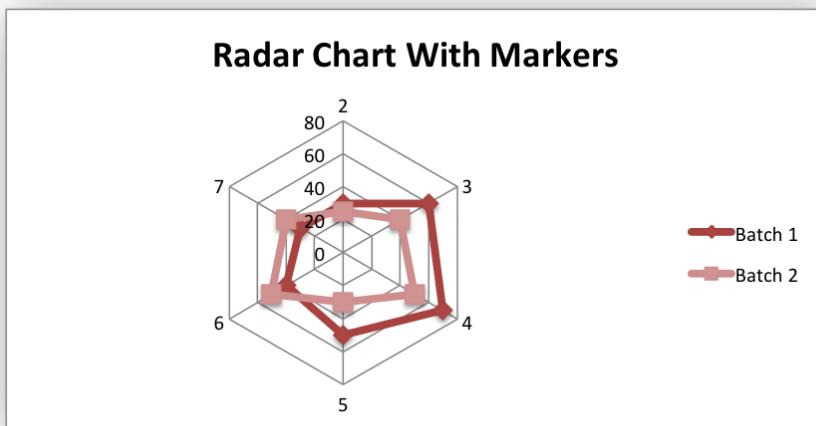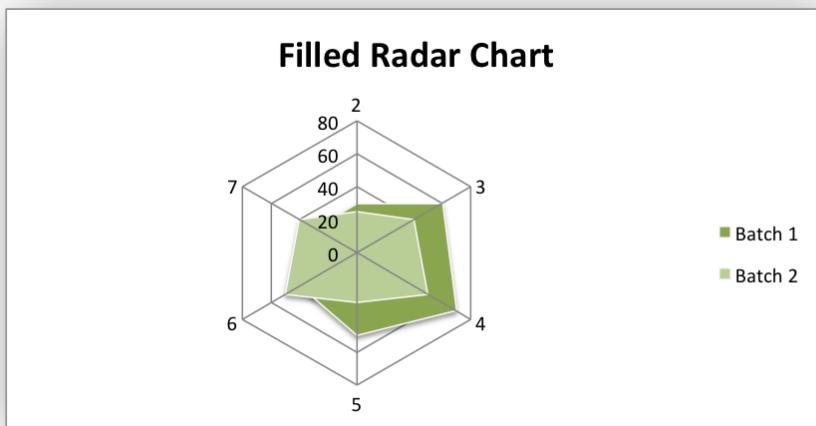
Chart 3 in the following example is a filled radar chart:

```
#############################################################################
#
# An example of creating Excel Radar charts with Python and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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.
chart1.set_title({"name": "Results of sample analysis"})

# 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.
chart2.set_title({"name": "Radar Chart With Markers"})

# Set an Excel chart style.
chart2.set_style(12)
```

```python
    # 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.
    chart3.set_title({"name": "Filled Radar Chart"})

    # 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()
```
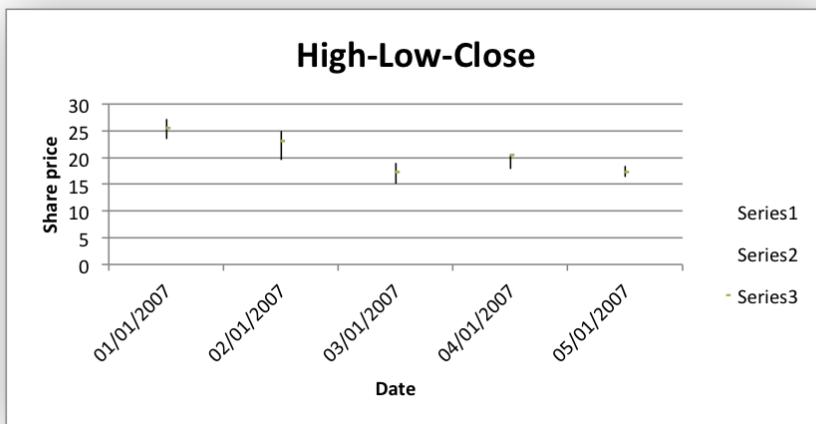
33.10 Example: Stock Chart

Example of creating and Excel HiLow-Close Stock chart.

Chart 1 in the following example is:

```python
#######################################################################
#
# An example of creating Excel Stock charts with Python and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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()
```
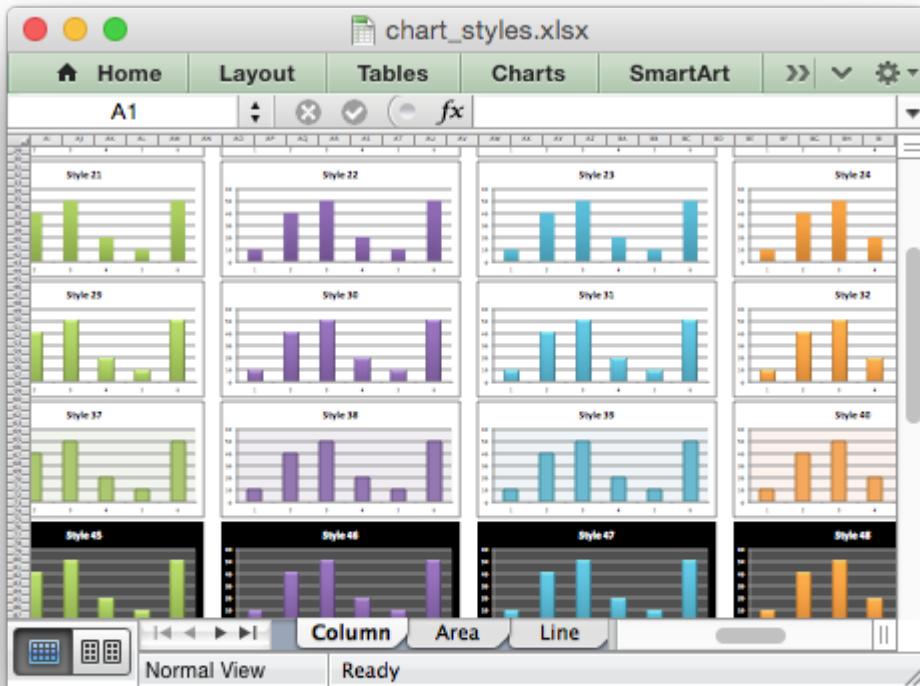
33.11 Example: Styles Chart

An example showing all 48 default chart styles available in Excel 2007 using the chart
set_style() method.

Note, these styles are not the same as the styles available in Excel 2013.

```
##########################################################################
#
# An example showing all 48 default chart styles available in Excel 2007
# using Python and XlsxWriter. Note, these styles are not the same as
# the styles available in Excel 2013.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook("chart_styles.xlsx")

# Show the styles for all of these chart types.
chart_types = ["column", "area", "line", "pie"]

for chart_type in chart_types:
    # Add a worksheet for each chart type.
    worksheet = workbook.add_worksheet(chart_type.title())
    worksheet.set_zoom(30)
    style_number = 1
```

```
        # Create 48 charts, each with a different style.
        for row_num in range(0, 90, 15):
            for col_num in range(0, 64, 8):
                chart = workbook.add_chart({"type": chart_type})
                chart.add_series({"values": "=Data!$A$1:$A$6"})
                chart.set_title({"name": "Style %d" % style_number})
                chart.set_legend({"none": True})
                chart.set_style(style_number)

                worksheet.insert_chart(row_num, col_num, chart)
                style_number += 1

    # Create a worksheet with data for the charts.
    data_worksheet = workbook.add_worksheet("Data")
    data = [10, 40, 50, 20, 10, 50]
    data_worksheet.write_column("A1", data)
    data_worksheet.hide()

    workbook.close()
```
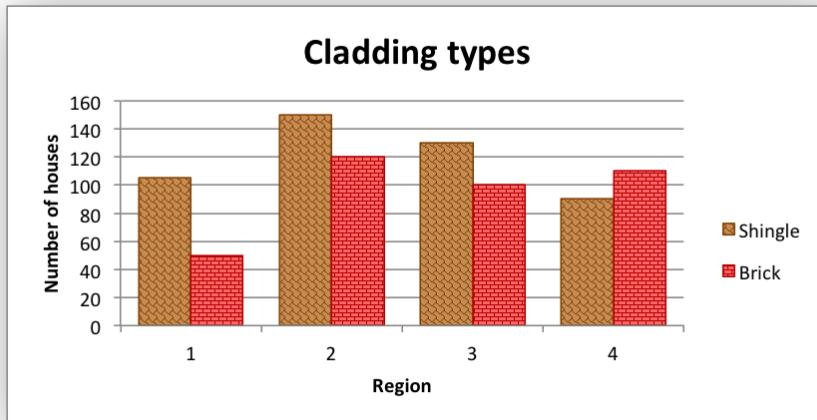
33.12 Example: Chart with Pattern Fills

Example of creating an Excel chart with pattern fills, in the columns.


```
#######################################################################
#
# An example of an Excel chart with patterns using Python and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook("chart_pattern.xlsx")
```

```python
worksheet = workbook.add_worksheet()
bold = workbook.add_format({"bold": 1})

# Add the worksheet data that the charts will refer to.
headings = ["Shingle", "Brick"]
data = [
    [105, 150, 130, 90],
    [50, 120, 100, 110],
]

worksheet.write_row("A1", headings, bold)
worksheet.write_column("A2", data[0])
worksheet.write_column("B2", data[1])

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

# Configure the charts. Add two series with patterns. The gap is used to make
# the patterns more visible.
chart.add_series(
    {
        "name": "=Sheet1!$A$1",
        "values": "=Sheet1!$A$2:$A$5",
        "pattern": {"pattern": "shingle", "fg_color": "#804000", "bg_color": "#c68c53"
        "border": {"color": "#804000"},
        "gap": 70,
    }
)

chart.add_series(
    {
        "name": "=Sheet1!$B$1",
        "values": "=Sheet1!$B$2:$B$5",
        "pattern": {
            "pattern": "horizontal_brick",
            "fg_color": "#b30000",
            "bg_color": "#ff6666",
        },
        "border": {"color": "#b30000"},
    }
)

# Add a chart title and some axis labels.
chart.set_title({"name": "Cladding types"})
chart.set_x_axis({"name": "Region"})
chart.set_y_axis({"name": "Number of houses"})

# Insert the chart into the worksheet.
worksheet.insert_chart("D2", chart)

workbook.close()
```
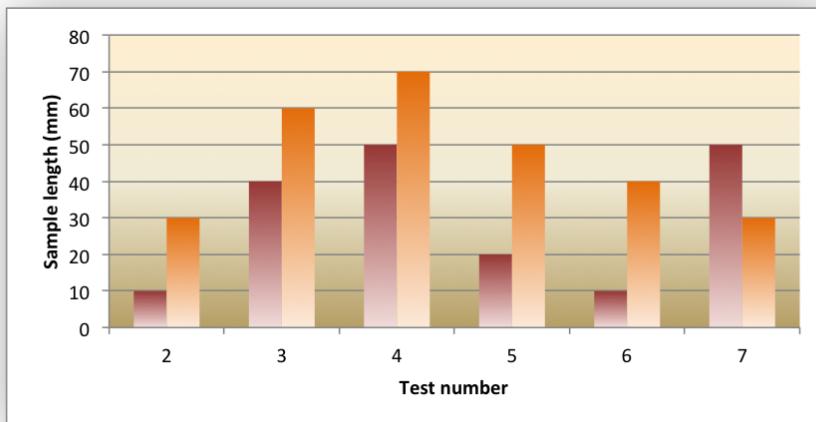
33.12. Example: Chart with Pattern Fills 585

33.13 Example: Chart with Gradient Fills

Example of creating an Excel chart with gradient fills, in the columns and in the plot area.


```python
#######################################################################
#
# An example of creating an Excel charts with gradient fills using
# Python and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook("chart_gradient.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.
chart = workbook.add_chart({"type": "column"})

# Configure the first series, including a gradient.
```

```
chart.add_series(
    {
        "name": "=Sheet1!$B$1",
        "categories": "=Sheet1!$A$2:$A$7",
        "values": "=Sheet1!$B$2:$B$7",
        "gradient": {"colors": ["#963735", "#F1DCDB"]},
    }
)

# Configure the second series, including a gradient.
chart.add_series(
    {
        "name": "=Sheet1!$C$1",
        "categories": "=Sheet1!$A$2:$A$7",
        "values": "=Sheet1!$C$2:$C$7",
        "gradient": {"colors": ["#E36C0A", "#FCEADA"]},
    }
)

# Set a gradient for the plotarea.
chart.set_plotarea({"gradient": {"colors": ["#FFEFD1", "#F0EBD5", "#B69F66"]}})


# Add some axis labels.
chart.set_x_axis({"name": "Test number"})
chart.set_y_axis({"name": "Sample length (mm)"})

# Turn off the chart legend.
chart.set_legend({"none": True})

# Insert the chart into the worksheet.
worksheet.insert_chart("E2", chart)

workbook.close()
```
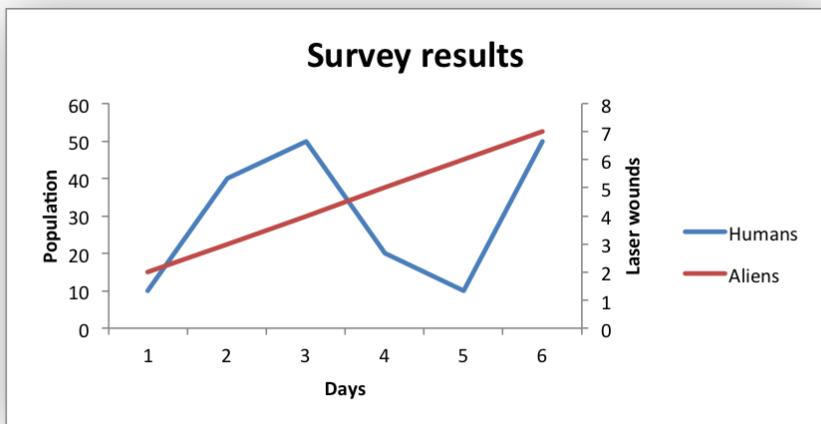
33.14 Example: Secondary Axis Chart

Example of creating an Excel Line chart with a secondary axis. Note, the primary and secondary chart type are the same. The next example shows a secondary chart of a different type.

```python
#############################################################################
#
# An example of creating an Excel Line chart with a secondary axis
# using Python and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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()
```
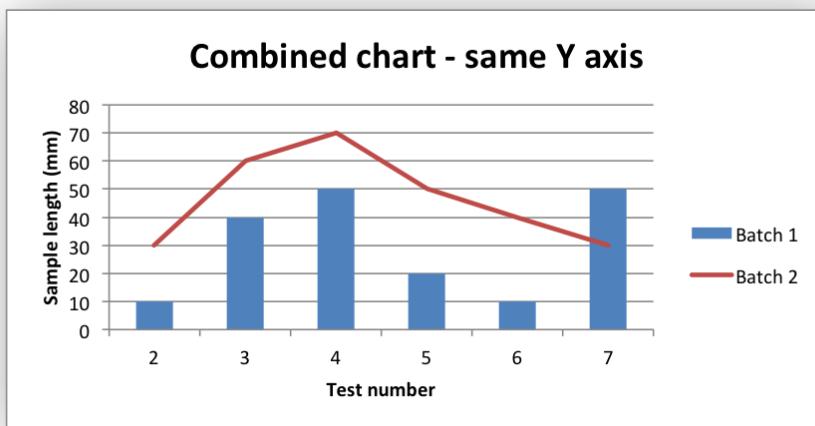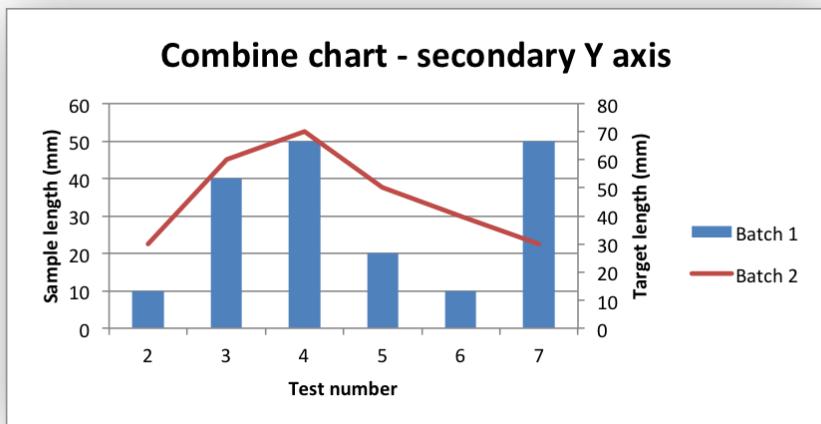
33.15 Example: Combined Chart

Example of creating combined Excel charts with two chart types.

In the first example we create a combined column and line chart that share the same X and Y axes.

In the second example we create a similar combined column and line chart except that the secondary chart has a secondary Y axis.

```python
#######################################################################
#
# An example of a Combined chart in XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

workbook = Workbook("chart_combined.xlsx")
worksheet = workbook.add_worksheet()

# Add a format for the headings.
bold = workbook.add_format({"bold": True})

# 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])


#
# In the first example we will create a combined column and line chart.
# They will share the same X and Y axes.
#

# Create a new column chart. This will use this as the primary chart.
column_chart1 = workbook.add_chart({"type": "column"})
```

```python
# Configure the data series for the primary chart.
column_chart1.add_series(
    {
        "name": "=Sheet1!$B$1",
        "categories": "=Sheet1!$A$2:$A$7",
        "values": "=Sheet1!$B$2:$B$7",
    }
)

# Create a new column chart. This will use this as the secondary chart.
line_chart1 = workbook.add_chart({"type": "line"})

# Configure the data series for the secondary chart.
line_chart1.add_series(
    {
        "name": "=Sheet1!$C$1",
        "categories": "=Sheet1!$A$2:$A$7",
        "values": "=Sheet1!$C$2:$C$7",
    }
)

# Combine the charts.
column_chart1.combine(line_chart1)

# Add a chart title and some axis labels. Note, this is done via the
# primary chart.
column_chart1.set_title({"name": "Combined chart - same Y axis"})
column_chart1.set_x_axis({"name": "Test number"})
column_chart1.set_y_axis({"name": "Sample length (mm)"})

# Insert the chart into the worksheet
worksheet.insert_chart("E2", column_chart1)

#
# In the second example we will create a similar combined column and line
# chart except that the secondary chart will have a secondary Y axis.
#

# Create a new column chart. This will use this as the primary chart.
column_chart2 = workbook.add_chart({"type": "column"})

# Configure the data series for the primary chart.
column_chart2.add_series(
    {
        "name": "=Sheet1!$B$1",
        "categories": "=Sheet1!$A$2:$A$7",
        "values": "=Sheet1!$B$2:$B$7",
    }
)

# Create a new column chart. This will use this as the secondary chart.
line_chart2 = workbook.add_chart({"type": "line"})
```

```python
# Configure the data series for the secondary chart. We also set a
# secondary Y axis via (y2_axis). This is the only difference between
# this and the first example, apart from the axis label below.
line_chart2.add_series(
    {
        "name": "=Sheet1!$C$1",
        "categories": "=Sheet1!$A$2:$A$7",
        "values": "=Sheet1!$C$2:$C$7",
        "y2_axis": True,
    }
)

# Combine the charts.
column_chart2.combine(line_chart2)

# Add a chart title and some axis labels.
column_chart2.set_title({"name": "Combine chart - secondary Y axis"})
column_chart2.set_x_axis({"name": "Test number"})
column_chart2.set_y_axis({"name": "Sample length (mm)"})

# Note: the y2 properties are on the secondary chart.
line_chart2.set_y2_axis({"name": "Target length (mm)"})

# Insert the chart into the worksheet
worksheet.insert_chart("E18", column_chart2)

workbook.close()
```
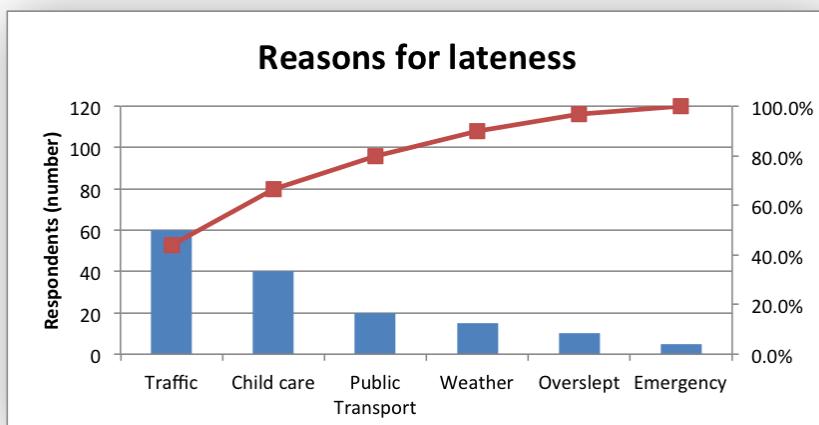
33.16 Example: Pareto Chart

Example of creating a Pareto chart with a secondary chart and axis.
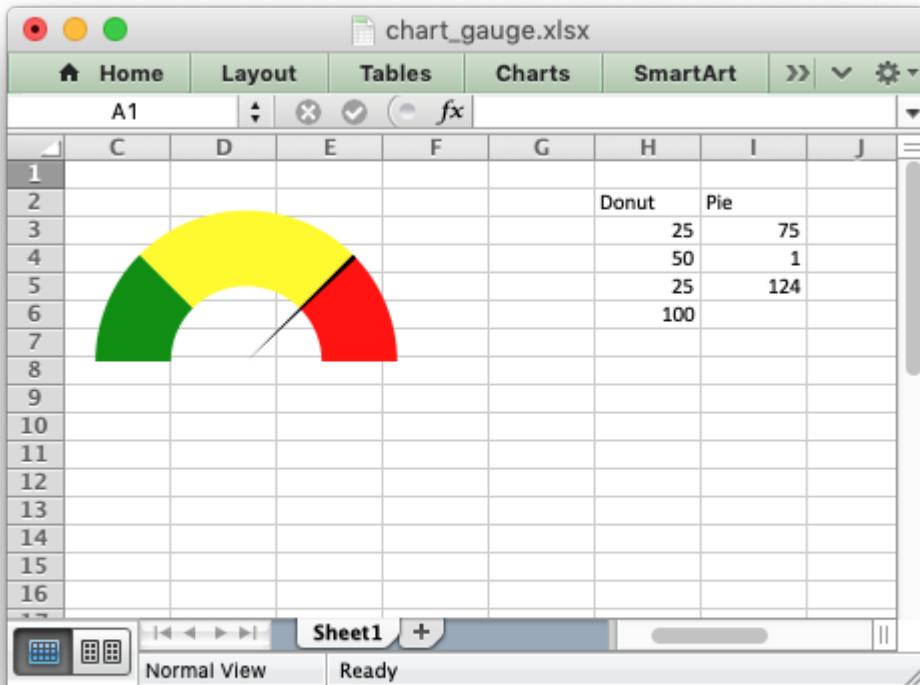

```python
######################################################################
#
```

```python
# An example of creating of a Pareto chart with Python and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook("chart_pareto.xlsx")
worksheet = workbook.add_worksheet()

# Formats used in the workbook.
bold = workbook.add_format({"bold": True})
percent_format = workbook.add_format({"num_format": "0.0%"})

# Widen the columns for visibility.
worksheet.set_column("A:A", 15)
worksheet.set_column("B:C", 10)

# Add the worksheet data that the charts will refer to.
headings = ["Reason", "Number", "Percentage"]

reasons = [
    "Traffic",
    "Child care",
    "Public Transport",
    "Weather",
    "Overslept",
    "Emergency",
]

numbers = [60, 40, 20, 15, 10, 5]
percents = [0.44, 0.667, 0.8, 0.9, 0.967, 1]

worksheet.write_row("A1", headings, bold)
worksheet.write_column("A2", reasons)
worksheet.write_column("B2", numbers)
worksheet.write_column("C2", percents, percent_format)


# Create a new column chart. This will be the primary chart.
column_chart = workbook.add_chart({"type": "column"})

# Add a series.
column_chart.add_series(
    {
        "categories": "=Sheet1!$A$2:$A$7",
        "values": "=Sheet1!$B$2:$B$7",
    }
)

# Add a chart title.
column_chart.set_title({"name": "Reasons for lateness"})
```

```python
    # Turn off the chart legend.
    column_chart.set_legend({"position": "none"})

    # Set the title and scale of the Y axes. Note, the secondary axis is set from
    # the primary chart.
    column_chart.set_y_axis({"name": "Respondents (number)", "min": 0, "max": 120})
    column_chart.set_y2_axis({"max": 1})

    # Create a new line chart. This will be the secondary chart.
    line_chart = workbook.add_chart({"type": "line"})

    # Add a series, on the secondary axis.
    line_chart.add_series(
        {
            "categories": "=Sheet1!$A$2:$A$7",
            "values": "=Sheet1!$C$2:$C$7",
            "marker": {"type": "automatic"},
            "y2_axis": 1,
        }
    )

    # Combine the charts.
    column_chart.combine(line_chart)

    # Insert the chart into the worksheet.
    worksheet.insert_chart("F2", column_chart)

    workbook.close()
```

33.17 Example: Gauge Chart

A Gauge Chart isn't a native chart type in Excel. It is constructed by combining a doughnut chart and a pie chart and by using some non-filled elements. This example follows the following online example of how to create a Gauge Chart in Excel: https://www.excel-easy.com/examples/gauge-chart.html

```
########################################################################
#
# An example of creating a Gauge Chart in Excel with Python and XlsxWriter.
#
# A Gauge Chart isn't a native chart type in Excel. It is constructed by
# combining a doughnut chart and a pie chart and by using some non-filled
# elements. This example follows the following online example of how to create
# a Gauge Chart in Excel: https://www.excel-easy.com/examples/gauge-chart.html
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook("chart_gauge.xlsx")
worksheet = workbook.add_worksheet()

chart_doughnut = workbook.add_chart({"type": "doughnut"})
chart_pie = workbook.add_chart({"type": "pie"})

# Add some data for the Doughnut and Pie charts. This is set up so the
# gauge goes from 0-100. It is initially set at 75%.
worksheet.write_column("H2", ["Donut", 25, 50, 25, 100])
worksheet.write_column("I2", ["Pie", 75, 1, "=200-I4-I3"])
```

```python
    # Configure the doughnut chart as the background for the gauge.
    chart_doughnut.add_series(
        {
            "name": "=Sheet1!$H$2",
            "values": "=Sheet1!$H$3:$H$6",
            "points": [
                {"fill": {"color": "green"}},
                {"fill": {"color": "yellow"}},
                {"fill": {"color": "red"}},
                {"fill": {"none": True}},
            ],
        }
    )

    # Rotate chart so the gauge parts are above the horizontal.
    chart_doughnut.set_rotation(270)

    # Turn off the chart legend.
    chart_doughnut.set_legend({"none": True})

    # Turn off the chart fill and border.
    chart_doughnut.set_chartarea(
        {
            "border": {"none": True},
            "fill": {"none": True},
        }
    )

    # Configure the pie chart as the needle for the gauge.
    chart_pie.add_series(
        {
            "name": "=Sheet1!$I$2",
            "values": "=Sheet1!$I$3:$I$6",
            "points": [
                {"fill": {"none": True}},
                {"fill": {"color": "black"}},
                {"fill": {"none": True}},
            ],
        }
    )

    # Rotate the pie chart/needle to align with the doughnut/gauge.
    chart_pie.set_rotation(270)

    # Combine the pie and doughnut charts.
    chart_doughnut.combine(chart_pie)

    # Insert the chart into the worksheet.
    worksheet.insert_chart("A1", chart_doughnut)

    workbook.close()
```
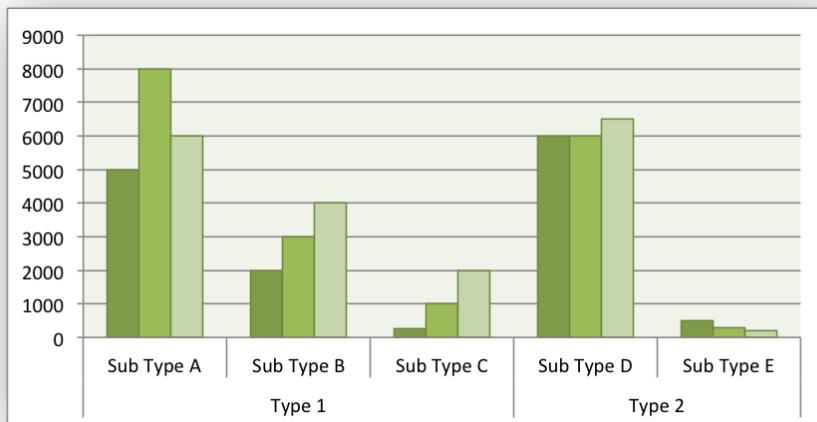
33.18 Example: Clustered Chart

Example of creating a clustered Excel chart where there are two levels of category on the X axis.

The categories in clustered charts are 2D ranges, instead of the more normal 1D ranges. The series are shown as formula strings for clarity but you can also use the a list syntax.

```python
###############################################################################
#
# A demo of a clustered category chart in XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
from xlsxwriter.workbook import Workbook

workbook = Workbook("chart_clustered.xlsx")
worksheet = workbook.add_worksheet()
bold = workbook.add_format({"bold": 1})

# Add the worksheet data that the charts will refer to.
headings = ["Types", "Sub Type", "Value 1", "Value 2", "Value 3"]
data = [
    ["Type 1", "Sub Type A", 5000, 8000, 6000],
    ["", "Sub Type B", 2000, 3000, 4000],
    ["", "Sub Type C", 250, 1000, 2000],
    ["Type 2", "Sub Type D", 6000, 6000, 6500],
    ["", "Sub Type E", 500, 300, 200],
]

worksheet.write_row("A1", headings, bold)

for row_num, row_data in enumerate(data):
    worksheet.write_row(row_num + 1, 0, row_data)

# Create a new chart object. In this case an embedded chart.
```

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

# Configure the series. Note, that the categories are 2D ranges (from column A
# to column B). This creates the clusters. The series are shown as formula
# strings for clarity but you can also use the list syntax. See the docs.
chart.add_series(
    {
        "categories": "=Sheet1!$A$2:$B$6",
        "values": "=Sheet1!$C$2:$C$6",
    }
)

chart.add_series(
    {
        "categories": "=Sheet1!$A$2:$B$6",
        "values": "=Sheet1!$D$2:$D$6",
    }
)

chart.add_series(
    {
        "categories": "=Sheet1!$A$2:$B$6",
        "values": "=Sheet1!$E$2:$E$6",
    }
)

# Set the Excel chart style.
chart.set_style(37)

# Turn off the legend.
chart.set_legend({"position": "none"})

# Insert the chart into the worksheet.
worksheet.insert_chart("G3", chart)

workbook.close()
```
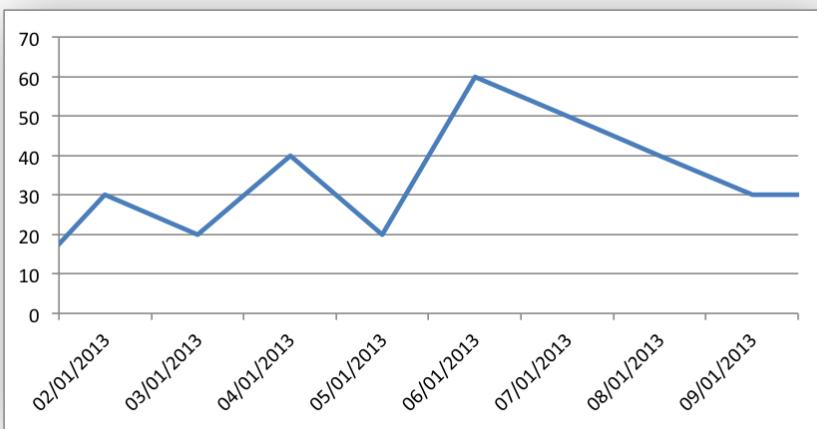
33.19 Example: Date Axis Chart

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.

```python
#############################################################################
#
# An example of creating an Excel charts with a date axis using
# Python and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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),
```

```python
        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()
```

33.20 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:
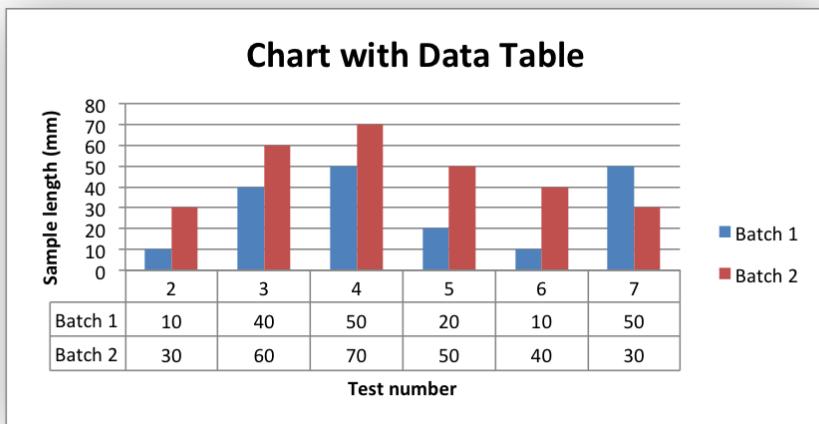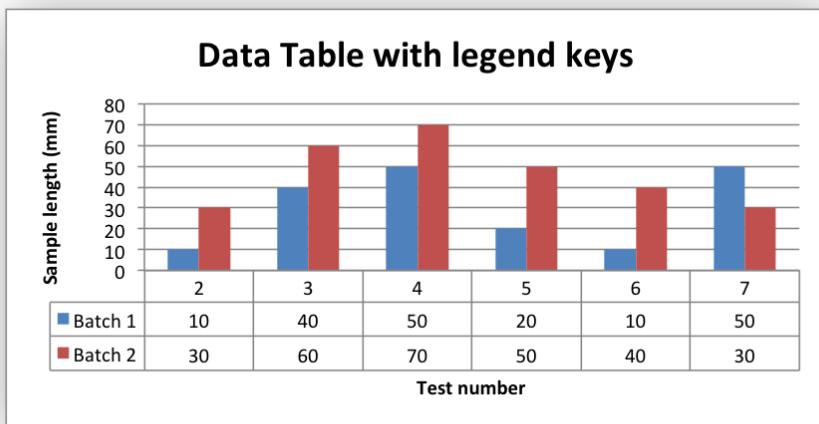
Chart 2 is a column chart with default data table with legend keys:


```
#############################################################################
#
# An example of creating Excel Column charts with data tables using
# Python and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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 = [
```

```python
        [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.
```

```python
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()
```

33.21 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:
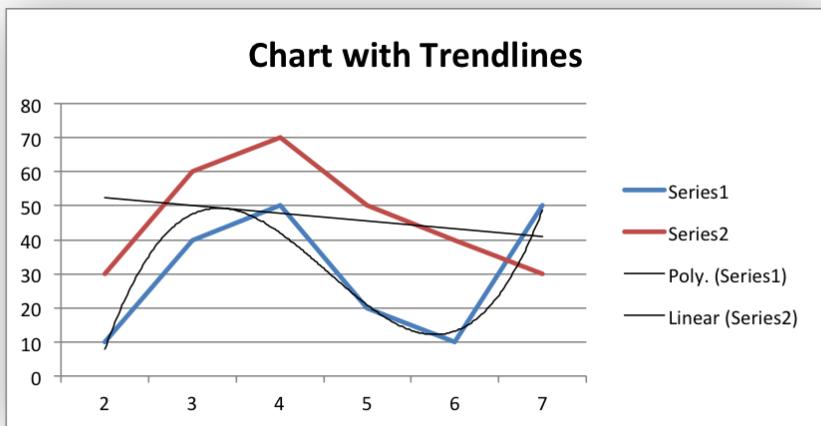
Chart 2 is a chart with data labels and markers:
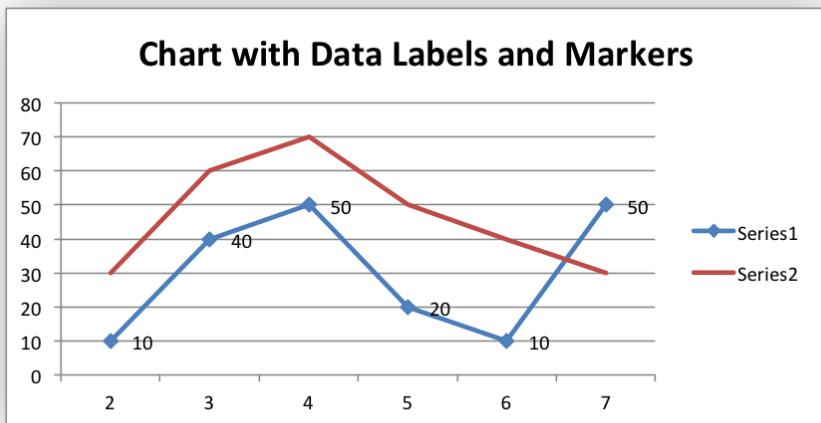
Chart 3 is a chart with error bars:
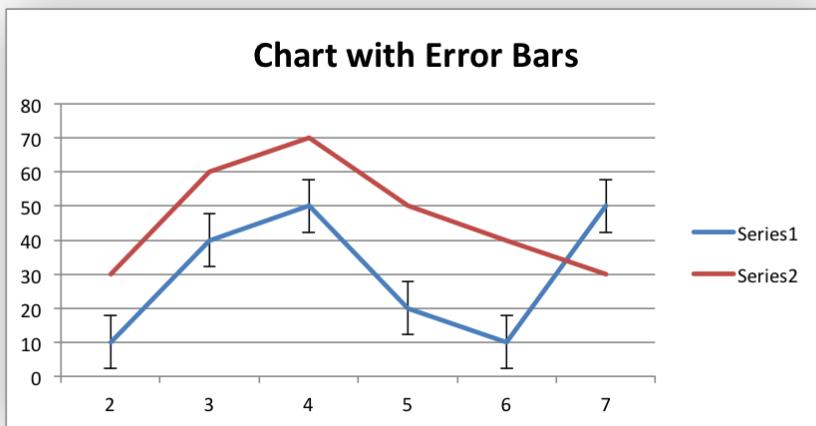
Chart 4 is a chart with up-down bars:
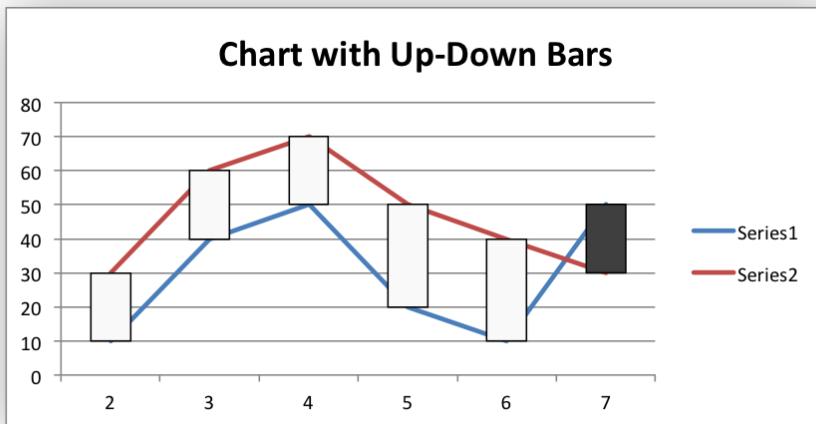
Chart 5 is a chart with hi-low lines:
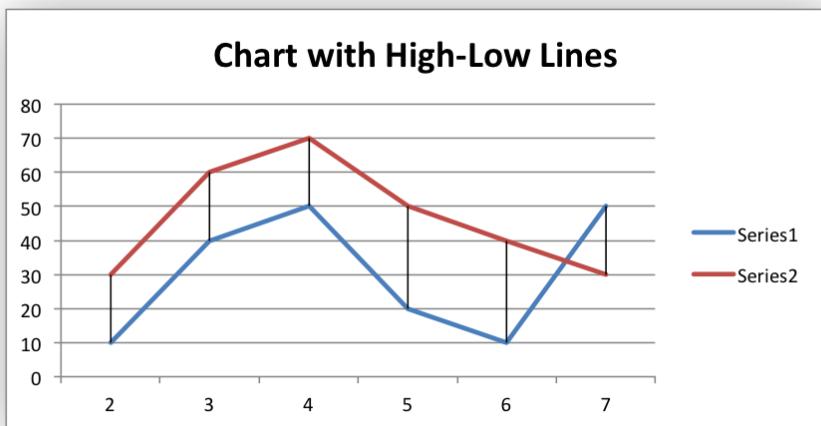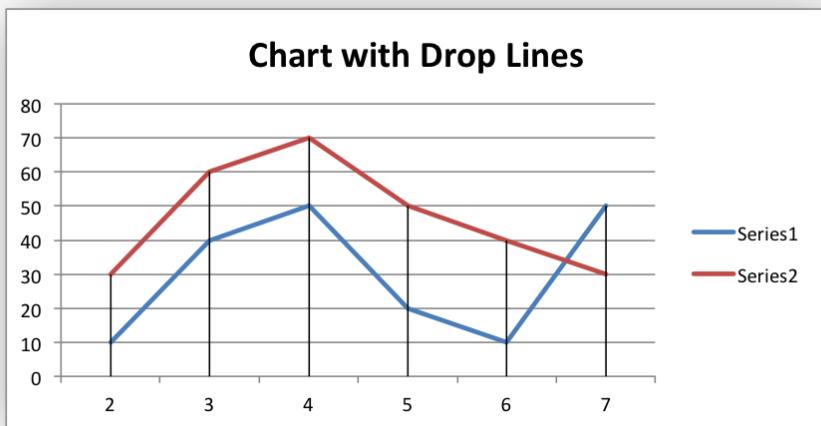
Chart 6 is a chart with drop lines:



```python
###########################################################################
#
# 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook("chart_data_tools.xlsx")
worksheet = workbook.add_worksheet()
bold = workbook.add_format({"bold": 1})
```

```python
# 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.
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.
```

```python
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.
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.
chart3.set_title({"name": "Chart with Error Bars"})
```

```python
    # 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.
    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.
    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.
    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()
```

33.22 Example: Charts with Data Labels

A demo of some of the Excel chart data labels options that are available via an XlsxWriter chart. These include custom labels with user text or text taken from cells in the worksheet. See also

Chart series option: Data Labels and *Chart series option: Custom Data Labels*.

Chart 1 in the following example is a chart with standard data labels:
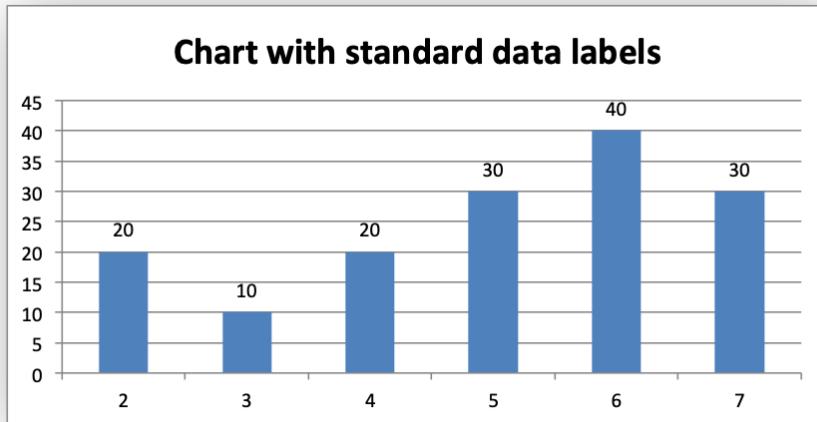
Chart 2 is a chart with Category and Value data labels:
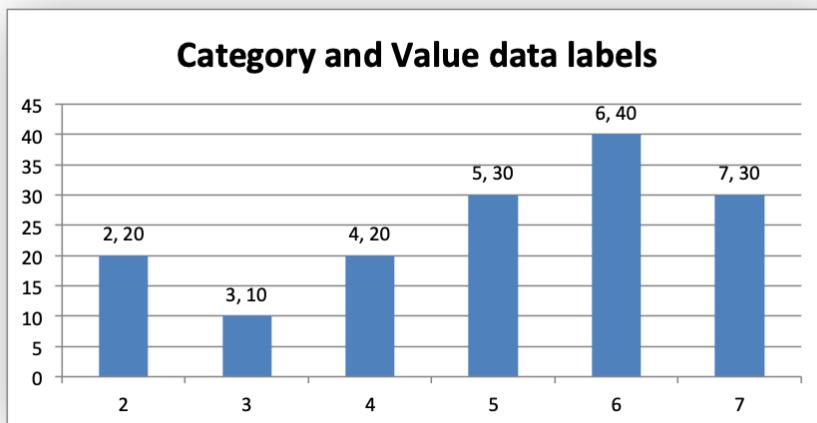
Chart 3 is a chart with data labels with a user defined font:
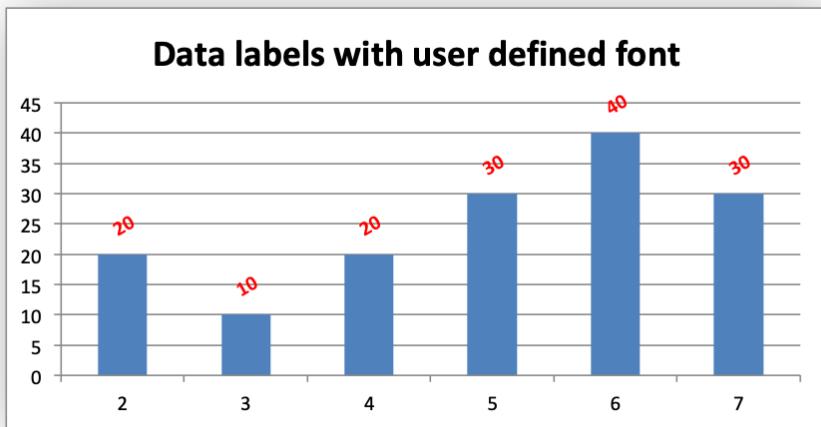
Chart 4 is a chart with standard data labels and formatting:
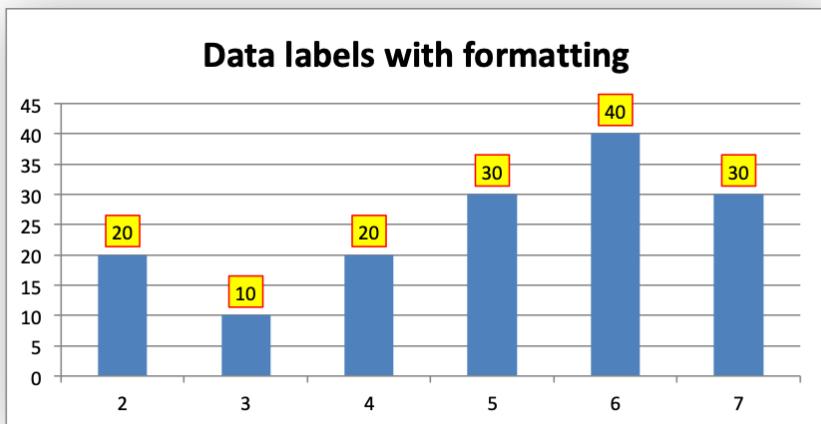
Chart 5 is a chart with custom string data labels:
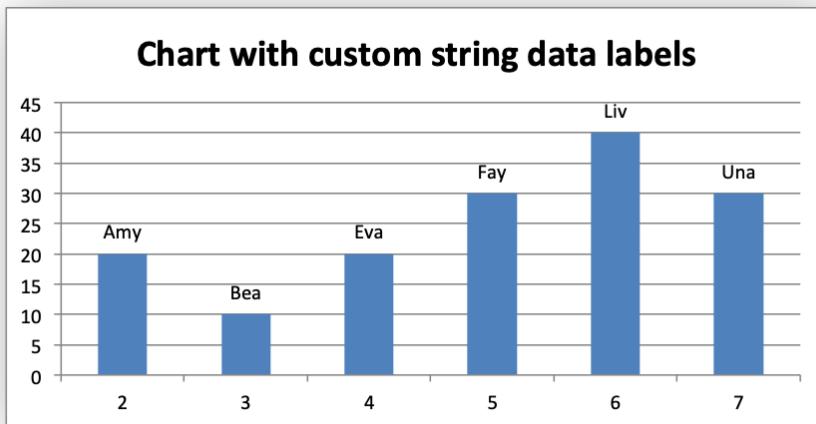
Chart 6 is a chart with custom data labels referenced from worksheet cells:
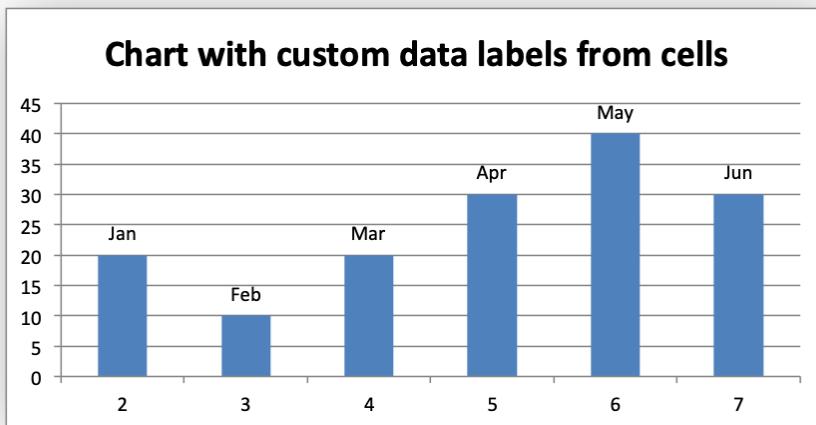
Chart 7 is a chart with a mix of custom and default labels. The None items will get the default value. We also set a font for the custom items as an extra example:
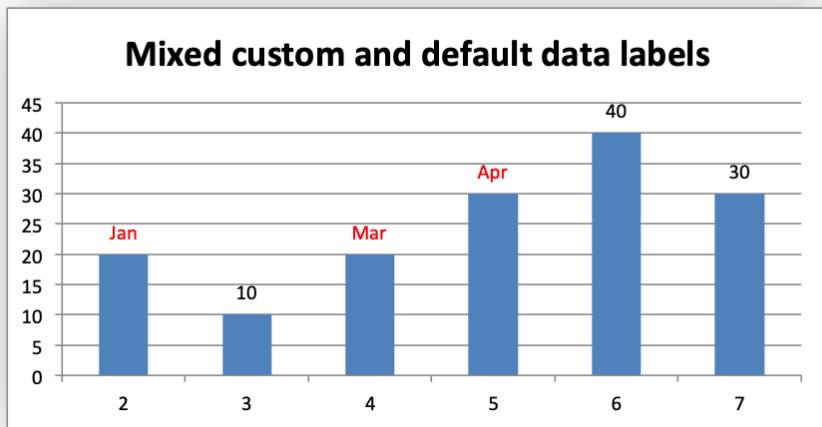
Chart 8 is a chart with some deleted custom labels and defaults (set with None values). This allows us to highlight certain values such as the minimum and maximum:
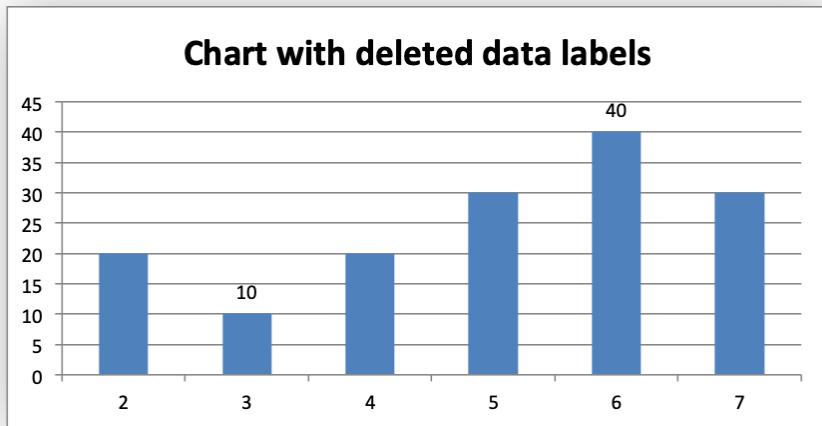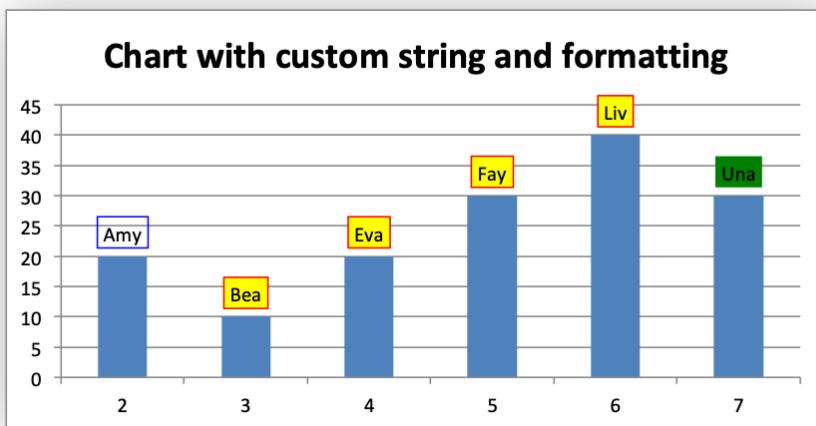
Chart 9 is a chart with custom string data labels and formatting:

```python
###########################################################################
#
# A demo of an various Excel chart data label features that are available
# via an XlsxWriter chart.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
import xlsxwriter

workbook = xlsxwriter.Workbook("chart_data_labels.xlsx")
worksheet = workbook.add_worksheet()
bold = workbook.add_format({"bold": 1})

# Add the worksheet data that the charts will refer to.
headings = ["Number", "Data", "Text"]

data = [
    [2, 3, 4, 5, 6, 7],
    [20, 10, 20, 30, 40, 30],
    ["Jan", "Feb", "Mar", "Apr", "May", "Jun"],
]

worksheet.write_row("A1", headings, bold)
worksheet.write_column("A2", data[0])
worksheet.write_column("B2", data[1])
worksheet.write_column("C2", data[2])

###########################################################################
#
# Example with standard data labels.
#

# Create a Column chart.
chart1 = workbook.add_chart({"type": "column"})
```

```python
    # Configure the data series and add the data labels.
    chart1.add_series(
        {
            "categories": "=Sheet1!$A$2:$A$7",
            "values": "=Sheet1!$B$2:$B$7",
            "data_labels": {"value": True},
        }
    )

    # Add a chart title.
    chart1.set_title({"name": "Chart with standard data labels"})

    # Turn off the chart legend.
    chart1.set_legend({"none": True})

    # Insert the chart into the worksheet (with an offset).
    worksheet.insert_chart("D2", chart1, {"x_offset": 25, "y_offset": 10})

    #######################################################################
    #
    # Example with value and category data labels.
    #

    # Create a Column chart.
    chart2 = workbook.add_chart({"type": "column"})

    # Configure the data series and add the data labels.
    chart2.add_series(
        {
            "categories": "=Sheet1!$A$2:$A$7",
            "values": "=Sheet1!$B$2:$B$7",
            "data_labels": {"value": True, "category": True},
        }
    )

    # Add a chart title.
    chart2.set_title({"name": "Category and Value data labels"})

    # Turn off the chart legend.
    chart2.set_legend({"none": True})

    # Insert the chart into the worksheet (with an offset).
    worksheet.insert_chart("D18", chart2, {"x_offset": 25, "y_offset": 10})

    #######################################################################
    #
    # Example with standard data labels with different font.
    #

    # Create a Column chart.
    chart3 = workbook.add_chart({"type": "column"})

    # Configure the data series and add the data labels.
```

```python
chart3.add_series(
    {
        "categories": "=Sheet1!$A$2:$A$7",
        "values": "=Sheet1!$B$2:$B$7",
        "data_labels": {
            "value": True,
            "font": {"bold": True, "color": "red", "rotation": -30},
        },
    }
)

# Add a chart title.
chart3.set_title({"name": "Data labels with user defined font"})

# Turn off the chart legend.
chart3.set_legend({"none": True})

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart("D34", chart3, {"x_offset": 25, "y_offset": 10})


#######################################################################
#
# Example with standard data labels and formatting.
#

# Create a Column chart.
chart4 = workbook.add_chart({"type": "column"})

# Configure the data series and add the data labels.
chart4.add_series(
    {
        "categories": "=Sheet1!$A$2:$A$7",
        "values": "=Sheet1!$B$2:$B$7",
        "data_labels": {
            "value": True,
            "border": {"color": "red"},
            "fill": {"color": "yellow"},
        },
    }
)

# Add a chart title.
chart4.set_title({"name": "Data labels with formatting"})

# Turn off the chart legend.
chart4.set_legend({"none": True})

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart("D50", chart4, {"x_offset": 25, "y_offset": 10})


#######################################################################
#
# Example with custom string data labels.
```

```python
    #

    # Create a Column chart.
    chart5 = workbook.add_chart({"type": "column"})

    # Some custom labels.
    custom_labels = [
        {"value": "Amy"},
        {"value": "Bea"},
        {"value": "Eva"},
        {"value": "Fay"},
        {"value": "Liv"},
        {"value": "Una"},
    ]

    # Configure the data series and add the data labels.
    chart5.add_series(
        {
            "categories": "=Sheet1!$A$2:$A$7",
            "values": "=Sheet1!$B$2:$B$7",
            "data_labels": {"value": True, "custom": custom_labels},
        }
    )

    # Add a chart title.
    chart5.set_title({"name": "Chart with custom string data labels"})

    # Turn off the chart legend.
    chart5.set_legend({"none": True})

    # Insert the chart into the worksheet (with an offset).
    worksheet.insert_chart("D66", chart5, {"x_offset": 25, "y_offset": 10})

    ######################################################################
    #
    # Example with custom data labels from cells.
    #

    # Create a Column chart.
    chart6 = workbook.add_chart({"type": "column"})

    # Some custom labels.
    custom_labels = [
        {"value": "=Sheet1!$C$2"},
        {"value": "=Sheet1!$C$3"},
        {"value": "=Sheet1!$C$4"},
        {"value": "=Sheet1!$C$5"},
        {"value": "=Sheet1!$C$6"},
        {"value": "=Sheet1!$C$7"},
    ]

    # Configure the data series and add the data labels.
    chart6.add_series(
```

```python
    {
        "categories": "=Sheet1!$A$2:$A$7",
        "values": "=Sheet1!$B$2:$B$7",
        "data_labels": {"value": True, "custom": custom_labels},
    }
)

# Add a chart title.
chart6.set_title({"name": "Chart with custom data labels from cells"})

# Turn off the chart legend.
chart6.set_legend({"none": True})

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart("D82", chart6, {"x_offset": 25, "y_offset": 10})


#######################################################################
#
# Example with custom and default data labels.
#

# Create a Column chart.
chart7 = workbook.add_chart({"type": "column"})

# The following is used to get a mix of default and custom labels. The 'None'
# items will get the default value. We also set a font for the custom items
# as an extra example.
custom_labels = [
    {"value": "=Sheet1!$C$2", "font": {"color": "red"}},
    None,
    {"value": "=Sheet1!$C$4", "font": {"color": "red"}},
    {"value": "=Sheet1!$C$5", "font": {"color": "red"}},
]

# Configure the data series and add the data labels.
chart7.add_series(
    {
        "categories": "=Sheet1!$A$2:$A$7",
        "values": "=Sheet1!$B$2:$B$7",
        "data_labels": {"value": True, "custom": custom_labels},
    }
)

# Add a chart title.
chart7.set_title({"name": "Mixed custom and default data labels"})

# Turn off the chart legend.
chart7.set_legend({"none": True})

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart("D98", chart7, {"x_offset": 25, "y_offset": 10})
```

```python
########################################################################
#
# Example with deleted custom data labels.
#

# Create a Column chart.
chart8 = workbook.add_chart({"type": "column"})

# Some deleted custom labels and defaults (set with None values). This allows
# us to highlight certain values such as the minimum and maximum.
custom_labels = [
    {"delete": True},
    None,
    {"delete": True},
    {"delete": True},
    None,
    {"delete": True},
]

# Configure the data series and add the data labels.
chart8.add_series(
    {
        "categories": "=Sheet1!$A$2:$A$7",
        "values": "=Sheet1!$B$2:$B$7",
        "data_labels": {"value": True, "custom": custom_labels},
    }
)

# Add a chart title.
chart8.set_title({"name": "Chart with deleted data labels"})

# Turn off the chart legend.
chart8.set_legend({"none": True})

# Insert the chart into the worksheet (with an offset).
worksheet.insert_chart("D114", chart8, {"x_offset": 25, "y_offset": 10})

########################################################################
#
# Example with custom string data labels and formatting.
#

# Create a Column chart.
chart9 = workbook.add_chart({"type": "column"})

# Some custom labels.
custom_labels = [
    {"value": "Amy", "border": {"color": "blue"}},
    {"value": "Bea"},
    {"value": "Eva"},
    {"value": "Fay"},
    {"value": "Liv"},
    {"value": "Una", "fill": {"color": "green"}},
```

```
    ]

    # Configure the data series and add the data labels.
    chart9.add_series(
        {
            "categories": "=Sheet1!$A$2:$A$7",
            "values": "=Sheet1!$B$2:$B$7",
            "data_labels": {
                "value": True,
                "custom": custom_labels,
                "border": {"color": "red"},
                "fill": {"color": "yellow"},
            },
        }
    )

    # Add a chart title.
    chart9.set_title({"name": "Chart with custom labels and formatting"})

    # Turn off the chart legend.
    chart9.set_legend({"none": True})

    # Insert the chart into the worksheet (with an offset).
    worksheet.insert_chart("D130", chart9, {"x_offset": 25, "y_offset": 10})

workbook.close()
```
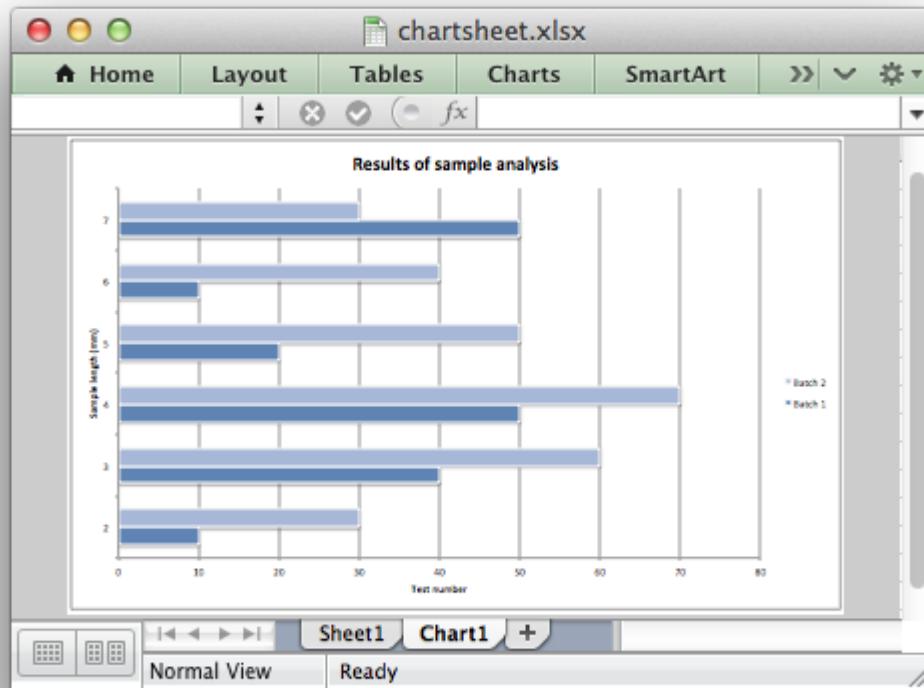
33.23 Example: Chartsheet

Example of creating an Excel Bar chart on a *chartsheet*.

```python
#############################################################################
#
# An example of creating an Excel chart in a chartsheet with Python
# and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, 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)

    # Display the chartsheet as the active sheet when the workbook is opened.
    chartsheet.activate()

    workbook.close()
```
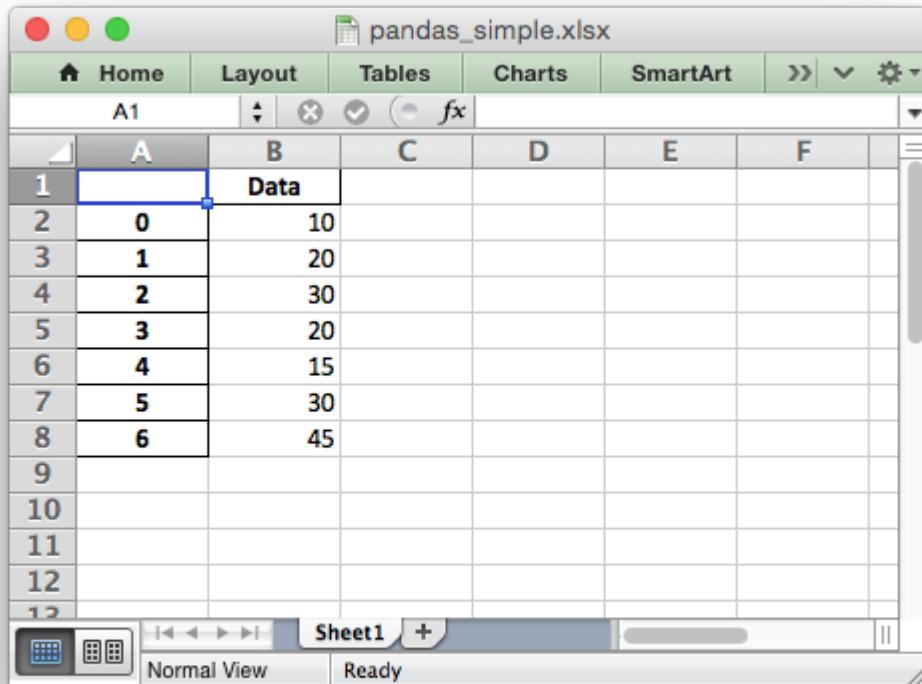
PANDAS WITH XLSXWRITER EXAMPLES

The following are some of the examples included in the examples directory of the XlsxWriter distribution.

They show how to use XlsxWriter with Pandas.

34.1 Example: Pandas Excel example

A simple example of converting a Pandas dataframe to an Excel file using Pandas and XlsxWriter. See *Working with Pandas and XlsxWriter* for more details.
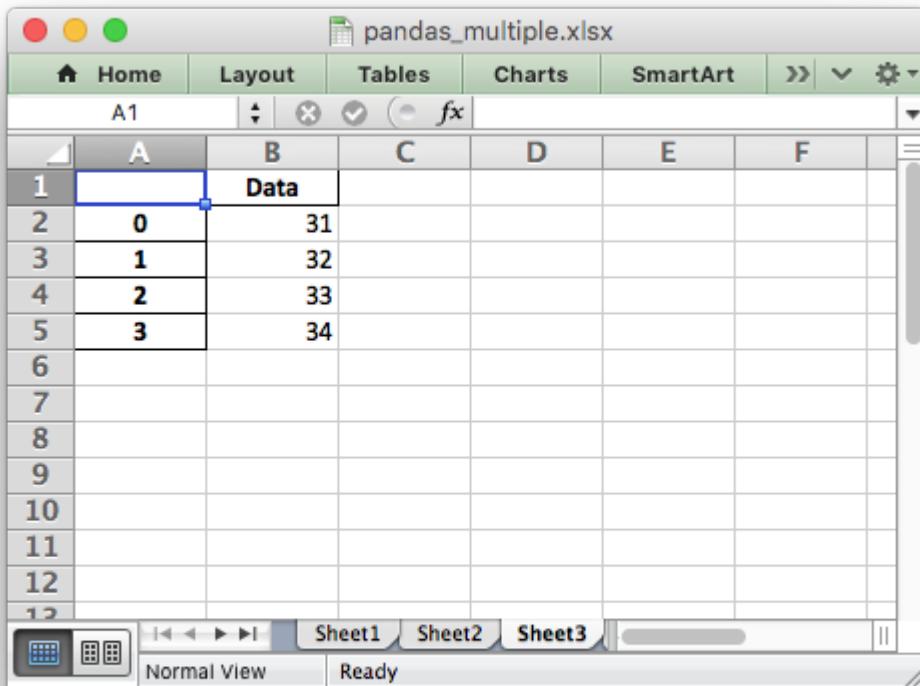
```python
###############################################################################
#
# A simple example of converting a Pandas dataframe to an xlsx file using
# Pandas and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import pandas as pd


# Create a Pandas dataframe from some data.
df = pd.DataFrame({"Data": [10, 20, 30, 20, 15, 30, 45]})

# Create a Pandas Excel writer using XlsxWriter as the engine.
writer = pd.ExcelWriter("pandas_simple.xlsx", engine="xlsxwriter")

# Convert the dataframe to an XlsxWriter Excel object.
df.to_excel(writer, sheet_name="Sheet1")

# Close the Pandas Excel writer and output the Excel file.
writer.close()
```

34.2 Example: Pandas Excel with multiple dataframes

An example of writing multiple dataframes to worksheets using Pandas and XlsxWriter.
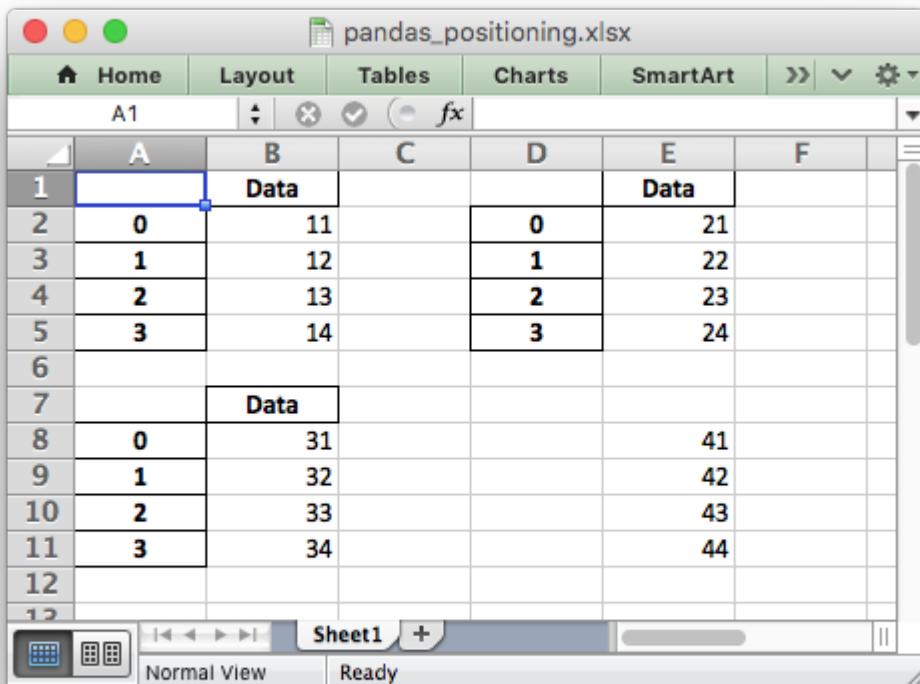
```
###############################################################################
#
# An example of writing multiple dataframes to worksheets using Pandas and
# XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import pandas as pd


# Create some Pandas dataframes from some data.
df1 = pd.DataFrame({"Data": [11, 12, 13, 14]})
df2 = pd.DataFrame({"Data": [21, 22, 23, 24]})
df3 = pd.DataFrame({"Data": [31, 32, 33, 34]})

# Create a Pandas Excel writer using XlsxWriter as the engine.
writer = pd.ExcelWriter("pandas_multiple.xlsx", engine="xlsxwriter")

# Write each dataframe to a different worksheet.
df1.to_excel(writer, sheet_name="Sheet1")
df2.to_excel(writer, sheet_name="Sheet2")
df3.to_excel(writer, sheet_name="Sheet3")
```

```
    # Close the Pandas Excel writer and output the Excel file.
    writer.close()
```

34.3 Example: Pandas Excel dataframe positioning

An example of positioning dataframes in a worksheet using Pandas and XlsxWriter. It also demonstrates how to write a dataframe without the header and index.


```
##############################################################################
#
# An example of positioning dataframes in a worksheet using Pandas and
# XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import pandas as pd


# Create some Pandas dataframes from some data.
df1 = pd.DataFrame({"Data": [11, 12, 13, 14]})
df2 = pd.DataFrame({"Data": [21, 22, 23, 24]})
```
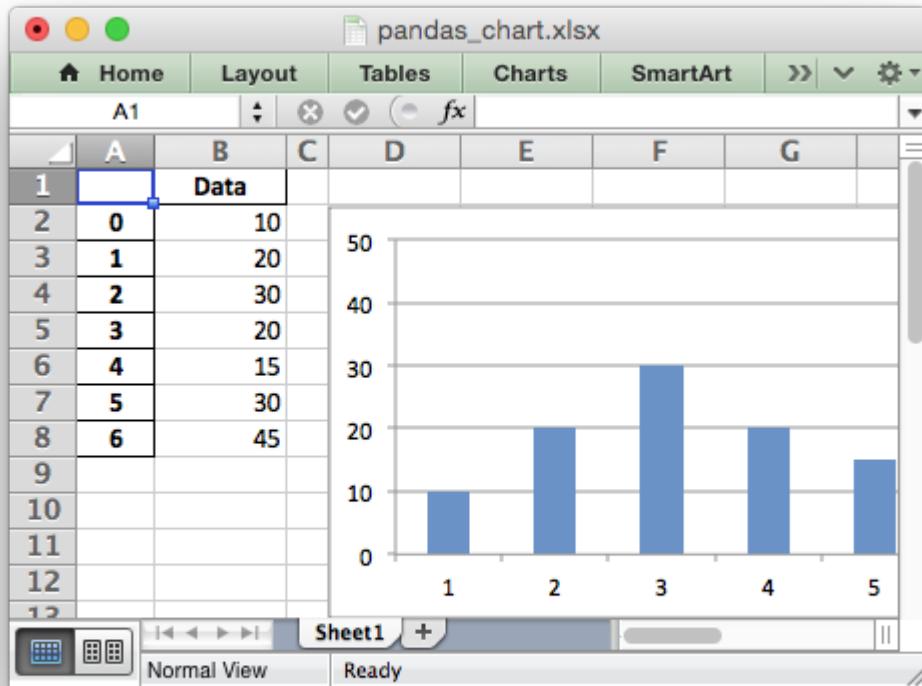
```
    df3 = pd.DataFrame({"Data": [31, 32, 33, 34]})
    df4 = pd.DataFrame({"Data": [41, 42, 43, 44]})

    # Create a Pandas Excel writer using XlsxWriter as the engine.
    writer = pd.ExcelWriter("pandas_positioning.xlsx", engine="xlsxwriter")

    # Position the dataframes in the worksheet.
    df1.to_excel(writer, sheet_name="Sheet1")  # Default position, cell A1.
    df2.to_excel(writer, sheet_name="Sheet1", startcol=3)
    df3.to_excel(writer, sheet_name="Sheet1", startrow=6)

    # It is also possible to write the dataframe without the header and index.
    df4.to_excel(
        writer, sheet_name="Sheet1", startrow=7, startcol=4, header=False, index=False
    )

    # Close the Pandas Excel writer and output the Excel file.
    writer.close()
```

34.4 Example: Pandas Excel output with a chart

A simple example of converting a Pandas dataframe to an Excel file with a chart using Pandas and XlsxWriter.

```
###############################################################################
#
# An example of converting a Pandas dataframe to an xlsx file with a chart
# using Pandas and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import pandas as pd


# Create a Pandas dataframe from some data.
df = pd.DataFrame({"Data": [10, 20, 30, 20, 15, 30, 45]})

# Create a Pandas Excel writer using XlsxWriter as the engine.
writer = pd.ExcelWriter("pandas_chart.xlsx", engine="xlsxwriter")

# Convert the dataframe to an XlsxWriter Excel object.
df.to_excel(writer, sheet_name="Sheet1")

# Get the xlsxwriter workbook and worksheet objects.
workbook = writer.book
```
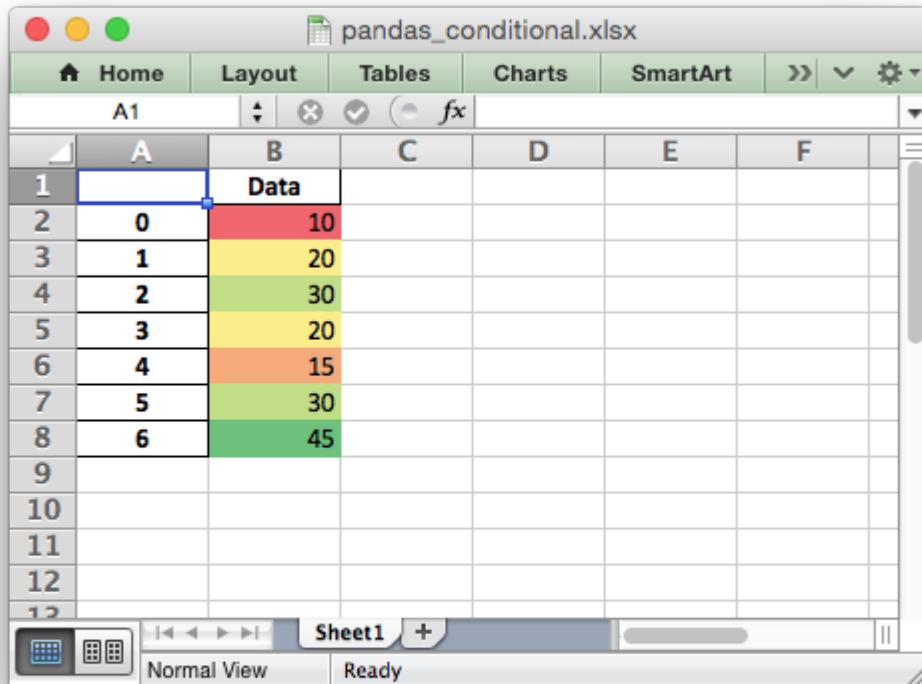
```
worksheet = writer.sheets["Sheet1"]

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

# Get the dimensions of the dataframe.
(max_row, max_col) = df.shape

# Configure the series of the chart from the dataframe data.
chart.add_series({"values": ["Sheet1", 1, 1, max_row, 1]})

# Insert the chart into the worksheet.
worksheet.insert_chart(1, 3, chart)

# Close the Pandas Excel writer and output the Excel file.
writer.close()
```

34.5 Example: Pandas Excel output with conditional formatting

An example of converting a Pandas dataframe to an Excel file with a conditional formatting using Pandas and XlsxWriter.

```
##########################################################################
#
# An example of converting a Pandas dataframe to an xlsx file with a
# conditional formatting using Pandas and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import pandas as pd


# Create a Pandas dataframe from some data.
df = pd.DataFrame({"Data": [10, 20, 30, 20, 15, 30, 45]})

# Create a Pandas Excel writer using XlsxWriter as the engine.
writer = pd.ExcelWriter("pandas_conditional.xlsx", engine="xlsxwriter")

# Convert the dataframe to an XlsxWriter Excel object.
df.to_excel(writer, sheet_name="Sheet1")

# Get the xlsxwriter workbook and worksheet objects.
workbook = writer.book
worksheet = writer.sheets["Sheet1"]

# Get the dimensions of the dataframe.
(max_row, max_col) = df.shape

# Apply a conditional format to the required cell range.
worksheet.conditional_format(1, max_col, max_row, max_col, {"type": "3_color_scale"})

# Close the Pandas Excel writer and output the Excel file.
writer.close()
```

34.6 Example: Pandas Excel output with an autofilter

An example of converting a Pandas dataframe to an Excel file with a autofilter, and filtered data, using Pandas and XlsxWriter. See *Working with Autofilters* for a more detailed explanation of autofilters.

```
################################################################################
#
# An example of converting a Pandas dataframe to an xlsx file with an
# autofilter and filtered data. See also autofilter.py.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import pandas as pd

# Create a Pandas dataframe by reading some data from a space-separated file.
df = pd.read_csv("autofilter_data.txt", sep=r"\s+")

# Create a Pandas Excel writer using XlsxWriter as the engine.
writer = pd.ExcelWriter("pandas_autofilter.xlsx", engine="xlsxwriter")

# Convert the dataframe to an XlsxWriter Excel object. We also turn off the
# index column at the left of the output dataframe.
df.to_excel(writer, sheet_name="Sheet1", index=False)

# Get the xlsxwriter workbook and worksheet objects.
workbook = writer.book
worksheet = writer.sheets["Sheet1"]
```

```python
    # Get the dimensions of the dataframe.
    (max_row, max_col) = df.shape

    # Make the columns wider for clarity.
    worksheet.set_column(0, max_col - 1, 12)

    # Set the autofilter.
    worksheet.autofilter(0, 0, max_row, max_col - 1)

    # Add an optional 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")

    # It isn't enough to just apply the criteria. The rows that don't match
    # must also be hidden. We use Pandas to figure our which rows to hide.
    for row_num in df.index[(df["Region"] != "East")].tolist():
        worksheet.set_row(row_num + 1, options={"hidden": True})

    # Close the Pandas Excel writer and output the Excel file.
    writer.close()
```

34.7 Example: Pandas Excel output with a worksheet table

An example of inserting a Pandas dataframe into an Excel worksheet table file using Pandas and XlsxWriter.

```
########################################################################
#
# An example of adding a dataframe to an worksheet table in an xlsx file
# using Pandas and XlsxWriter.
#
# 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.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import pandas as pd

# Create a Pandas dataframe from some data.
df = pd.DataFrame(
    {
        "Country": ["China", "India", "United States", "Indonesia"],
        "Population": [1404338840, 1366938189, 330267887, 269603400],
        "Rank": [1, 2, 3, 4],
    }
)

# Order the columns if necessary.
df = df[["Rank", "Country", "Population"]]
```

```
    # Create a Pandas Excel writer using XlsxWriter as the engine.
    writer = pd.ExcelWriter("pandas_table.xlsx", engine="xlsxwriter")

    # Write the dataframe data to XlsxWriter. Turn off the default header and
    # index and skip one row to allow us to insert a user defined header.
    df.to_excel(writer, sheet_name="Sheet1", startrow=1, header=False, index=False)

    # Get the xlsxwriter workbook and worksheet objects.
    workbook = writer.book
    worksheet = writer.sheets["Sheet1"]

    # Get the dimensions of the dataframe.
    (max_row, max_col) = df.shape

    # Create a list of column headers, to use in add_table().
    column_settings = [{"header": column} for column in df.columns]

    # Add the Excel table structure. Pandas will add the data.
    worksheet.add_table(0, 0, max_row, max_col - 1, {"columns": column_settings})

    # Make the columns wider for clarity.
    worksheet.set_column(0, max_col - 1, 12)

    # Close the Pandas Excel writer and output the Excel file.
    writer.close()
```

34.8 Example: Pandas Excel output with datetimes

An example of converting a Pandas dataframe with datetimes to an Excel file with a default date-
time and date format using Pandas and XlsxWriter.

```
##############################################################################
#
# An example of converting a Pandas dataframe with datetimes to an xlsx file
# with a default datetime and date format using Pandas and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import pandas as pd
from datetime import datetime, date

# Create a Pandas dataframe from some datetime data.
df = pd.DataFrame(
    {
        "Date and time": [
            datetime(2015, 1, 1, 11, 30, 55),
            datetime(2015, 1, 2, 1, 20, 33),
            datetime(2015, 1, 3, 11, 10),
            datetime(2015, 1, 4, 16, 45, 35),
            datetime(2015, 1, 5, 12, 10, 15),
        ],
        "Dates only": [
```

```
                date(2015, 2, 1),
                date(2015, 2, 2),
                date(2015, 2, 3),
                date(2015, 2, 4),
                date(2015, 2, 5),
        ],
    }
)

# Create a Pandas Excel writer using XlsxWriter as the engine.
# Also set the default datetime and date formats.
writer = pd.ExcelWriter(
    "pandas_datetime.xlsx",
    engine="xlsxwriter",
    datetime_format="mmm d yyyy hh:mm:ss",
    date_format="mmmm dd yyyy",
)

# Convert the dataframe to an XlsxWriter Excel object.
df.to_excel(writer, sheet_name="Sheet1")

# Get the xlsxwriter workbook and worksheet objects in order
# to set the column widths and make the dates clearer.
workbook = writer.book
worksheet = writer.sheets["Sheet1"]

# Get the dimensions of the dataframe.
(max_row, max_col) = df.shape

# Set the column widths, to make the dates clearer.
worksheet.set_column(1, max_col, 20)

# Close the Pandas Excel writer and output the Excel file.
writer.close()
```

34.9 Example: Pandas Excel output with column formatting

An example of converting a Pandas dataframe to an Excel file with column formats using Pandas and XlsxWriter.

It isn't possible to format any cells that already have a format such as the index or headers or any cells that contain dates or datetimes.

Note: This feature requires Pandas >= 0.16.

```
############################################################################
#
# An example of converting a Pandas dataframe to an xlsx file
# with column formats using Pandas and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import pandas as pd

# Create a Pandas dataframe from some data.
df = pd.DataFrame(
    {
        "Numbers": [1010, 2020, 3030, 2020, 1515, 3030, 4545],
        "Percentage": [0.1, 0.2, 0.33, 0.25, 0.5, 0.75, 0.45],
    }
)

# Create a Pandas Excel writer using XlsxWriter as the engine.
writer = pd.ExcelWriter("pandas_column_formats.xlsx", engine="xlsxwriter")

# Convert the dataframe to an XlsxWriter Excel object.
```

```
    df.to_excel(writer, sheet_name="Sheet1")

    # Get the xlsxwriter workbook and worksheet objects.
    workbook = writer.book
    worksheet = writer.sheets["Sheet1"]

    # Add some cell formats.
    format1 = workbook.add_format({"num_format": "#,##0.00"})
    format2 = workbook.add_format({"num_format": "0%"})

    # Note: It isn't possible to format any cells that already have a format such
    # as the index or headers or any cells that contain dates or datetimes.

    # Set the column width and format.
    worksheet.set_column(1, 1, 18, format1)

    # Set the format but not the column width.
    worksheet.set_column(2, 2, None, format2)

    # Close the Pandas Excel writer and output the Excel file.
    writer.close()
```
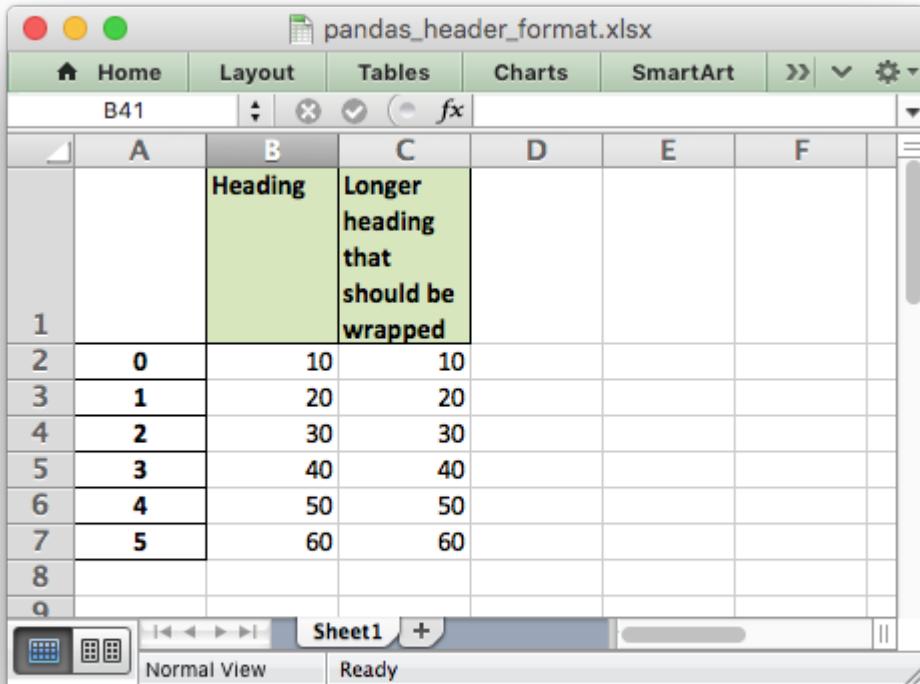
34.10 Example: Pandas Excel output with user defined header format

An example of converting a Pandas dataframe to an Excel file with a user defined header format using Pandas and XlsxWriter.

```
###############################################################################
#
# An example of converting a Pandas dataframe to an xlsx file
# with a user defined header format.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import pandas as pd

# Create a Pandas dataframe from some data.
data = [10, 20, 30, 40, 50, 60]
df = pd.DataFrame({"Heading": data, "Longer heading that should be wrapped": data})

# Create a Pandas Excel writer using XlsxWriter as the engine.
writer = pd.ExcelWriter("pandas_header_format.xlsx", engine="xlsxwriter")

# Convert the dataframe to an XlsxWriter Excel object. Note that we turn off
# the default header and skip one row to allow us to insert a user defined
# header.
df.to_excel(writer, sheet_name="Sheet1", startrow=1, header=False)

# Get the xlsxwriter workbook and worksheet objects.
```
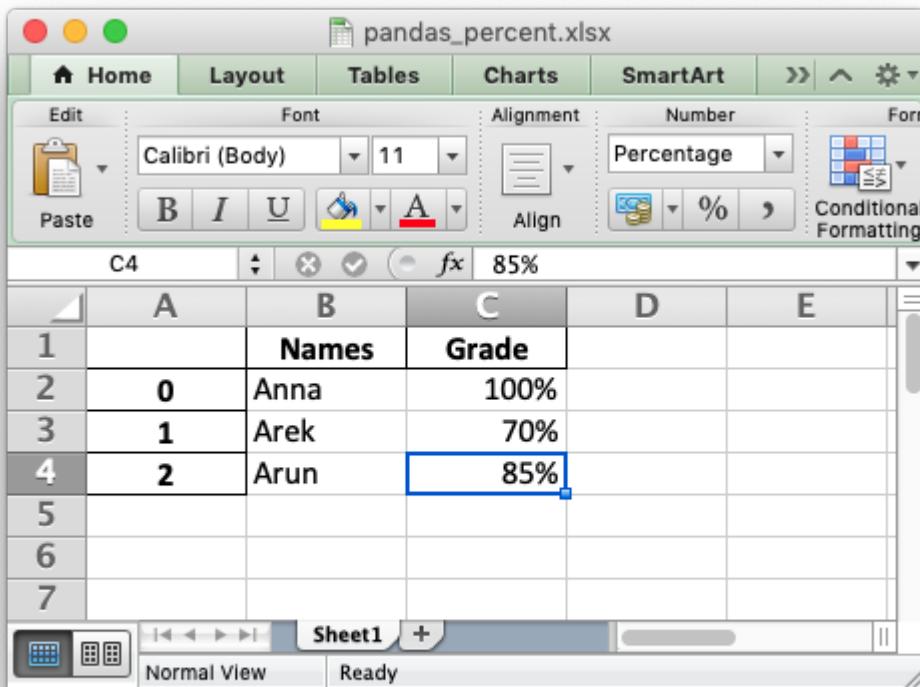
```
    workbook = writer.book
    worksheet = writer.sheets["Sheet1"]

    # Add a header format.
    header_format = workbook.add_format(
        {
            "bold": True,
            "text_wrap": True,
            "valign": "top",
            "fg_color": "#D7E4BC",
            "border": 1,
        }
    )

    # Write the column headers with the defined format.
    for col_num, value in enumerate(df.columns.values):
        worksheet.write(0, col_num + 1, value, header_format)

    # Close the Pandas Excel writer and output the Excel file.
    writer.close()
```

34.11 Example: Pandas Excel output with percentage formatting

To create a percentage in Excel the data must be a number, must be divided by 100 and must have a percentage number format applied.

Here is a simple example of converting some string percentage data in a Pandas dataframe to percentage numbers in an xlsx file using XlsxWriter as the Pandas excel engine:

```
###############################################################################
#
# An example of converting some string percentage data in a Pandas dataframe
# to percentage numbers in an xlsx file with using XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import pandas as pd


# Create a Pandas dataframe from some data.
df = pd.DataFrame({"Names": ["Anna", "Arek", "Arun"], "Grade": ["100%", "70%", "85%"]}

# Convert the percentage strings to percentage numbers.
df["Grade"] = df["Grade"].str.replace("%", "")
df["Grade"] = df["Grade"].astype(float)
df["Grade"] = df["Grade"].div(100)

# Create a Pandas Excel writer using XlsxWriter as the engine.
writer = pd.ExcelWriter("pandas_percent.xlsx", engine="xlsxwriter")

# Convert the dataframe to an XlsxWriter Excel object.
df.to_excel(writer, sheet_name="Sheet1")
```
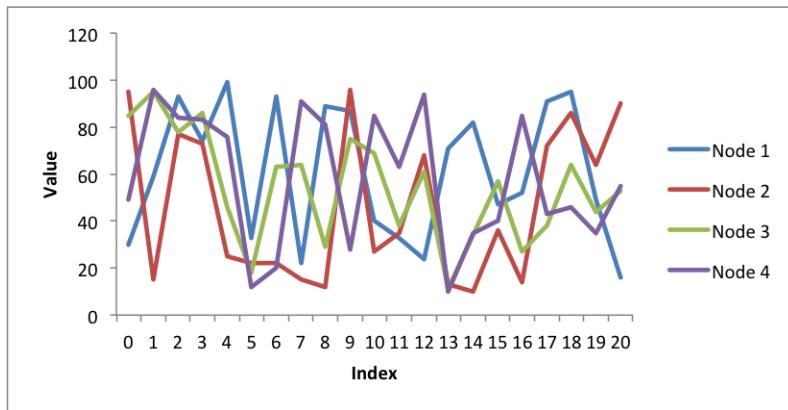
```python
    # Get the xlsxwriter workbook and worksheet objects.
    workbook = writer.book
    worksheet = writer.sheets["Sheet1"]

    # Add a percent number format.
    percent_format = workbook.add_format({"num_format": "0%"})

    # Apply the number format to Grade column.
    worksheet.set_column(2, 2, None, percent_format)

    # Close the Pandas Excel writer and output the Excel file.
    writer.close()
```

34.12 Example: Pandas Excel output with a line chart

A simple example of converting a Pandas dataframe to an Excel file with a line chart using Pandas and XlsxWriter.


```python
    ##############################################################################
    #
    # An example of converting a Pandas dataframe to an xlsx file with a line
    # chart using Pandas and XlsxWriter.
    #
    # SPDX-License-Identifier: BSD-2-Clause
    # Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
    #

    import pandas as pd
    import random

    # Create some sample data to plot.
    categories = ["Node 1", "Node 2", "Node 3", "Node 4"]
    index_1 = range(0, 21, 1)
    multi_iter1 = {"index": index_1}

    for category in categories:
        multi_iter1[category] = [random.randint(10, 100) for x in index_1]
```

```python
    # Create a Pandas dataframe from the data.
    index_2 = multi_iter1.pop("index")
    df = pd.DataFrame(multi_iter1, index=index_2)
    df = df.reindex(columns=sorted(df.columns))

    # Create a Pandas Excel writer using XlsxWriter as the engine.
    sheet_name = "Sheet1"
    writer = pd.ExcelWriter("pandas_chart_line.xlsx", engine="xlsxwriter")
    df.to_excel(writer, sheet_name=sheet_name)

    # Access the XlsxWriter workbook and worksheet objects from the dataframe.
    workbook = writer.book
    worksheet = writer.sheets[sheet_name]

    # Get the dimensions of the dataframe.
    (max_row, max_col) = df.shape

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

    # Configure the series of the chart from the dataframe data.
    for i in range(len(categories)):
        col = i + 1
        chart.add_series(
            {
                "name": ["Sheet1", 0, col],
                "categories": ["Sheet1", 1, 0, max_row, 0],
                "values": ["Sheet1", 1, col, max_row, col],
            }
        )

    # Configure the chart axes.
    chart.set_x_axis({"name": "Index"})
    chart.set_y_axis({"name": "Value", "major_gridlines": {"visible": False}})

    # Insert the chart into the worksheet.
    worksheet.insert_chart(1, 6, chart)

    # Close the Pandas Excel writer and output the Excel file.
    writer.close()
```
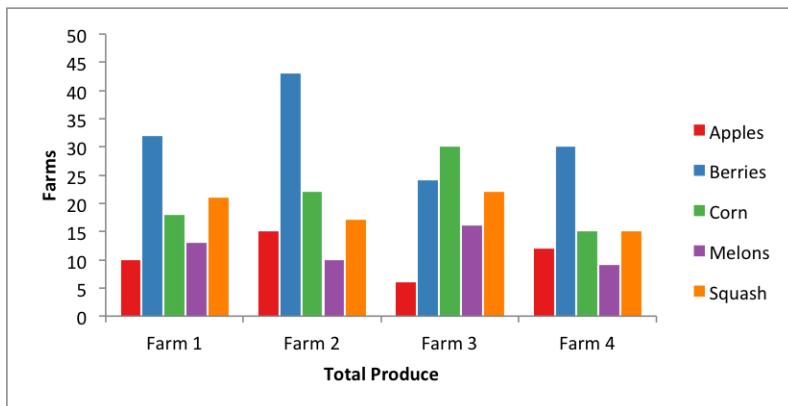
34.13 Example: Pandas Excel output with a column chart

An example of converting a Pandas dataframe to an Excel file with a column chart using Pandas and XlsxWriter.

```
##########################################################################
#
# An example of converting a Pandas dataframe to an xlsx file with a grouped
# column chart using Pandas and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import pandas as pd

# Some sample data to plot.
farm_1 = {"Apples": 10, "Berries": 32, "Squash": 21, "Melons": 13, "Corn": 18}
farm_2 = {"Apples": 15, "Berries": 43, "Squash": 17, "Melons": 10, "Corn": 22}
farm_3 = {"Apples": 6, "Berries": 24, "Squash": 22, "Melons": 16, "Corn": 30}
farm_4 = {"Apples": 12, "Berries": 30, "Squash": 15, "Melons": 9, "Corn": 15}

data = [farm_1, farm_2, farm_3, farm_4]
index = ["Farm 1", "Farm 2", "Farm 3", "Farm 4"]

# Create a Pandas dataframe from the data.
df = pd.DataFrame(data, index=index)

# Create a Pandas Excel writer using XlsxWriter as the engine.
sheet_name = "Sheet1"
writer = pd.ExcelWriter("pandas_chart_columns.xlsx", engine="xlsxwriter")
df.to_excel(writer, sheet_name=sheet_name)

# Access the XlsxWriter workbook and worksheet objects from the dataframe.
workbook = writer.book
worksheet = writer.sheets[sheet_name]

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

# Some alternative colors for the chart.
colors = ["#E41A1C", "#377EB8", "#4DAF4A", "#984EA3", "#FF7F00"]

# Configure the series of the chart from the dataframe data.
for col_num in range(1, len(farm_1) + 1):
```

```python
    chart.add_series(
        {
            "name": ["Sheet1", 0, col_num],
            "categories": ["Sheet1", 1, 0, 4, 0],
            "values": ["Sheet1", 1, col_num, 4, col_num],
            "fill": {"color": colors[col_num - 1]},
            "overlap": -10,
        }
    )

# Configure the chart axes.
chart.set_x_axis({"name": "Total Produce"})
chart.set_y_axis({"name": "Farms", "major_gridlines": {"visible": False}})

# Insert the chart into the worksheet.
worksheet.insert_chart(1, 5, chart)

# Close the Pandas Excel writer and output the Excel file.
writer.close()
```
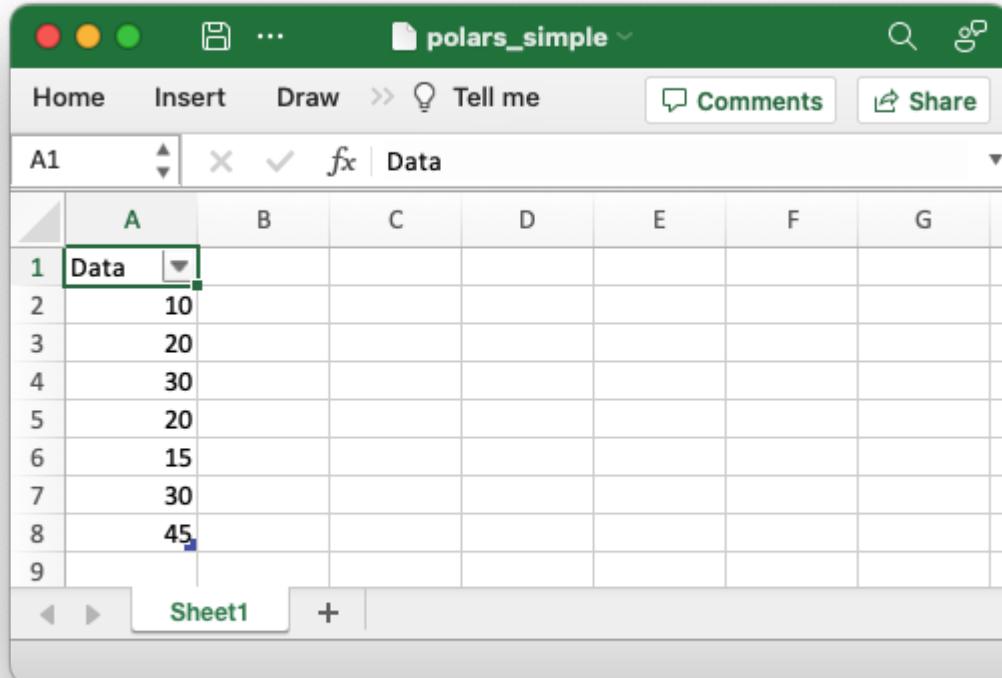
POLARS WITH XLSXWRITER EXAMPLES

The following are some of the examples included in the examples directory of the XlsxWriter distribution.

They show how to use XlsxWriter with Polars .

35.1 Example: Polars Excel getting started example

A simple getting started example of converting a Polars dataframe to an Excel file using Polars and XlsxWriter. See *Working with Polars and XlsxWriter* for more details.
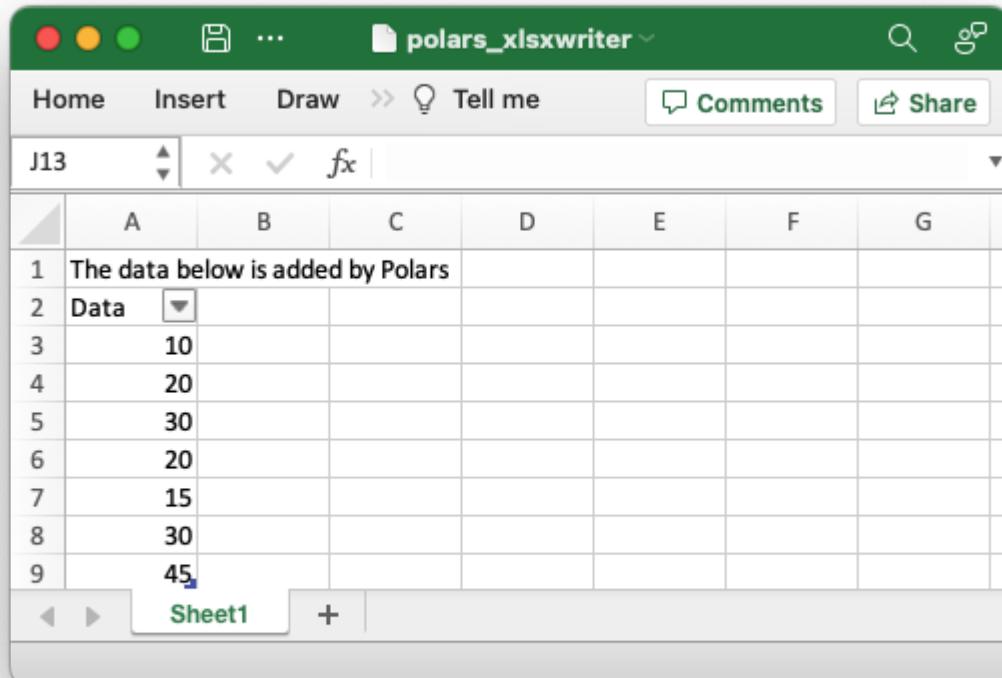
```
##############################################################################
#
# A simple example of converting a Polars dataframe to an xlsx file using
# Polars and XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import polars as pl

# Create a Pandas dataframe from some data.
df = pl.DataFrame({"Data": [10, 20, 30, 20, 15, 30, 45]})

# Write the dataframe to a new Excel file.
df.write_excel(workbook="polars_simple.xlsx")
```

35.2 Example: Polars integration with XlsxWriter

An example of adding a Polars dataframe to a worksheet created by XlsxWriter. See *Working with Polars and XlsxWriter* for more details.

```python
##############################################################################
#
# An example of adding a Polars dataframe to a worksheet created by XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import xlsxwriter
import polars as pl

with xlsxwriter.Workbook("polars_xlsxwriter.xlsx") as workbook:
    # Create a new worksheet.
    worksheet = workbook.add_worksheet()

    # Do something with the worksheet.
    worksheet.write("A1", "The data below is added by Polars")

    df = pl.DataFrame({"Data": [10, 20, 30, 20, 15, 30, 45]})

    # Write the Polars data to the worksheet created above, at an offset to
    # avoid overwriting the previous text.
    df.write_excel(workbook=workbook, worksheet="Sheet1", position="A2")
```
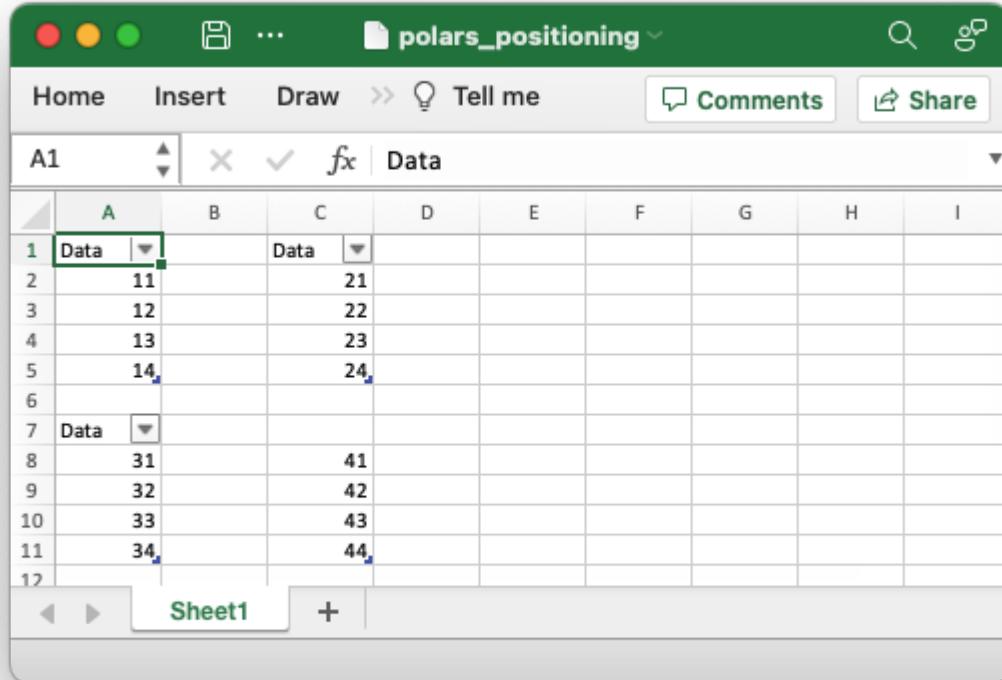
35.3 Example: Polars Excel dataframe positioning

An example of positioning dataframes in a worksheet using Polars and XlsxWriter. It also demonstrates how to write a dataframe without a header. See *Working with Polars and XlsxWriter* for more details.

```
##############################################################################
#
# An example of positioning dataframes in a worksheet using Polars and
# XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import xlsxwriter
import polars as pl

# Create some Polars dataframes from some data.
df1 = pl.DataFrame({"Data": [11, 12, 13, 14]})
df2 = pl.DataFrame({"Data": [21, 22, 23, 24]})
df3 = pl.DataFrame({"Data": [31, 32, 33, 34]})
df4 = pl.DataFrame({"Data": [41, 42, 43, 44]})

with xlsxwriter.Workbook("polars_positioning.xlsx") as workbook:
    # Write the dataframe to the default worksheet and position: Sheet1!A1.
    df1.write_excel(workbook=workbook)

    # Write the dataframe using a cell string position.
    df2.write_excel(workbook=workbook, worksheet="Sheet1", position="C1")
```
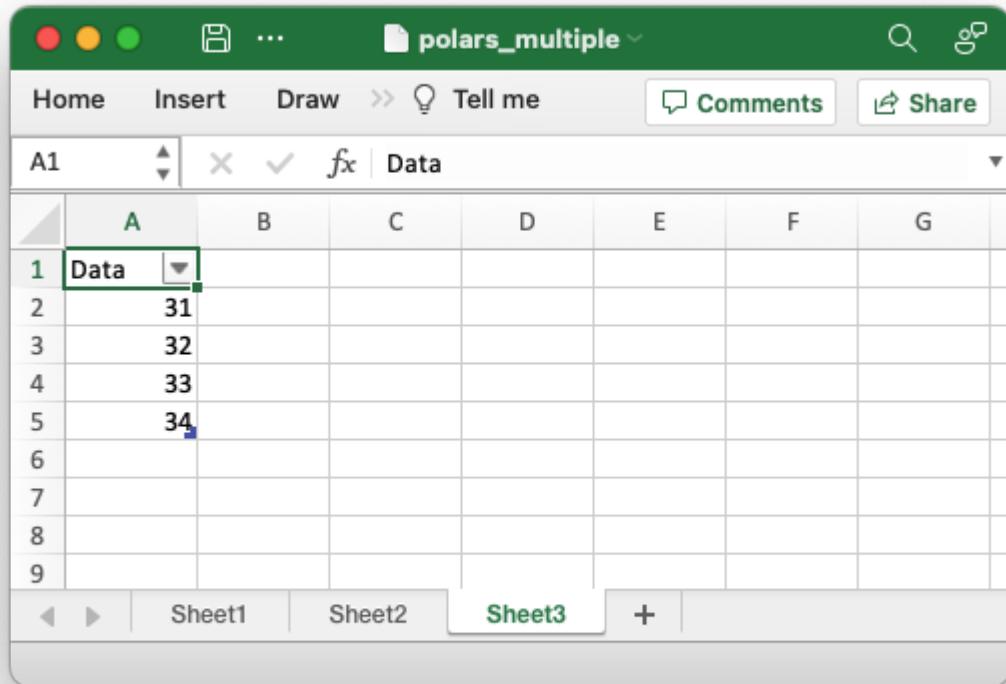
```python
# Write the dataframe using a (row, col) tuple position.
df3.write_excel(workbook=workbook, worksheet="Sheet1", position=(6, 0))

# Write the dataframe without the header.
df4.write_excel(
    workbook=workbook, worksheet="Sheet1", position="C8", has_header=False
)
```

35.4 Example: Polars Excel with multiple dataframes

An example of writing multiple dataframes to worksheets using Polars and XlsxWriter. See *Working with Polars and XlsxWriter* for more details.


```python
########################################################################
#
# An example of writing multiple dataframes to worksheets using Polars and
# XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#
```
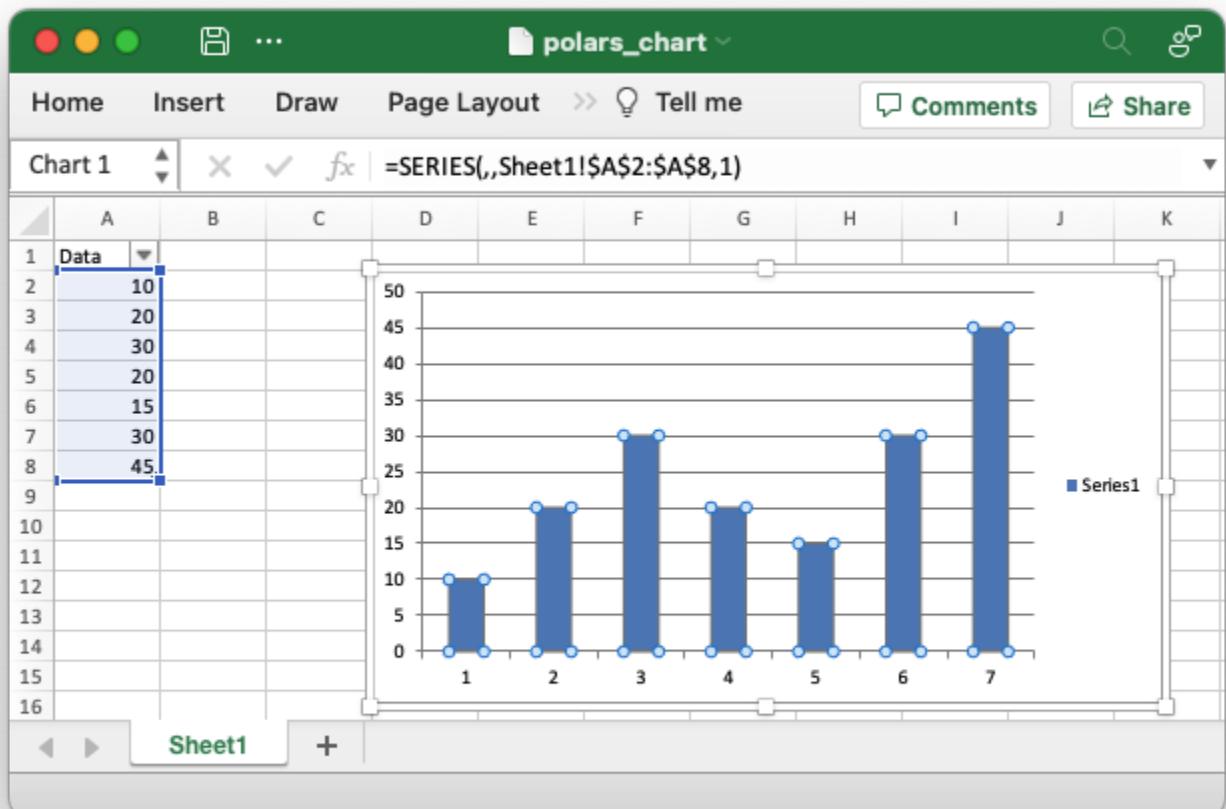
```python
import xlsxwriter
import polars as pl

# Create some Polars dataframes from some data.
df1 = pl.DataFrame({"Data": [11, 12, 13, 14]})
df2 = pl.DataFrame({"Data": [21, 22, 23, 24]})
df3 = pl.DataFrame({"Data": [31, 32, 33, 34]})

with xlsxwriter.Workbook("polars_multiple.xlsx") as workbook:
    df1.write_excel(workbook=workbook)
    df2.write_excel(workbook=workbook)
    df3.write_excel(workbook=workbook)
```

35.5 Example: Polars Excel output with a chart

An example of adding a Polars dataframe to a worksheet created by XlsxWriter and then adding a chart of the data. See *Working with Polars and XlsxWriter* for more details.

```
##############################################################################
#
# An example of adding a Polars dataframe to a worksheet created by XlsxWriter
# and then adding a chart of the data.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import xlsxwriter
import polars as pl

df = pl.DataFrame({"Data": [10, 20, 30, 20, 15, 30, 45]})

with xlsxwriter.Workbook("polars_chart.xlsx") as workbook:
    # Create the worksheet so we can reuse it later.
    worksheet = workbook.add_worksheet()

    # Write the Polars data to the worksheet created above.
    df.write_excel(workbook=workbook, worksheet="Sheet1")

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

    # Get the dimensions of the dataframe.
    (max_row, max_col) = df.shape

    # Configure the series of the chart from the dataframe data.
    chart.add_series({"values": ["Sheet1", 1, max_col - 1, max_row, max_col - 1]})

    # Insert the chart into the worksheet.
    worksheet.insert_chart(1, 3, chart)
```

35.6 Example: Polars Excel output with conditional formatting

An example of adding a dataframe with conditional formatting to a worksheet using Polars and XlsxWriter. See *Working with Polars and XlsxWriter* for more details.
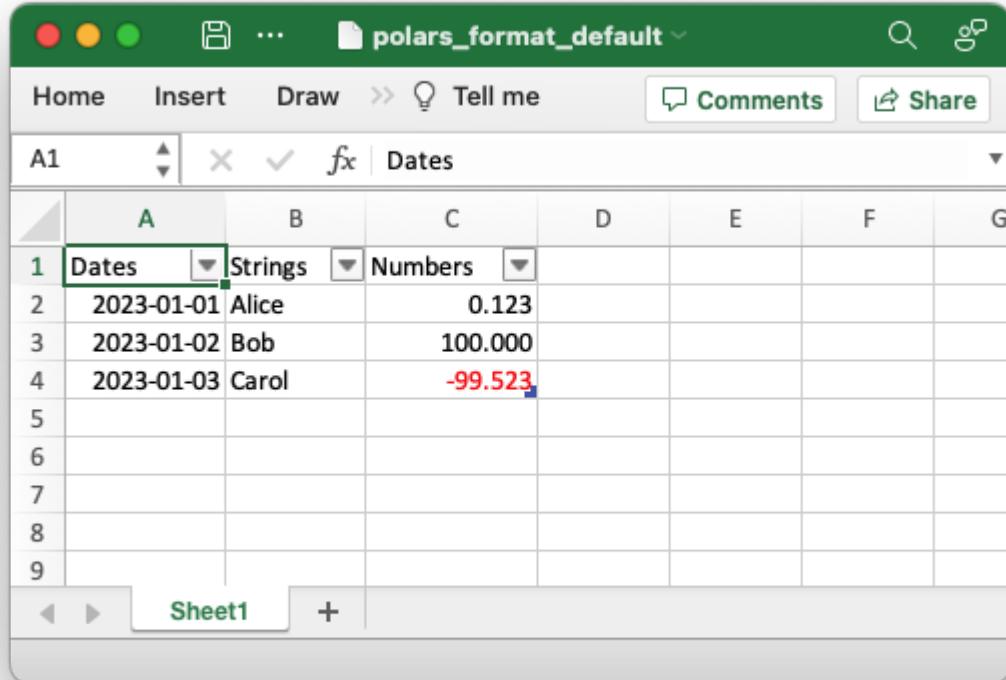
```
##############################################################################
#
# An example of adding a Polars dataframe to a worksheet with a conditional
# format.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import polars as pl

df = pl.DataFrame({"Data": [10, 20, 30, 20, 15, 30, 45]})

df.write_excel(
    workbook="pandas_conditional.xlsx",
    conditional_formats={"Data": {"type": "3_color_scale"}},
)
```

35.7 Example: Polars default format example

A simple example of converting a Polars dataframe to an xlsx file with default formatting. See
Working with Polars and XlsxWriter for more details.

```
##############################################################################
#
# A simple example of converting a Polars dataframe to an xlsx file with
# default formatting.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

from datetime import date
import polars as pl

# Create a Pandas dataframe with some sample data.
df = pl.DataFrame(
    {
        "Dates": [date(2023, 1, 1), date(2023, 1, 2), date(2023, 1, 3)],
        "Strings": ["Alice", "Bob", "Carol"],
        "Numbers": [0.12345, 100, -99.523],
    }
)

# Write the dataframe to a new Excel file with autofit on.
df.write_excel(workbook="polars_format_default.xlsx", autofit=True)
```

35.8 Example: Polars custom format example

A simple example of converting a Polars dataframe to an xlsx file with custom formatting of the worksheet table. See *Working with Polars and XlsxWriter* for more details.


```
################################################################################
#
# A simple example of converting a Polars dataframe to an xlsx file with
# custom formatting of the worksheet table.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

from datetime import date
import polars as pl

# Create a Pandas dataframe with some sample data.
df = pl.DataFrame(
    {
        "Dates": [date(2023, 1, 1), date(2023, 1, 2), date(2023, 1, 3)],
        "Strings": ["Alice", "Bob", "Carol"],
        "Numbers": [0.12345, 100, -99.523],
    }
```

```
)

# Write the dataframe to a new Excel file with formatting options.
df.write_excel(
    workbook="polars_format_custom.xlsx",
    # Set an alternative table style.
    table_style="Table Style Medium 4",
    # See the floating point precision for reals.
    float_precision=6,
    # Set an alternative number/date format for Polar Date types.
    dtype_formats={pl.Date: "yyyy mm dd;@"},
    # Add totals to the numeric columns.
    column_totals=True,
    # Autofit the column widths.
    autofit=True,
)
```

35.9 Example: Polars Excel output with sparklines

An example of adding a sparklines to the output of a Polars dataframe. See *Working with Polars and XlsxWriter* for more details.

```python
###############################################################################
#
# An example of writing multiple dataframes to worksheets using Polars and
# XlsxWriter.
#
# SPDX-License-Identifier: BSD-2-Clause
# Copyright 2013-2023, John McNamara, jmcnamara@cpan.org
#

import polars as pl
from polars.datatypes import INTEGER_DTYPES

df = pl.DataFrame(
    {
        "Zone": ["North", "South", "East", "West", "Central"],
        "Q1": [100, 55, -20, 0, 35],
        "Q2": [30, -10, 15, 60, 20],
        "Q3": [-50, 0, 40, 80, 80],
        "Q4": [75, 55, 25, -10, -55],
    }
)

# Write the dataframe with sparklines and some additional formatting.
df.write_excel(
    workbook="polars_sparklines.xlsx",
    # Set an alternative table style.
    table_style="Table Style Light 2",
    # Specify an Excel number format for integer types.
    dtype_formats={INTEGER_DTYPES: "#,##0_);(#,##0)"},
    # Configure sparklines to the dataframe.
    sparklines={
        # We use the default options with just  the source columns.
        "Trend": ["Q1", "Q2", "Q3", "Q4"],
        # We also add a customized sparkline type, with a positioning directive.
        "Change": {
            "columns": ["Q1", "Q2", "Q3", "Q4"],
            "insert_after": "Zone",
            "type": "win_loss",
        },
    },
    column_totals=["Q1", "Q2", "Q3", "Q4"],
    # Hide the default gridlines on the worksheet.
    hide_gridlines=True,
)
```

ALTERNATIVE MODULES FOR HANDLING EXCEL FILES

The following are some Python alternatives to XlsxWriter.

36.1 **OpenPyXL**

From the openpyxl documentation:

> openpyxl is a Python library to read/write Excel 2010 xlsx/xlsm/xltx/xltm files.

36.2 **Xlwings**

From the xlwings webpage:

> Leverage Python's scientific stack for interactive data analysis using Jupyter Note-books, NumPy, Pandas, scikit-learn etc. As such, xlwings is a free alternative to tools like Power BI or Tableau (Windows & Mac).

36.3 **XLWT**

From the xlwt documentation:

> xlwt is a library for writing data and formatting information to older Excel files (ie: .xls)

36.4 **XLRD**

From the xlrd documentation:

> xlrd is a library for reading data and formatting information from Excel files, whether they are .xls or .xlsx files.

LIBRARIES THAT USE OR ENHANCE XLSXWRITER

The following are some libraries or applications that wrap or extend XlsxWriter.

37.1 Pandas

Python Pandas is a Python data analysis library. It can read, filter and re-arrange small and large data sets and output them in a range of formats including Excel.

XlsxWriter is available as an Excel output engine in Pandas. See also See *Working with Pandas and XlsxWriter*.

37.2 XlsxPandasFormatter

XlsxPandasFormatter is a helper class that wraps the worksheet, workbook and dataframe objects written by Pandas to_excel() method using the xlsxwriter engine to allow consistent formatting of cells.

KNOWN ISSUES AND BUGS

This section lists known issues and bugs and gives some information on how to submit bug reports.

38.1 "Content is Unreadable. Open and Repair"

You may occasionally 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.

The error message and the actual file aren't helpful in debugging issues like this. If you do encounter this warning you should open an issue on GitHub with a program to replicate it (see *Reporting Bugs*).

38.2 "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.

38.3 Formulas displayed as #NAME? until edited

There are a few reasons why a formula written by XlsxWriter would generate a #NAME? error in Excel:

- Invalid formula syntax.

- Non-English function names.

- Semi-colon separators instead of commas.

- Use of Excel 2010 and later functions without a prefix.

See *Working with Formulas* and *Dealing with formula errors* for a more details and a explanation of how to debug the issue.

38.4 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.

See *Formula Results* for more details and a workaround.

38.5 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.

38.6 Charts series created from Worksheet Tables cannot have user defined names

In Excel, charts created from *Worksheet Tables* have a limitation where the data series name, if specified, must refer to a cell within the table.

To workaround this Excel limitation you can specify a user defined name in the table and refer to that from the chart. See *Charts from Worksheet Tables*.

REPORTING BUGS

Here are some tips on reporting bugs in XlsxWriter.

39.1 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.

39.2 Read the documentation

Read or search the XlsxWriter documentation to see if the issue you are encountering is already explained.

39.3 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.

39.4 Use the official XlsxWriter Issue tracker on GitHub

The official XlsxWriter Issue tracker is on GitHub.

39.5 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 analyze 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:
>
> ```python
> 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.

FREQUENTLY ASKED QUESTIONS

The section outlines some answers to frequently asked questions.

40.1 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.

40.2 Q. Why do my formulas show a zero result in some, non-Excel applications?

Due to a wide range of possible formulas and the 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', None, 4)
```

See also *Formula Results*.

Note: **LibreOffice** doesn't recalculate Excel formulas that reference other cells by default, in which case you will get the default XlsxWriter value of 0. You can work around this by setting the "LibreOffice Preferences -> LibreOffice Calc -> Formula -> Recalculation on File Load" option to "Always recalculate" (see the LibreOffice documentation). Or, you can set a blank result in the formula, which will also force recalculation:

```
worksheet.write_formula('A1', '=Sheet1!$A$1', None, '')
```

40.3 Q. Why do my formulas have a @ in them?

Microsoft refers to the @ in formulas as the Implicit Intersection Operator. It indicates that an input range is being reduced from multiple values to a single value. In some cases it is just a warning indicator and doesn't affect the calculation or result. However, in practical terms it generally means that your formula should be written as an array formula using either `write_array_formula()` or `write_dynamic_array_formula()`.

For more details see the *Dynamic Array support* and *Dynamic Arrays - The Implicit Intersection Operator "@"* sections of the XlsxWriter documentation.

40.4 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.

40.5 Q. Is feature X supported or will it be supported?

All supported features are documented. Future features are on the Roadmap.

40.6 Q. Can I password protect an XlsxWriter xlsx file

Although it is possible to password protect a worksheet using the Worksheet `protect()` method it isn't possible to password protect the entire workbook/file using XlsxWriter.

The reason for this is that a protected/encrypted xlsx file is in a different format from an ordinary xlsx file. This would require a lot of additional work, and testing, and isn't something that is on the XlsxWriter roadmap.

However, it is possible to password protect an XlsxWriter generated file using a third party open source tool called msoffice-crypt. This works for macOS, Linux and Windows:

```
msoffice-crypt.exe -e -p password clear.xlsx encrypted.xlsx
```

40.7 Q. Do people actually ask these questions frequently, or at all?

Apart from this question, yes.

CHANGES IN XLSXWRITER

This section shows changes and bug fixes in the XlsxWriter module.

41.1 Release 3.1.7 - October 9 2023

- Add the `very_hidden()` method to hide a worksheet. This is similar to the `hide()` method except that the worksheet cannot be unhidden in the the Excel user interface. The Excel worksheet "xlSheetVeryHidden" option can only be unset programmatically via VBA.

  Feature Request #947.

- Added fixes for column formulas in tables that were overridden by table data and which also didn't take future functions into account.

  Issue #1015.

41.2 Release 3.1.6 - October 1 2023

- Added support for chart leader lines for chart types other than Pie and Doughnut.

  Feature Request #1012.

41.3 Release 3.1.5 - September 24 2023

- Added support for adding signed VBA macros to workbooks via the via the `add_signed_vba_project()` method. See Working with VBA Macros.

  Feature Request #1008.

41.4 Release 3.1.4 - September 18 2023

- Added support for enabling the Excel "Show #N/A as an empty cell" chart option via the `show_na_as_empty_cell()` method.

  Feature Request #1008.

41.5 Release 3.1.3 - September 8 2023

- Added support for custom total formulas to worksheet tables.

  Feature Request #982.

41.6 Release 3.1.2 - May 28 2023

- Added worksheet page break preview mode via `set_pagebreak_view()`.

41.7 Release 3.1.1 - May 21 2023

- Add support for new Excel dynamic functions added in 2023.

  Issue #984.

- Added support for adding a color to the *invert_if_negative* chart option.

  Feature Request #854.

41.8 Release 3.1.0 - April 13 2023

- Minor fix for cell color issue.

41.9 Release 3.0.9 - March 10 2023

- Add documentation and examples on *Working with Polars and XlsxWriter* to demonstrate new Polars integration of XlsxWriter in write_excel().

- Add fix for rare issue with duplicate number formats.

41.10 Release 3.0.8 - February 3 2023

- Fix for `autofit()` exception when user defined column width was None.

41.11 Release 3.0.7 - January 14 2023

- Improved `autofit()` algorithm to account for the additional width of autofilter and table dropdowns.

- Improved `autofit()` take user defined column widths into account.

  Autofit will now only update the width for an existing column if it is greater than the user defined value. This allows the user to pre-load a minimum column width.

  Feature Request #936.

41.12 Release 3.0.6 - January 5 2023

- Added simulated worksheet `autofit()` method.

  Feature Request #936.

41.13 Release 3.0.5 - January 1 2023

- Added `OverlappingRange` exception which is raised during Worksheet `add_table()` or `merge_range()` if the range overlaps an existing worksheet table or merge range. This is a file corruption error in Excel. See *The Exceptions Class*.

  Issue #848

41.14 Release 3.0.4 - December 28 2022

- Roll up release of several minor fixes.
- Drop support for EOL Python 3.4 and 3.5.

41.15 Release 3.0.3 - February 27 2022

- Added `print_black_and_white()` worksheet method to set "Black and White" print options.

  Feature Request #862.

41.16 Release 3.0.2 - October 31 2021

- Added `set_top_left_cell()` worksheet method to position the first visible cell in a worksheet.

  Feature Request #837.

41.17 Release 3.0.1 - August 10 2021

- Add python_requires version to setup.py to help pip find the correct matching version for Python 2 and 3.

41.18 Release 3.0.0 - August 10 2021

- This is the first Python 3 only version of XlsxWriter. It is approximately 10-15% faster than the Python2/3 version.

  Python 2 users should use XlsxWriter version 2.0.0, see below.

41.19 Release 2.0.0 - August 9 2021

- This is the last XlsxWriter release to support Python 2. From now bug fixes and new features will only be available for Python 3. The deprecation notice for Python 2 support in XlsxWriter has been in place since May 2020 and the Python community support for Python 2 ended in January 2020. Downloads of XlsxWriter for Python 2 is currently less than 6% of all downloads of the library, and decreasing month by month.

  Python 2 users should still be able to install versions of XlsxWriter up to this release but not any releases afterwards.

  Feature Request #720.

41.20 Release 1.4.5 - July 29 2021

- Added Description/Alt Text and Decorative accessibility options for charts, textboxes and buttons. These options were already available for images.

41.21 Release 1.4.4 - July 4 2021

- Added some performance improvements. Performance for larger files should be 5-10% better.

41.22 Release 1.4.3 - May 12 2021

- Added support for background images in worksheets. See `set_background()` and *Example: Setting the Worksheet Background*.

41.23 Release 1.4.2 - May 7 2021

- Added support for GIF image files (and in Excel 365, animated GIF files).

41.24 Release 1.4.1 - May 6 2021

- Added support for dynamic arrays and new Excel 365 functions like UNIQUE and FILTER. See `write_dynamic_array_formula()`, *Dynamic Array support* and *Example: Dynamic array formulas*.

- Added constructor option "use_future_functions" to enable newer Excel "future" functions in Formulas. See *Formulas added in Excel 2010 and later*, and the `Workbook()` constructor.

41.25 Release 1.4.0 - April 23 2021

- Added fix for issue for where a y_axis font rotation of 0 was ignored.

41.26 Release 1.3.9 - April 15 2021

- Added option to set row heights and column widths in pixels via the `set_row_pixels()` and `set_column_pixels()` methods.

41.27 Release 1.3.8 - March 29 2021

- Added ability to add accessibility options "description" and "decorative" to images via `insert_image()`. Feature Request #768.

- Added fix for datetime.timedelta values that exceed the Excel 1900 leap day (timedeltas greater than 59 days, in hours). This is a backward incompatible change. Issue #731.

- Added the worksheet `read_only_recommended()` method to set the Excel "Read-only Recommended" option that is available when saving a file. Feature Request #784.

- Fixed issue where temp files used in *constant_memory* mode weren't closed/deleted if the workbook object was garbage collected. Issue #764.

- Fixed issue where pattern formats without colors were given a default black fill color. Issue #790.

- Added option to set a chart crossing to 'min' as well as the existing 'max' option. The 'min' option isn't available in the Excel interface but can be enabled via VBA. Feature Request #773.

41.28 Release 1.3.7 - October 13 2020

- Fixed issue where custom chart data labels didn't inherit the position of the data labels in the series. Issue #754.

- Added text alignment for textboxes. The existing options allowed the text area to be aligned but didn't offer control over the text within that area.

- Added Python 3.9 to the test matrix.

41.29 Release 1.3.6 - September 23 2020

- Added the worksheet `unprotect_range()` method to allow ranges within a protected worksheet to be unprotected. Feature Request #507.

- There are now over 1500 test cases in the test suite, including 900 tests that compare the output from XlsxWriter, byte for byte, against test files created in Excel. This is to ensure the maximum possible compatibility with Excel.

41.30 Release 1.3.5 - September 21 2020

- Fixed issue where relative url links in images didn't work. Issue #751.

- Added `use_zip64` as a constructor option. Issue #745.

- Added check, and warning, for worksheet tables with no data row. Either with or without a header row. Issue #715 and Issue #679.

- Add a warning when the string length in `write_rich_string()` exceeds Excel's limit. Issue #372.

41.31 Release 1.3.4 - September 16 2020

- Replaced internal MD5 digest used to check for duplicate images with a SHA256 digest to avoid issues on operating systems such as Red Hat in FIPS mode which don't support MD5 for security reasons. Issue #749.

41.32 Release 1.3.3 - August 13 2020

- Added `ignore_errors()` worksheet method to to allow Excel worksheet errors/warnings to be ignored in user defined ranges. See also *Example: Ignoring Worksheet errors and warnings*. Feature Request #678.

- Added warning when closing a file more than once via `close()` to help avoid errors where a file is closed within a loop or at the wrong scope level.

41.33 Release 1.3.2 - August 6 2020

- Added Border, Fill, Pattern and Gradient formatting to chart data labels and chart custom data labels. See *Chart series option: Data Labels* and *Chart series option: Custom Data Labels*.

41.34 Release 1.3.1 - August 3 2020

- Fix for issue where array formulas weren't included in the output file for certain ranges/conditions. Issue #735.

41.35 Release 1.3.0 - July 30 2020

- Added support for chart *custom data labels*. Feature Request #343.

41.36 Release 1.2.9 - May 29 2020

- Added support for `stacked` and `percent_stacked` Line charts.

41.37 Release 1.2.8 - February 22 2020

- Fix for issue where duplicate images with hyperlinks weren't handled correctly. Issue #686.

- Removed `ReservedWorksheetName` exception which was used with the reserved worksheet name "History" since this name is allowed in some Excel variants. Issue #688.

- Fix for worksheet objects (charts, images and textboxes) that are inserted with an offset that starts in a hidden cell. Issue #676.

- Fix to allow handling of NoneType in `add_write_handler()`. Issue #677.

41.38 Release 1.2.7 - December 23 2019

- Fix for duplicate images being copied to an XlsxWriter file. Excel uses an optimization where it only stores one copy of a repeated/duplicate image in a workbook. XlsxWriter didn't do this which meant that the file size would increase when then was a large number of repeated images. This release fixes that issue and replicates Excel's behavior. Issue #615.

- Added documentation on *Number Format Categories* and *Number Formats in different locales*.

- Added note to `protect()` about how it is possible to encrypt an XlsxWriter file using a third party, cross platform, open source tool called msoffice-crypt.

41.39 Release 1.2.6 - November 15 2019

- Added option to remove style from worksheet tables. Feature Request #670.

41.40 Release 1.2.5 - November 10 2019

- Added option to add hyperlinks to textboxes. See *Textbox Hyperlink*. Feature Request #419.

41.41 Release 1.2.4 - November 9 2019

- Added option to link textbox text from a cell. See *Textbox Textlink*. Feature Request #516.

- Added option to rotate text in a textbox. See *Textbox formatting: Text Rotation*. Feature Request #638.

41.42 Release 1.2.3 - November 7 2019

- Increased allowable worksheet url length from 255 to 2079 characters, as supported in more recent versions of Excel. A lower or user defined limit can be set via the `max_url_length` property in the `Workbook()` constructor.

- Fixed several issues with hyperlinks in worksheet images.

41.43 Release 1.2.2 - October 16 2019

- Fixed Python 3.8.0 warnings. Issue #660.

41.44 Release 1.2.1 - September 14 2019

- Added the `add_write_handler()` method to allow user defined types to be handled by the `write()` method. See *Writing user defined types* for more information. Feature Request #631.

- Add support for East Asian vertical fonts in charts. Feature Request #648.

41.45 Release 1.2.0 - August 26 2019

- Refactored exception handling around the workbook file `close()` method to allow exceptions to be caught and handled. See *Example: Catch exception on closing*. Also refactored the code to clean up temp files in the event of an exception. :issue'471' and Issue #647.

- Added the option to allow chart fonts to be rotated to 270 degrees to give a stacked orientation. See *Chart Fonts*. Issue #648.

41.46 Release 1.1.9 - August 19 2019

- Another fix for issues where zipfile.py raises "ZIP does not support timestamps before 1980" exception. Issue #651.

41.47 Release 1.1.8 - May 5 2019

- Added ability to combine Doughnut and Pie charts.

- Added gauge chart example which is a combination of a Doughnut and a Pie chart. See *Example: Gauge Chart*.

41.48 Release 1.1.7 - April 20 2019

- Added docs on *Working with Object Positioning*.

- Added fix for sizing of cell comment boxes when they cross columns/rows that have size changes that occur after the comment is written. Issue #403 and Issue #312.

- Added fix for the sizing of worksheet objects (images, charts, textboxes) when the underlying cell sizes have changed and the "object_position" parameter has been set to 1 "Move and size with cells". An additional mode 4 has been added to simulate inserting the object in hidden rows. Issue #618.

- Added object positioning for charts and textboxes, it was already supported for images. Note, the parameter is now called `object_position`. The previous parameter name `positioning` is deprecated but still supported for images. Issue #568.

41.49 Release 1.1.6 - April 7 2019

- Fixed issue where images that started in hidden rows/columns weren't placed correctly in the worksheet. Issue #613.

- Fixed the mime-type reported by system `file(1)`. The mime-type reported by "file –mime-type"/magic was incorrect for XlsxWriter files since it expected the `[Content_types]` to be the first file in the zip container. Issue #614.

41.50 Release 1.1.5 - February 22 2019

- This version removes support for end of life Pythons 2.5, 2.6, 3.1, 3.2 and 3.3. For older, unsupported versions of Python use version 1.1.4 of XlsxWriter.

41.51 Release 1.1.4 - February 10 2019

- Fix for issues where zipfile.py raises "ZIP does not support timestamps before 1980" exception. Issue #535.

41.52 Release 1.1.3 - February 9 2019

- Fix handling of `'num_format':  '0'` in duplicate formats. Issue #584.

41.53 Release 1.1.2 - October 20 2018

- Fix for issue where `in_memory` files weren't compressed. Issue #573.
- Fix `write()` so that it handles array formulas as documented. Issue #418.
- Fix for issue with special characters in worksheet table functions. Issue #442.
- Added warnings for input issues in `write_rich_string()` such as blank strings, double formats or insufficient parameters. Issue #425.

41.54 Release 1.1.1 - September 22 2018

- Added comment font name and size options. Issue #201.
- Fix for issue when using text boxes in the same workbook as a chartsheet. Issue #420.

41.55 Release 1.1.0 - September 2 2018

- Added functionality to align chart category axis labels. See the `label_align` property of the `set_x_axis()` method.
- Added worksheet `hide_row_col_headers()` method to turn off worksheet row and column headings. Issue #480.
- Added the `set_tab_ratio()` method to set the ratio between the worksheet tabs and the horizontal slider. Issue #481.
- Fixed issue with icon conditional formats when the values were zero. Issue #565.

41.56 Release 1.0.9 - August 27 2018

- Fix for issue with formulas quoted as strings in conditional formats, introduced in version 1.0.7. Issue #564.

41.57 Release 1.0.8 - August 27 2018

- Added named exceptions to XlsxWriter. See *The Exceptions Class*.

- Removed the implicit `close()` in the destructor since it wasn't guaranteed to work correctly and raised a confusing exception when any other exception was triggered. **Note that this is a backward incompatible change.** The `with` context manager is a better way to close automatically, see `close()`.

- Added border, fill, pattern and gradient formatting options to `set_legend()`. Issue #545.

- Added `top_right` position to `set_legend()`. Issue #537.

41.58 Release 1.0.7 - August 16 2018

- Fix for unicode type error in Python 3. Issue #554.

41.59 Release 1.0.6 - August 15 2018

- Added some performance improvements. Pull Request #551.

41.60 Release 1.0.5 - May 19 2018

- Added example of how to subclass the Workbook and Worksheet objects. See *Example: Example of subclassing the Workbook and Worksheet classes* and *Example: Advanced example of subclassing*.

- Added support for WMF and EMF image formats to the Worksheet `add_image()` method.

41.61 Release 1.0.4 - April 14 2018

- Set the xlsx internal file member datetimes to 1980-01-01 00:00:00 like Excel so that apps can produce a consistent binary file once the workbook `set_properties()` created date is set. Pull Request #495.

- Fix for jpeg images that reported unknown height/width due to unusual SOF markers. Issue #506.

- Added support for blanks in list autofilter. Issue #505.

41.62 Release 1.0.3 - April 10 2018

- Added Excel 2010 data bar features such as solid fills and control over the display of negative values. See *Working with Conditional Formatting* and *Example: Conditional Formatting*.

Feature Request #502.

- Fixed `set_column()` parameter names to match docs and other methods. Note, this is a backward incompatible change. Issue #504.

- Fixed missing plotarea formatting in pie/doughnut charts.

41.63 Release 1.0.2 - October 14 2017

- Fix for cases where the hyperlink style added in the previous release didn't work. Feature Request #455.

41.64 Release 1.0.1 - October 14 2017

- Changed default `write_url()` format to the Excel hyperlink style so that it changes when the theme is changed and also so that it indicates that the link has been clicked. Feature Request #455.

41.65 Release 1.0.0 - September 16 2017

- Added icon sets to conditional formatting. See *Working with Conditional Formatting* and *Example: Conditional Formatting*. Feature Request #387.

41.66 Release 0.9.9 - September 5 2017

- Added `stop_if_true` parameter to conditional formatting. Feature Request #386.

41.67 Release 0.9.8 - July 1 2017

- Fixed issue where spurious deprecation warning was raised in `-Werror` mode. Issue #451.

41.68 Release 0.9.7 - June 25 2017

- Minor bug and doc fixes.

41.69 Release 0.9.6 - Dec 26 2016

- Fix for table with data but without a header. Issue #405.
- Add a warning when the number of series in a chart exceeds Excel's limit of 1. Issue #399.

41.70 Release 0.9.5 - Dec 24 2016

- Fix for missing *remove_timezone* option in Chart class. Pull Request #404 from Thomas Arnhold.

41.71 Release 0.9.4 - Dec 2 2016

- Added user definable removal of timezones in datetimes. See the `Workbook()` constructor option `remove_timezone` and *Timezone Handling in XlsxWriter*. Issue #257.

- Fix duplicate header warning in `add_table()` when there is only one user defined header. Issue #380.

- Fix for *center_across* property in `add_format()`. Issue #381.

41.72 Release 0.9.3 - July 8 2016

- Added check to `add_table()` to prevent duplicate header names which leads to a corrupt Excel file. Issue #362.

41.73 Release 0.9.2 - June 13 2016

- Added workbook `set_size()` method to set the workbook window size.

41.74 Release 0.9.1 - June 8 2016

- Added font support to chart `set_table()`.

- Documented used of font rotation in chart *data labels*. Issue #337.

41.75 Release 0.9.0 - June 7 2016

- Added *trendline properties*: `intercept`, `display_equation` and `display_r_squared`. Feature Request #357.

41.76 Release 0.8.9 - June 1 2016

- Fix for `insert_image()` issue when handling images with zero dpi. Issue #356.

41.77 Release 0.8.8 - May 31 2016

- Added workbook `set_custom_property()` method to set custom document properties. Feature Request #355.

41.78 Release 0.8.7 - May 13 2016

- Fix for issue when inserting read-only images on Windows. Issue #352.

- Added `get_worksheet_by_name()` method to allow the retrieval of a worksheet from a workbook via its name.

- Fixed issue where internal file creation and modification dates were in the local timezone instead of UTC.

41.79 Release 0.8.6 - April 27 2016

- Fix for `external:` urls where the target/anchor contains spaces. Issue #350.

41.80 Release 0.8.5 - April 17 2016

- Added additional documentation on *Working with Pandas and XlsxWriter* and *Pandas with XlsxWriter Examples*.
- Added fix for `set_center_across()` format method.

41.81 Release 0.8.4 - January 16 2016

- Fix for `write_url()` exception when the URL contains two # location/anchors. Note, URLs like this aren't strictly valid and cannot be entered manually in Excel. Issue #330.

41.82 Release 0.8.3 - January 14 2016

- Added options to configure chart axis tick placement. See `set_x_axis()`.

41.83 Release 0.8.2 - January 13 2016

- Added transparency option to solid fill colors in chart areas (*Chart formatting: Solid Fill*). Feature Request #298.

41.84 Release 0.8.1 - January 12 2016

- Added option to set chart tick interval. Feature Request #251.

41.85 Release 0.8.0 - January 10 2016

- Added additional documentation on *Working with Formulas*.

41.86 Release 0.7.9 - January 9 2016

- Added chart pattern fills, see *Chart formatting: Pattern Fill* and *Example: Chart with Pattern Fills*. Feature Request #268.

41.87 Release 0.7.8 - January 6 2016

- Add checks for valid and non-duplicate worksheet table names. Issue #319.

41.88 Release 0.7.7 - October 19 2015

- Added support for table header formatting and a fix for wrapped lines in the header. Feature Request #287.

41.89 Release 0.7.6 - October 7 2015

- Fix for images with negative offsets. Issue #273.

41.90 Release 0.7.5 - October 4 2015

- Allow hyperlinks longer than 255 characters when the link and anchor are each less than or equal to 255 characters.
- Added `hyperlink_base` document property. Feature Request #306.

41.91 Release 0.7.4 - September 29 2015

- Added option to allow data validation input messages with the 'any' validate parameter.
- Fixed url encoding of links to external files and directories. Issue #278.

41.92 Release 0.7.3 - May 7 2015

- Added documentation on *Working with Pandas and XlsxWriter* and *Pandas with XlsxWriter Examples*.
- Added support for `with` context manager. :PR'239'.

41.93 Release 0.7.2 - March 29 2015

- Added support for textboxes in worksheets. See `insert_textbox()` and *Working with Textboxes* for more details. Feature Request #107.

41.94 Release 0.7.1 - March 23 2015

- Added gradient fills to chart objects such as the plot area of columns. See *Chart formatting: Gradient Fill* and *Example: Chart with Gradient Fills*. Feature Request #228.

41.95 Release 0.7.0 - March 21 2015

- Added support for display units in chart axes. See `set_x_axis()`. Feature Request #185.
- Added `nan_inf_to_errors` `Workbook()` constructor option to allow mapping of Python *nan/inf* value to Excel error formulas in `write()` and `write_number()`. Feature Request #150.

41.96 Release 0.6.9 - March 19 2015

- Added support for clustered category charts. See *Example: Clustered Chart* for details. Feature Request #180.

- Refactored the *The Format Class* and formatting documentation.

41.97 Release 0.6.8 - March 17 2015

- Added option to combine two different chart types. See the *Combined Charts* section and *Example: Combined Chart* and *Example: Pareto Chart* for more details. Feature Request #72.

41.98 Release 0.6.7 - March 1 2015

- Added option to add function value in worksheet `add_table()`. Feature Request #216.

- Fix for A1 row/col numbers below lower bound. Issue #212.

41.99 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*.

41.100 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.

41.101 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. Feature Request #178.

- Fix for issue where the bar/column chart gap and overlap weren't applied to the secondary axis. Issue #177.

41.102 Release 0.6.3 - November 6 2014

- Added support for adding VBA macros to workbooks. See *Working with VBA Macros*. Feature Request #126.

41.103 Release 0.6.2 - November 1 2014

- Added chart axis line and fill properties. Feature Request #88.

41.104 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.

41.105 Release 0.6.0 - October 15 2014

- Added option to add images to headers and footers. See *Example: Adding Headers and Footers to Worksheets*. Feature Request #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()`. Feature Request #134.

41.106 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.

41.107 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.

41.108 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.

41.109 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.

41.110 Release 0.5.5 - May 6 2014

- Fix for incorrect chart offsets in `insert_chart()` and `set_size()`.

41.111 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. Feature Request #117.

- Fix for chart `error_bar` exceptions. Issue #115.

- Added clearer reporting of nested exceptions in `write()` methods. Pull Request #108.

- Added support for `inside_base` data label position in charts.

41.112 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.

41.113 Release 0.5.2 - December 31 2013

- Added date axis handling to charts. See *Example: Date Axis Chart*. Feature Request #73.
- Added support for non-contiguous chart ranges. Feature Request #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.

41.114 Release 0.5.1 - December 2 2013

- Added interval unit option for category axes. Feature Request #69.
- Fix for axis name font rotation.
- Fix for several minor issues with Pie chart legends.

41.115 Release 0.5.0 - November 17 2013

- Added *Chartsheets* to allow single worksheet charts. Feature Request #10.

41.116 Release 0.4.9 - November 17 2013

- Added *chart object positioning and sizing* to allow positioning of plotarea, legend, title and axis names. Feature Request #66.
- Added `set_title()` none option to turn off automatic titles.
- Improved `define_name()` name validation.
- Fix to prevent modification of user parameters in `conditional_format()`.

41.117 Release 0.4.8 - November 13 2013

- Added `in_memory` `Workbook()` constructor option to allow XlsxWriter to work on Google App Engine. Feature Request #28.

41.118 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 Bootstrap theme.
- Added *Example: Merging Cells with a Rich String*.

41.119 Release 0.4.6 - October 23 2013

• Added font formatting to chart legends.

41.120 Release 0.4.5 - October 21 2013

• Added `position_axis` chart axis option.

• Added optional list handling for chart names.

41.121 Release 0.4.4 - October 16 2013

• Documented use of *cell utility* functions.

• Fix for tables added in non-sequential order. Closes Issue #51 reported by calfzhou.

41.122 Release 0.4.3 - September 12 2013

• Fix for comments overlying columns with non-default width. Issue #45.

41.123 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().

41.124 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.

41.125 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.

41.126 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.

41.127 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.

41.128 Release 0.3.7 - August 16 2013

- Added `write_boolean()` function to write Excel boolean values. Feature Request #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 behavior can be obtained by setting the constructor option to True. **Note** This is a backward incompatibility.

41.129 Release 0.3.6 - July 26 2013

- Simplified import based on a suggestion from John Yeung. Feature Request #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. Feature Request #5.

41.130 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.

41.131 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 Issue #27 Thanks to John Yeung.

- Added chart axis font rotation.

41.132 Release 0.3.3 - June 10 2013

- Minor packaging fixes. Issue #14 and Issue #19.

- Fixed explicit UTF-8 file encoding for Python 3. Pull Request #15 from Alexandr Shadchin.

- Fixed invalid string formatting resulted in misleading stack trace. Pull Request #21 from Andrei Korostelev.

41.133 Release 0.3.2 - May 1 2013

- Speed optimizations. The module is now 10-15% faster on average.

41.134 Release 0.3.1 - April 27 2013

- Added chart support. See the *The Chart Class*, *Working with Charts* and *Chart Examples*.

41.135 Release 0.3.0 - April 7 2013

- Added worksheet sparklines. See *Working with Sparklines*, *Example: Sparklines (Simple)* and *Example: Sparklines (Advanced)*

41.136 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.

41.137 Release 0.2.8 - April 4 2013

- Added worksheet outlines and grouping. See *Working with Outlines and Grouping*.

41.138 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.

41.139 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.

41.140 Release 0.2.5 - April 1 2013

- Added additional `Workbook()` parameters `'tmpdir'` and `'date_1904'`.

41.141 Release 0.2.4 - March 31 2013

- Added `Workbook()` `'constant_memory'` constructor property to minimize 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.

41.142 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.

41.143 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.

41.144 Release 0.2.1 - March 25 2013

- Added support for datetime.datetime, datetime.date and datetime.time to the `write_datetime()` method. *Issue #3*. Thanks to Eduardo (eazb) and Josh English for the prompt.

41.145 Release 0.2.0 - March 24 2013

- Added conditional formatting. See the `conditional_format()` method, *Working with Conditional Formatting* and *Example: Conditional Formatting*.

41.146 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

41.147 Release 0.1.8 - March 18 2013

- Fixed Python 3 support.

41.148 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*.

41.149 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.

41.150 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.

41.151 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*.

41.152 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 color.

41.153 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.

41.154 Release 0.1.1 - March 3 2013

- Added the `write_url()` worksheet method for writing hyperlinks to a worksheet.

41.155 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

41.156 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.

41.157 Release 0.0.8 - February 26 2013

- Added the `merge_range()` method to merge worksheet cells.

41.158 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()`

41.159 Release 0.0.6 - February 22 2013

- Added page setup method.
    - `print_row_col_headers()`

41.160 Release 0.0.5 - February 21 2013

- Added page setup methods.
    - `repeat_rows()`
    - `repeat_columns()`

41.161 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()`

41.162 Release 0.0.3 - February 19 2013

- Added page setup method.

  - `set_margins()`

41.163 Release 0.0.2 - February 18 2013

- Added page setup methods.

  - `set_landscape()`

  - `set_portrait()`

  - `set_page_view()`

  - `set_paper()`

  - `print_across()`

41.164 Release 0.0.1 - February 17 2013

- First public release.

AUTHOR

XlsxWriter was written by John McNamara.

- GitHub
- Twitter @jmcnamara13

42.1 Asking questions

If you have questions about XlsxWriter here are some ways to deal with them:

- **Bug Reports**:

  See the *Reporting Bugs* section of the docs.

- **Feature Requests**:

  Open a Feature Request issue on Github issues.

- **Pull Requests**:

  See the Contributing Guide. Note, all Pull Requests must start with an Issue Tracker.

- **General Questions**:

  General questions about how to use the module should be asked on StackOverflow. Add the `xlsxwriter` tag to the question.

  Questions on StackOverflow have the advantage of (usually) getting several answers and it also leaves a searchable question for someone else.

- **Email**:

  If none of the above apply you can contact me at jmcnamara@cpan.org.

42.2 Sponsorship and Donations

I write and maintain a series of open source libraries for creating Excel files. The most commonly used are XlsxWriter in Python, Libxlsxwriter in C and Excel::Writer::XLSX and Spreadsheet::WriteExcel in Perl.

My aim is to write well documented and well tested code that does what the user needs and doesn't get in their way. You can help make this continue, or show your appreciation for work to date, by becoming a GitHub Sponsor.

Or make a one-off donation via PayPal.

LICENSE

XlsxWriter is released under a BSD 2-Clause license.

BSD 2-Clause License

use_zip64() (built-in function), 39

V

very_hidden() (built-in function), 87

W

Workbook() (built-in function), 21
worksheets() (built-in function), 38
write() (built-in function), 41
write_array_formula() (built-in function), 48
write_blank() (built-in function), 51
write_boolean() (built-in function), 52
write_column() (built-in function), 59
write_comment() (built-in function), 82
write_datetime() (built-in function), 52
write_dynamic_array_formula() (built-in function), 50
write_formula() (built-in function), 47
write_number() (built-in function), 46
write_rich_string() (built-in function), 56
write_row() (built-in function), 58
write_string() (built-in function), 45
write_url() (built-in function), 53

X

xl_cell_to_rowcol() (built-in function), 193
xl_col_to_name() (built-in function), 194
xl_range() (built-in function), 194
xl_range_abs() (built-in function), 194
xl_rowcol_to_cell() (built-in function), 193
XlsxFileError, 183
XlsxInputError, 184
XlsxWriterException, 183