Creating Excel files with xlsxwriter.lua

Release 0.0.4

John McNamara

April 14, 2014

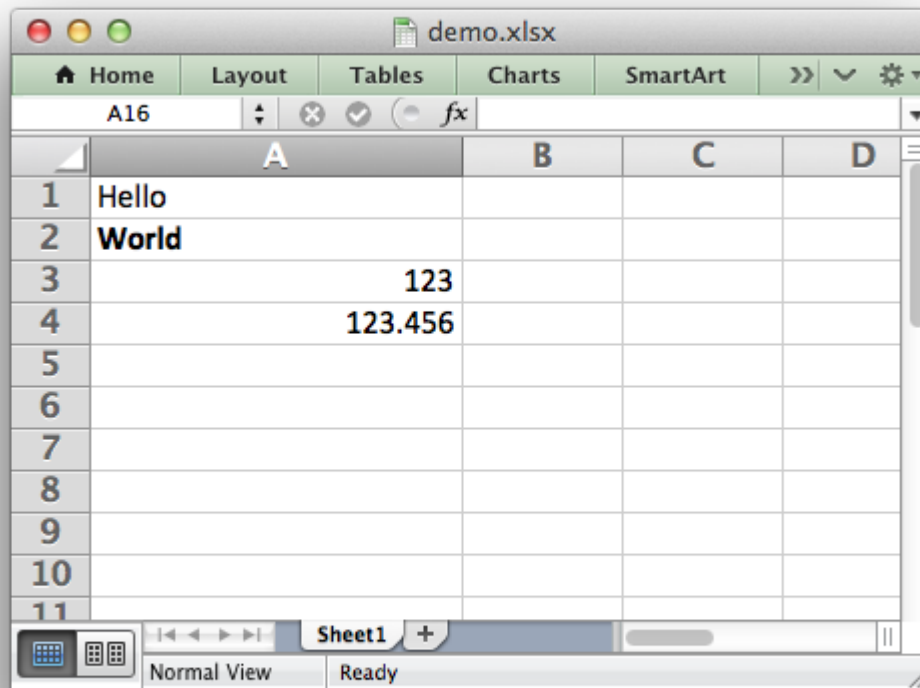**Xlsxwriter** is a Lua module for creating Excel XLSX files.

Xlsxwriter can be used to write text, numbers and formulas to multiple worksheets in an Excel 2007+ XLSX file. It supports features such as:

- 100% compatible Excel XLSX files.
- Full formatting.
- Memory optimisation mode for writing large files.
- Merged cells.
- Worksheet setup methods.
- Defined names.
- Document properties.

It works with Lua 5.1 and Lua 5.2.

CONTENTS   **1**

INTRODUCTION

Xlsxwriter is a Lua 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.

The main advantages of using Xlswriter are:

- 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.

However:

- It can only create **new files**. It cannot read or modify existing files.

Xlsxwriter is a Lua port of the Perl Excel::Writer::XLSX and the Python XlsxWriter modules and is licensed under an MIT/X11 *License*.

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

Xlsxwriter is a pure Lua module and doesn't need a native compiler to install. However, it has a dependency on the ZipWriter module which does have binary dependencies.

These dependencies are handled automatically if you use the `luarocks` or `luadist` methods shown below.

2.1.1 Using luarocks

The easiest way to install `xlsxwriter` is with the luarocks utility:

```
$ sudo luarocks install xlsxwriter
```

2.1.2 Using luadist

Another easy "packaged" way of installing `xlsxwriter` is with the luadist distribution:

```
$ sudo luadist install xlsxwriter
```

2.1.3 Cloning from GitHub

The `xlsxwriter` source code and bug tracker is in the xlsxwriter.lua repository on GitHub. You can clone the repository and install from it as follows:

```
$ git clone https://github.com/jmcnamara/xlsxwriter.lua.git

$ cd xlsxwriter.lua
$ sudo luarocks make
# or
$ sudo luadist make
```

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:

```lua
local Workbook = require "xlsxwriter.workbook"

local workbook  = Workbook:new("hello_world.xlsx")
local worksheet = workbook:add_worksheet()

worksheet:write("A1", "Hello world")

workbook:close()
```

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

```
$ lua hello.lua
```

This will output a file called `hello.xlsx` which should look something like the following:
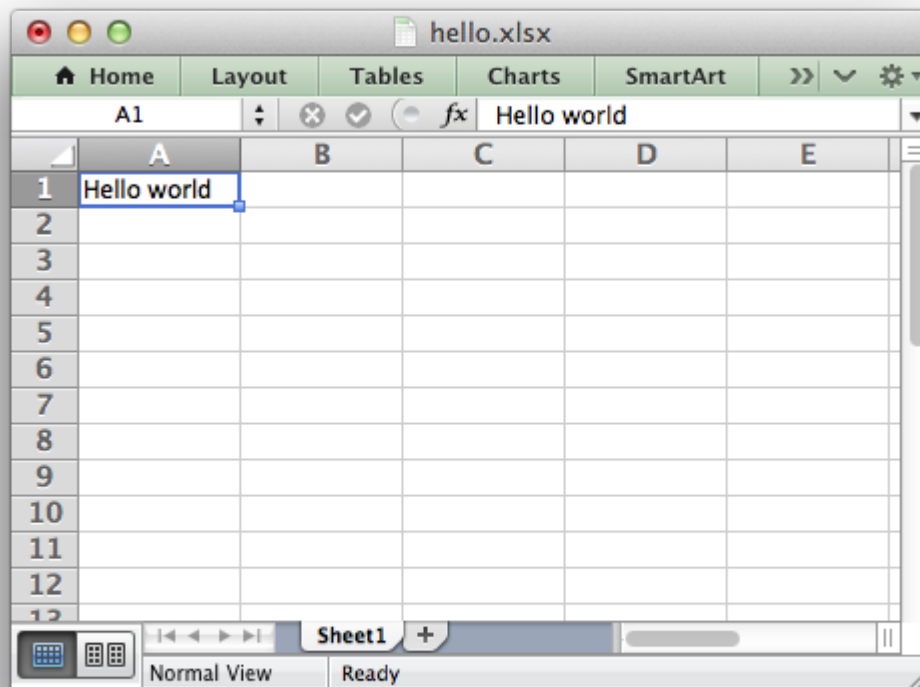
If you downloaded a tarball or cloned the repo, as shown above, you should also have a directory called examples with some sample applications that demonstrate different features of xlsxwriter.

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 Lua 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:

```lua
local Workbook = require "xlsxwriter.workbook"


-- Create a workbook and add a worksheet.
local workbook  = Workbook:new("Expensese01.xlsx")
local worksheet = workbook:add_worksheet()

-- Some data we want to write to the worksheet.
local expenses = {
  {"Rent", 1000},
  {"Gas",   100},
  {"Food",  300},
  {"Gym",    50},
}

-- Start from the first cell. Rows and columns are zero indexed.
local row = 0
local col = 0

-- Iterate over the data and write it out element by element.
for _, expense in ipairs(expenses) do
  local item, cost = unpack(expense)
  worksheet:write(row, col,     item)
  worksheet:write(row, col + 1, cost)
  row = row + 1
end
```

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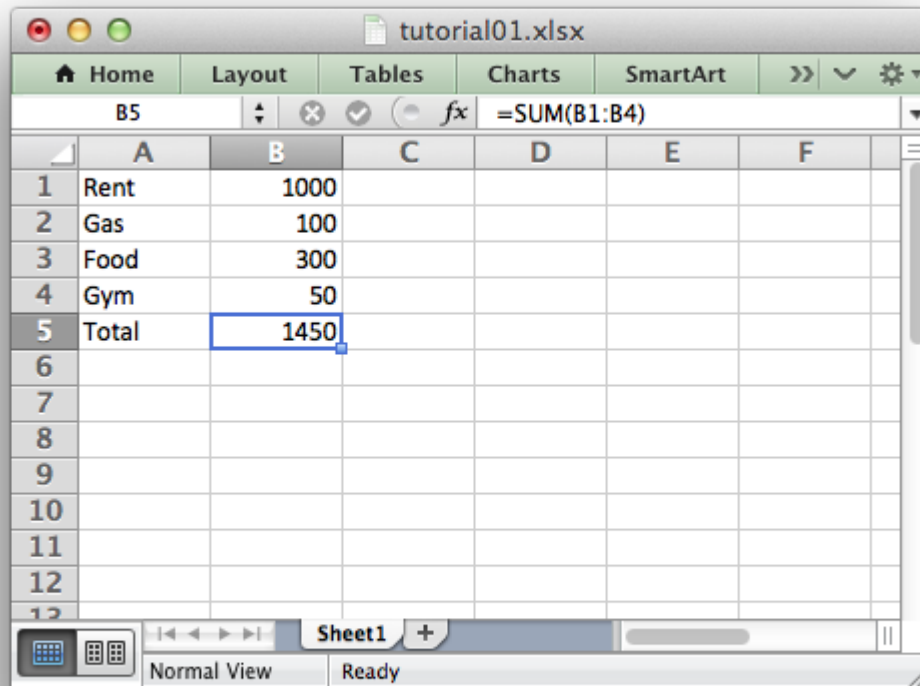
This is a simple example but the steps involved are representative of all programs that use xl-sxwriter, so let's break it down into separate parts.

The first step is to import the module:

```
    local Workbook = require "xlsxwriter.workbook"
```

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

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

```
    local workbook = Workbook:new("Expensese01.xlsx")
```

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

```
    local 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 the `xlsxwriter` API *rows* and *columns* are zero indexed. Thus, 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 element by element.
for _, expense in ipairs(expenses) do
  local item, cost = unpack(expense)
  worksheet:write(row, col,     item)
  worksheet:write(row, col + 1, cost)
  row = row + 1
end
```

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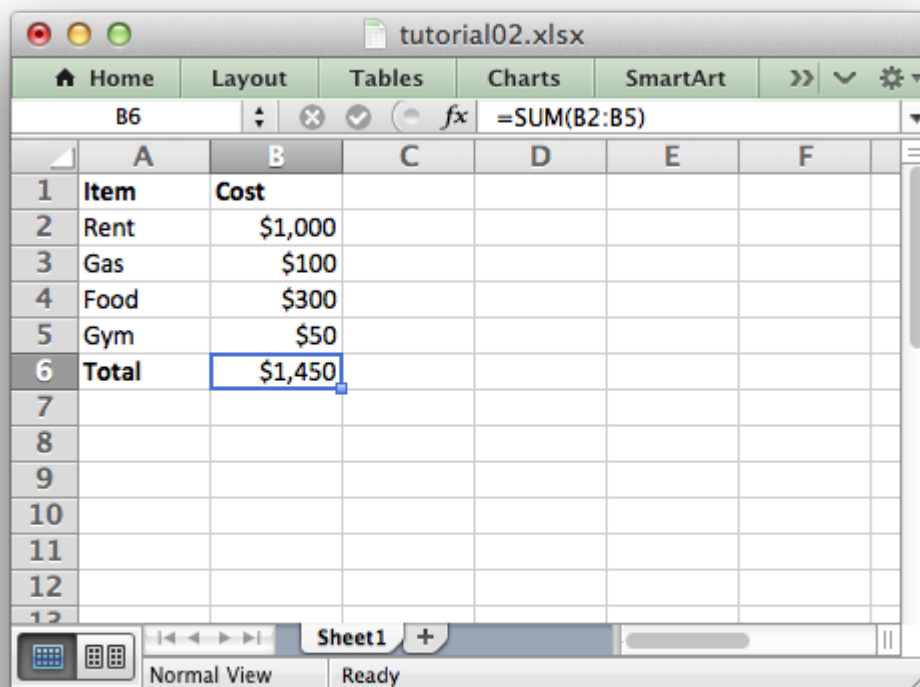
And that's it. We now have a file that can be read by Excel and other spreadsheet applications.

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 Lua 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:

```lua
local Workbook = require "xlsxwriter.workbook"


-- Create a workbook and add a worksheet.
local workbook  = Workbook:new("Expensese02.xlsx")
local worksheet = workbook:add_worksheet()

-- Add a bold format to use to highlight cells.
local bold = workbook:add_format({bold = true})

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

-- Write some data header.
worksheet:write("A1", "Item", bold)
worksheet:write("B1", "Cost", bold)

-- Some data we want to write to the worksheet.
local expenses = {
  {"Rent", 1000},
  {"Gas",   100},
  {"Food",  300},
  {"Gym",    50},
}

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

-- Iterate over the data and write it out element by element.
for _, expense in ipairs(expenses) do
  local item, cost = unpack(expense)
  worksheet:write(row, col,     item)
  worksheet:write(row, col + 1, cost, money)
  row = row + 1
end

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

workbook:close()
```

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

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

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

```lua
-- Add a bold format to use to highlight cells.
local bold = workbook:add_format({bold = true})

-- Add a number format for cells with money.
local 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.
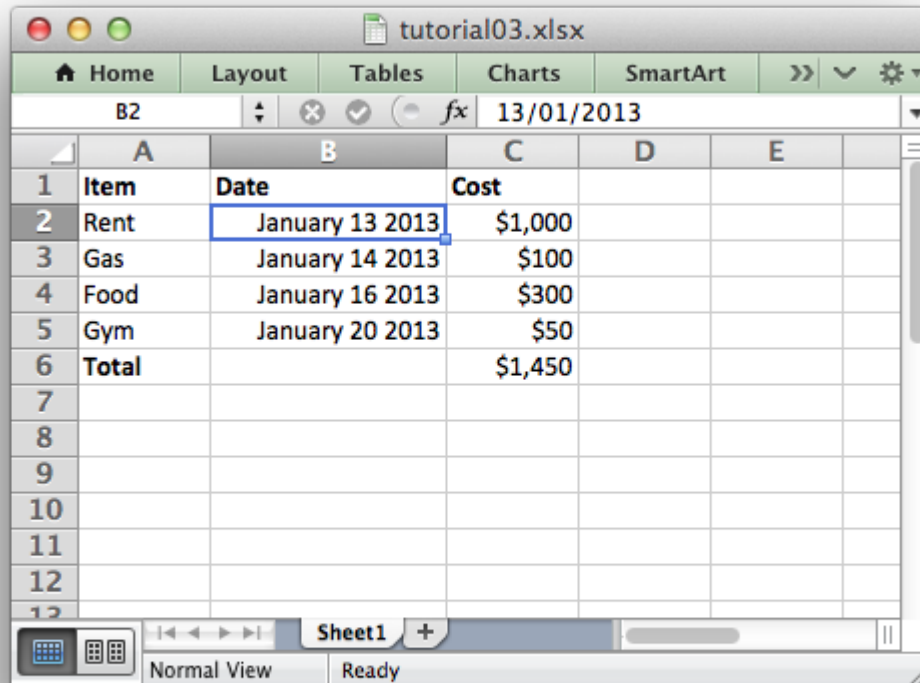
TUTORIAL 3: WRITING DIFFERENT TYPES OF DATA TO THE XLSX FILE

In the previous section we created a simple spreadsheet with formatting using Lua and the `xl-sxwriter` 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:

```lua
local Workbook = require "xlsxwriter.workbook"


-- Create a workbook and add a worksheet.
local workbook  = Workbook:new("Expensese03.xlsx")
local worksheet = workbook:add_worksheet()

-- Add a bold format to use to highlight cells.
local bold = workbook:add_format({bold = true})

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

-- Add an Excel date format.
local date_format = workbook:add_format({num_format = "mmmm d yyyy"})

-- Adjust the column width.
worksheet:set_column("B:B", 15)

-- Write some data header.
```

```lua
    worksheet:write("A1", "Item", bold)
    worksheet:write("B1", "Date", bold)
    worksheet:write("C1", "Cost", bold)

    -- Some data we want to write to the worksheet.
    local 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.
    local row = 1
    local col = 0

    -- Iterate over the data and write it out element by element.
    for _, expense in ipairs(expenses) do
      local item, date, cost = unpack(expense)

      worksheet:write_string     (row, col,     item)
      worksheet:write_date_string(row, col + 1, date, date_format)
      worksheet:write_number     (row, col + 2, cost, money)
      row = row + 1
    end

    -- Write a total using a formula.
    worksheet:write(row, 0, "Total",       bold)
    worksheet:write(row, 2, "=SUM(C2:C5)", money)

    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 Lua 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_boolean()

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_date_string(row, col + 1, date, date_format)
    worksheet:write_number      (row, col + 2, cost, money)
```

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. Since there is no native Lua date or time types `xlsxwriter` provides the `write_date_string()` and `write_date_time()` methods to convert dates and times into Excel date and time numbers.

In the example above we use `write_date_string()` but we also need to add the number format to ensure that Excel displays it as as date:

```
    ...

    local date_format = workbook:add_format({num_format = "mmmm d yyyy"})
    ...

    for _, expense in ipairs(expenses) do
      ...
      worksheet:write_date_string(row, col + 1, date, date_format)
      ...
    end
```

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:new(filename[,options])
    Create a new `xlsxwriter` Workbook object.

        Parameters

                • **filename** – The name of the new Excel file to create.

                • **options** – Optional workbook parameters. See below.

        Return type A Workbook object.

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

```lua
local Workbook = require "xlsxwriter.workbook"

workbook  = Workbook:new("filename.xlsx")
worksheet = workbook:add_worksheet()

worksheet:write(0, 0, "Hello Excel")

workbook:close()
```
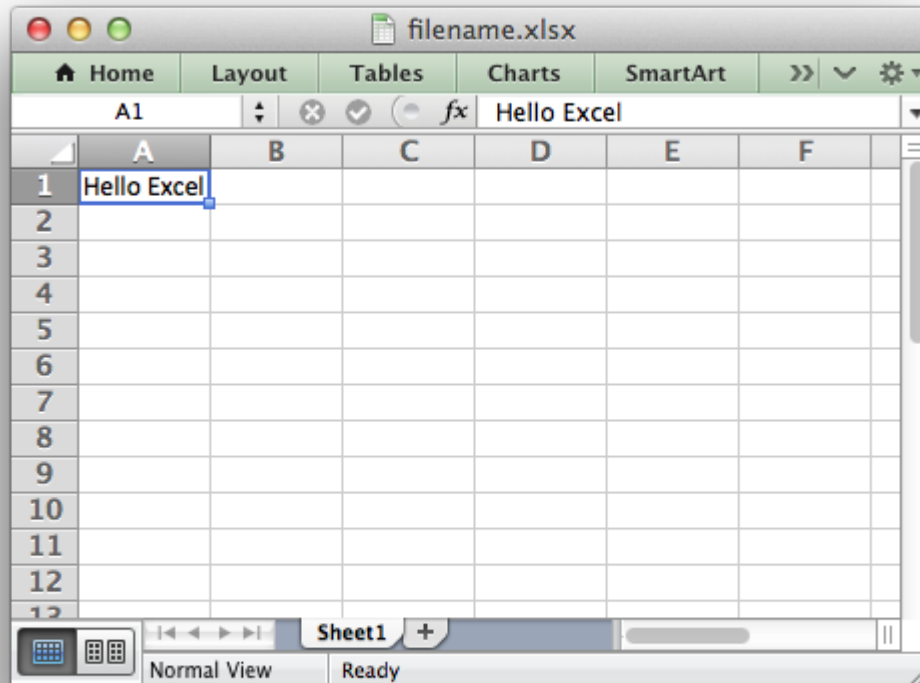
The constructor options are:

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

  ```
  workbook = Workbook:new(filename, {constant_memory = true})
  ```

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

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

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

6.2 workbook:add_worksheet()

add_worksheet([*sheetname*])
    Add a new worksheet to a workbook:

        Parameters sheetname – Optional worksheet name, defaults to Sheet1, etc.

Return type A *worksheet* object.

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

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

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

```
worksheet1 = workbook:add_worksheet()          -- Sheet1
worksheet2 = workbook:add_worksheet("Foglio2") -- Foglio2
worksheet3 = workbook:add_worksheet("Data")    -- Data
worksheet4 = workbook:add_worksheet()          -- Sheet4
```
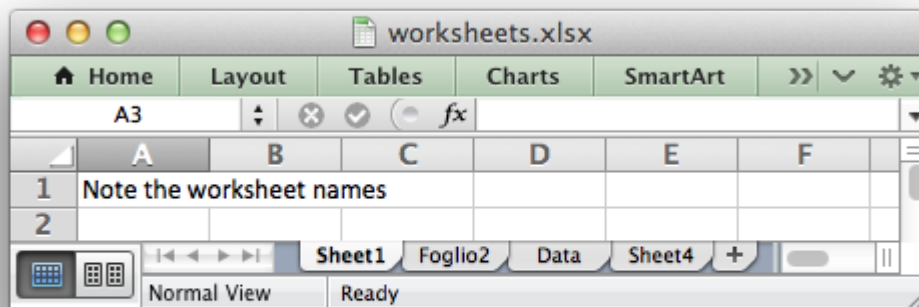

The worksheet name must be a valid Excel worksheet name, i.e. it cannot contain any of the characters [ ] : * ? / \ and it must be less than 32 characters.

In addition, you cannot use the same, case insensitive, sheetname for more than one worksheet.

6.3 workbook:add_format()

add_format([*properties*])
    Create a new Format object to formats cells in worksheets.

        Paramionary properties An optional table 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 table of property values or later via method calls:

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

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

6.4 workbook:close()

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

This should be done for every file.

>   workbook:close()

Currently, there is no implicit `close()`.

6.5 workbook:set_properties()

set_properties()
>   Set the document properties such as Title, Author etc.
>
>   >   **Parameters properties** – Dictionary of document properties.

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

The properties that can be set are:

-   `title`

-   `subject`

-   `author`

-   `manager`

-   `company`

-   `category`

-   `keywords`

-   `comments`

-   `status`

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

```
workbook:set_properties({
    title =    'This is an example spreadsheet',
    subject =  'With document properties',
    author =   'John McNamara',
    manager =  'Dr. Heinz Doofenshmirtz',
```

```
        company =  'of Wolves',
        category = 'Example spreadsheets',
        keywords = 'Sample, Example, Properties',
        comments = 'Created with Lua and the xlsxwriter module'})
```
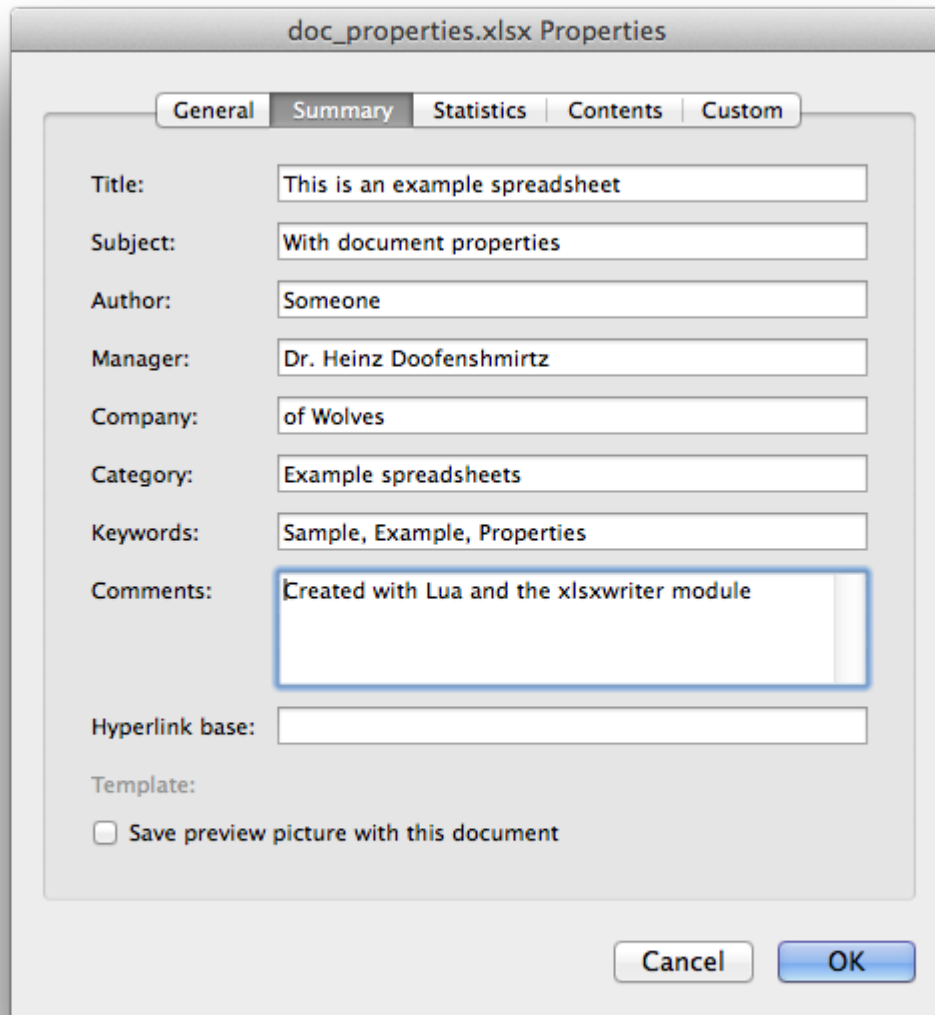

See also *Example: Setting Document Properties*.

6.6 workbook:define_name()

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

> **Parameters**
>
> - **name** – The defined name.
>
> - **formula** – 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 defined names can then be used in formulas:

```
workbook:define_name("Exchange_rate", "=0.96")
worksheet:write("B3", "=Exchange_rate")
```
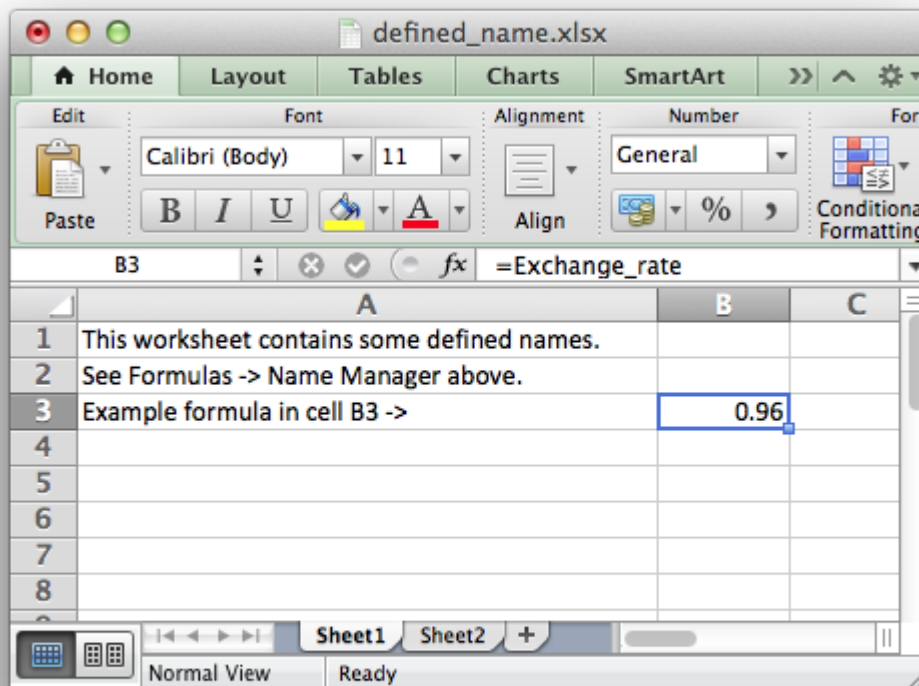

As in Excel a name defined like this is "global" to the workbook and can be referred to from any worksheet:

```
-- 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")
```

See also *Example: Adding Defined Names*.

6.7 workbook:worksheets()

worksheets()
> Return a sequence of the worksheet objects in the workbook:
>
>> **Return type** A sequence of *worksheet* objects.

The `worksheets()` method returns a table/sequence of the worksheets in a workbook suitable for iteration with `ipairs()`. This is useful if you want to repeat an operation on each worksheet in a workbook:

```lua
for _, worksheet in ipairs(workbook:worksheets()) do
    worksheet:write("A1", "Hello")
end
```

THE WORKSHEET CLASS

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

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

```
workbook   = Workbook:new("filename.xlsx")

worksheet1 = workbook:add_worksheet()
worksheet2 = workbook:add_worksheet()

worksheet1:write("A1", 123)
```
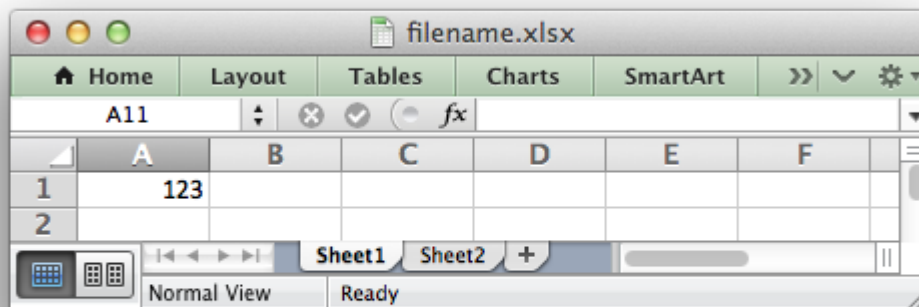

7.1 worksheet:write()

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 or format.

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

- `write_string()`

- `write_number()`

- `write_blank()`

- `write_formula()`

- `write_boolean()`

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

- Variables of Lua type `number` are written using `write_number()`.

- Empty strings and `nil` are written using `write_blank()`.

- Variables of Lua type `boolean` are written using `write_boolean()`.

Strings are then handled as follows:

- Strings that start with `"="` are taken to match a formula and are written using `write_formula()`.

- Strings that don't match any of the above criteria are written using `write_string()`.

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, nil)           -- write_blank()
```

This creates a worksheet like the following:
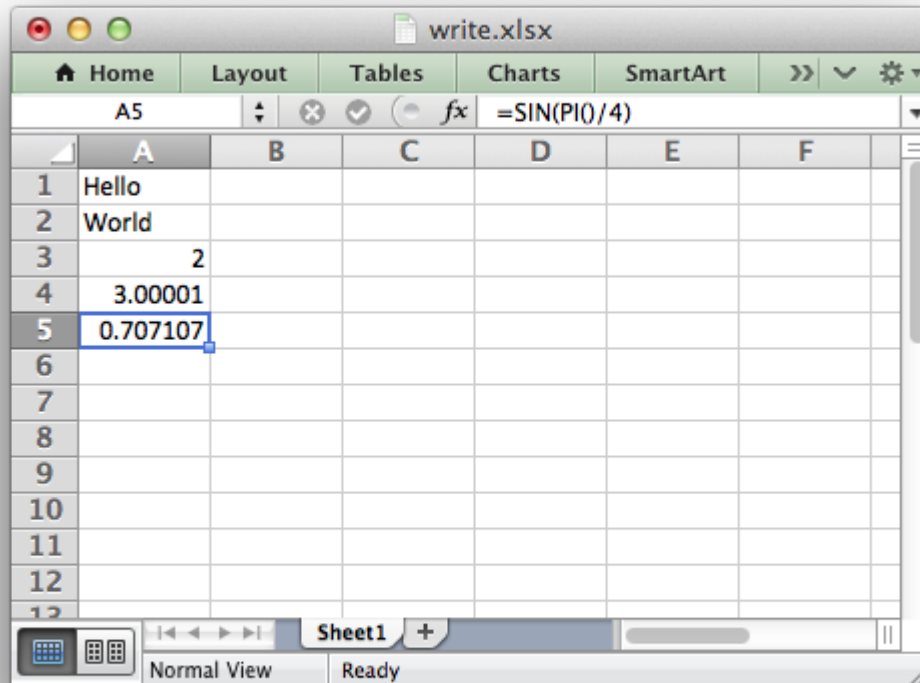
The `write()` method supports two forms of notation to designate the position of cells: **Row-column** notation and **A1** notation:

```
-- These are equivalent.
worksheet:write(0, 0, "Hello")
worksheet:write("A1", "Hello")
```

See *Working with Cell Notation* for more details.

The `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:

```
format = workbook:add_format({bold = true, italic = true})

worksheet:write(0, 0, "Hello", format) -- Cell is bold and italic.
```

7.2 worksheet:write_string()

write_string(*row, col, string*[, *format*])
    Write a string to a worksheet cell.

        Parameters

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

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

- **string** – String to write to cell.

- **format** – Optional *Format* object.

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

```
worksheet:write_string(0, 0, "Your text here")
worksheet:write_string("A2", "or here")
```

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

The `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 in Excel must be UTF-8 encoded. With `xlsxwriter` all that is required is that the source file is UTF-8 encoded and Lua will handle the UTF-8 strings like any other strings:

```
worksheet:write("A1", "Some UTF-8 text")
```
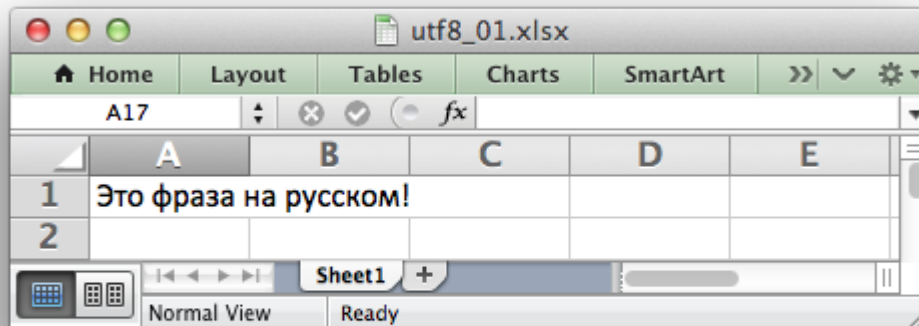

There are some sample UTF-8 sample programs in the `examples` directory of the `xlsxwriter` repository.

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

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

7.3 worksheet:write_number()

write_number(*row*, *col*, *number*[, *format*])
  Write a number to a worksheet cell.

  Parameters

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

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

  • **number** – Number to write to cell.

  • **format** – Optional *Format* object.

The write_number() method writes Lua number type variable to the cell specified by row and column:

```
worksheet:write_number(0, 0, 123456)
worksheet:write_number("A2", 2.3451)
```

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

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

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

7.4 worksheet:write_formula()

write_formula(*row*, *col*, *formula*[, *format*[, *value*]])
  Write a formula to a worksheet cell.

  Parameters

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

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

  • **formula** – Formula to write to cell.

  • **format** – Optional *Format* object.

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

```
worksheet:write_formula(0, 0, "=B3 + B4")
worksheet:write_formula(1, 0, "=SIN(PI()/4)")
worksheet:write_formula(2, 0, "=SUM(B1:B5)")
worksheet:write_formula("A4", "=IF(A3>1,"Yes", "No")")
```

```
    worksheet:write_formula("A5", "=AVERAGE(1, 2, 3, 4)")
    worksheet:write_formula("A6", "=DATEVALUE("1-Jan-2013")")
```

Array formulas are also supported:

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

See also the `write_array_formula()` method below.

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

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

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

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

```
    worksheet:write("A1", "=2+2", num_format, 4)
```

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

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

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

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

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

7.5 worksheet:write_array_formula()

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

> Write an array formula to a worksheet cell.
>
> > **Parameters**
> >
> > - **first_row** – The first row of the range. (All zero indexed.)
> > - **first_col** – The first column of the range.
> > - **last_row** – The last row of the range.
> > - **last_col** – The last col of the range.
> > - **formula** – Array formula to write to cell.
> > - **format** – Optional *Format* object.

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

An array formula is indicated by a pair of braces around the formula: {=SUM(A1:B1*A2:B2)}.

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

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

If the array formula returns a single value then the `first_` and `last_` parameters should be the same:

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

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

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

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

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

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

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

See also *Example: Array formulas*.

7.6 worksheet:write_blank()

write_blank(*row*, *col*, *blank*[, *format*])
> Write a blank worksheet cell.

> **Parameters**

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

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

>> • **blank** – `nil` or empty string. The value is ignored.

>> • **format** – Optional *Format* object.

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

```
worksheet:write_blank(0, 0, nil, format)
```

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

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

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

```
worksheet:write(0, 0, nil, format) -- write_blank()
worksheet:write(0, 1, nil)         -- Ignored
```

This seemingly uninteresting fact means that you can write tables of data without special treatment for `nil` or empty string values.

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

7.7 worksheet:write_boolean()

write_boolean(*row*, *col*, *boolean*[, *format*])
> Write a boolean value to a worksheet cell.

> **Parameters**

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

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

>> • **boolean** – Boolean value to write to cell.

>> • **format** – Optional *Format* object.

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

```
    worksheet:write_boolean(0, 0, true)
    worksheet:write_boolean("A2", false)
```

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

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

7.8 worksheet:write_date_time()

write_date_time(*row*, *col*, *date_time*[, *format*])
    Write a date or time to a worksheet cell.

        Parameters

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

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

                • **date_time** – A `os.time()` style table of date values.

                • **format** – Optional *Format* object.

The `write_date_time()` method can be used to write a date or time in `os.time()` style format to the cell specified by `row` and `column`:

```
    worksheet:write_date_time(0, 0, date_time, date_format)
```

The date_time should be a table of values like those used for os.time():

Key	Value
year	4 digit year
month	1 - 12
day	1 - 31
hour	0 - 23
min	0 - 59
sec	0 - 59.999

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

```
    date_format = workbook:add_format({num_format = "d mmmm yyyy"})
    date_time   = {year = 2014, month = 3, day = 17}

    worksheet:write_date_time("A1", date_time, date_format)
```

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

7.9 worksheet:write_date_string()

write_date_string(*row*, *col*, *date_string*[, *format*])
      Write a date or time to a worksheet cell.

            Parameters

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

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

                        • **date_string** – A os.time() style table of date values.

                        • **format** – Optional *Format* object.

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

```
worksheet:write_date_string(0, 0, date_string, date_format)
```

The date_string should be in the following format:

```
yyyy-mm-ddThh:mm:ss.sss
```

This conforms to an ISO8601 date but it should be noted that the full range of ISO8601 formats are not supported.

The following variations on the date_string parameter are permitted:

```
yyyy-mm-ddThh:mm:ss.sss  -- Standard format.
yyyy-mm-ddThh:mm:ss.sssZ -- Additional Z (but not time zones).
yyyy-mm-dd               -- Date only, no time.
           hh:mm:ss.sss  -- Time only, no date.
           hh:mm:ss      -- No fractional seconds.
```

Note that the T is required for cases with both date and time and seconds are required for all times.

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

```
date_format = workbook:add_format({num_format = "d mmmm yyyy"})

worksheet:write_date_string("A1", "2014-03-17", date_format)
```

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

7.10 worksheet:set_row()

set_row(*row*, *height*, *format*, *options*)
      Set properties for a row of cells.

            Parameters

                        • **row** – The worksheet row (zero indexed).

- **height** – The row height.
- **format** – Optional *Format* object.
- **options** – Optional row parameters: hidden, level, collapsed.

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

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

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

```
format = workbook:add_format({bold = true})

worksheet:set_row(0, 20, format)
```

If you wish to set the format of a row without changing the height you can pass `nil` as the height parameter or use the default row height of 15:

```
worksheet:set_row(1, nil, format)
worksheet:set_row(1, 15,  format) -- Same as above.
```

The `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, nil, 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 table with the following possible keys:

- `"hidden"`
- `"level"`
- `"collapsed"`

Options can be set as follows:

```
worksheet:set_row(0, 20, format, {hidden = true})

-- Or use defaults for other properties and set the options only.
worksheet:set_row(0, nil, nil, {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, nil, nil, {hidden = true})
```

The `"level"` parameter is used to set the outline level of the row. 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, nil, nil, {level = 1})
    worksheet:set_row(1, nil, nil, {level = 1})
    worksheet:set_row(2, nil, nil, {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, nil, nil, {hidden = true, level = 1})
    worksheet:set_row(2, nil, nil, {hidden = true, level = 1})
```

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

```
    worksheet:set_row(3, nil, nil, {collapsed = true})
```

7.11 worksheet:set_column()

set_column(*first_col*, *last_col*, *width*, *format*, *options*)
 Set properties for one or more columns of cells.

  Parameters

    • **first_col** – First column (zero-indexed).

    • **last_col** – Last column (zero-indexed). Can be same as firstcol.

    • **width** – The width of the column(s).

    • **format** – Optional *Format* object.

    • **options** – Optional parameters: hidden, level, collapsed.

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

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

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

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

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

Examples:

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

```
    worksheet:set_column("E:E", 20) -- Column  E   width set to 20.
    worksheet:set_column("F:H", 30) -- Columns F-H width set to 30.
```

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

As usual the `format` *Format* parameter is optional. If you wish to set the format without changing the width you can pass `nil` as the width parameter:

```
    format = workbook:add_format({bold = true})

    worksheet:set_column(0, 0, nil, format)
```

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

```
    worksheet:set_column("A:A", nil, 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, nil, format1)          -- Set format for row 1.
    worksheet:set_column("A:A", nil, format2) -- Set format for col 1.

    worksheet:write("A1", "Hello")              -- Defaults to format1
    worksheet:write("A2", "Hello")              -- Defaults to format2
```

The `options` parameters are the same as shown in `set_row()` above.

7.12 worksheet:get_name()

get_name()
> Retrieve the worksheet name.

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

```
    print(worksheet:get_name())
```

There is no `set_name()` method since the name needs to set when the worksheet object is created. The only safe way to set the worksheet nameis via the `add_worksheet()` method.

7.13 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()
```
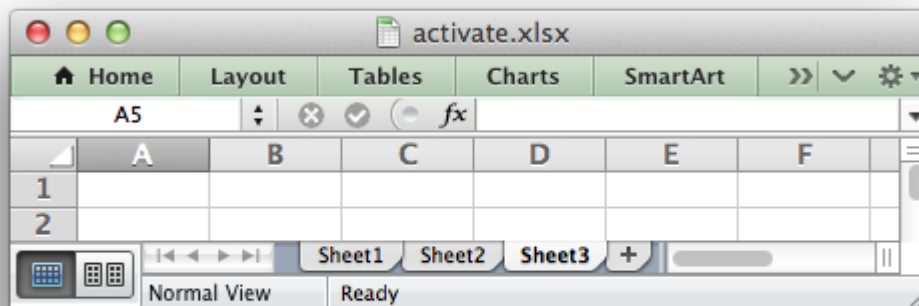

More than one worksheet can be selected via the `select()` method, see below, however only one worksheet can be active.

The default active worksheet is the first worksheet:

7.14 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.

7.15 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.
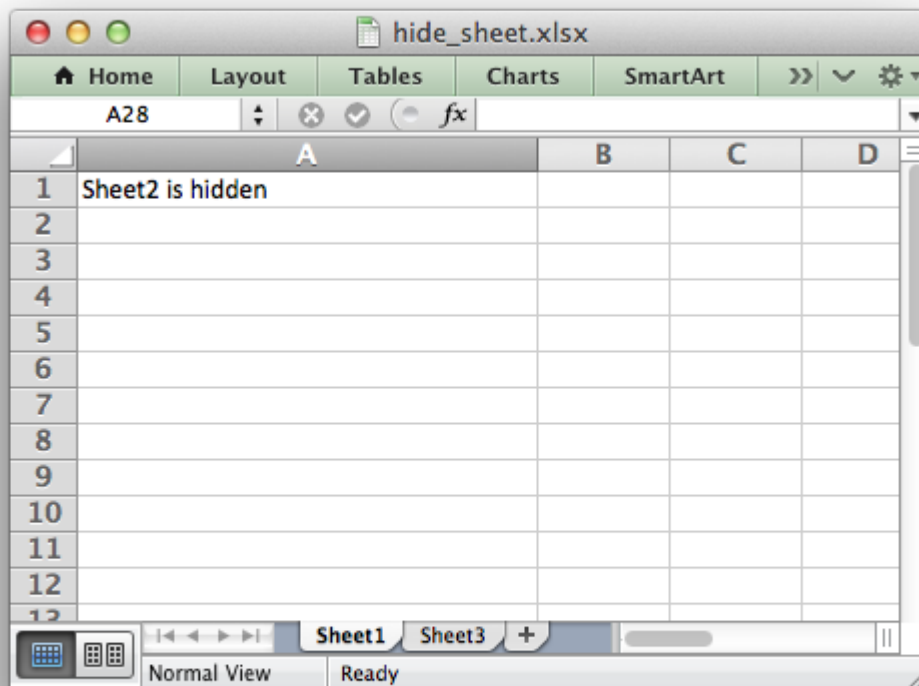
A hidden worksheet can not be activated or selected so this method is mutually exclusive with the `activate()` and `select()` methods. In addition, since the first worksheet will default to being the active worksheet, you cannot hide the first worksheet without activating another sheet:

```
worksheet2:activate()
worksheet1:hide()
```

See *Example: Hiding Worksheets* for more details.

7.16 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()`:

```lua
for i = 1, 20 do
  workbook:add_worksheet
end

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:

7.17 worksheet:merge_range()

merge_range(*first_row*, *first_col*, *last_row*, *last_col*, *format*)
>   Merge a range of cells.

>   **Parameters**

>   • **first_row** – The first row of the range. (All zero indexed.)

>   • **first_col** – The first column of the range.

>   • **last_row** – The last row of the range.

>   • **last_col** – The last col of the range.

>   • **data** – Cell data to write.

>   • **format** – Optional *Format* object.

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

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

```lua
merge_format = workbook:add_format({align = "center"})

worksheet:merge_range("B3:D4", "Merged Cells", merge_format)
```

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

```lua
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)
```
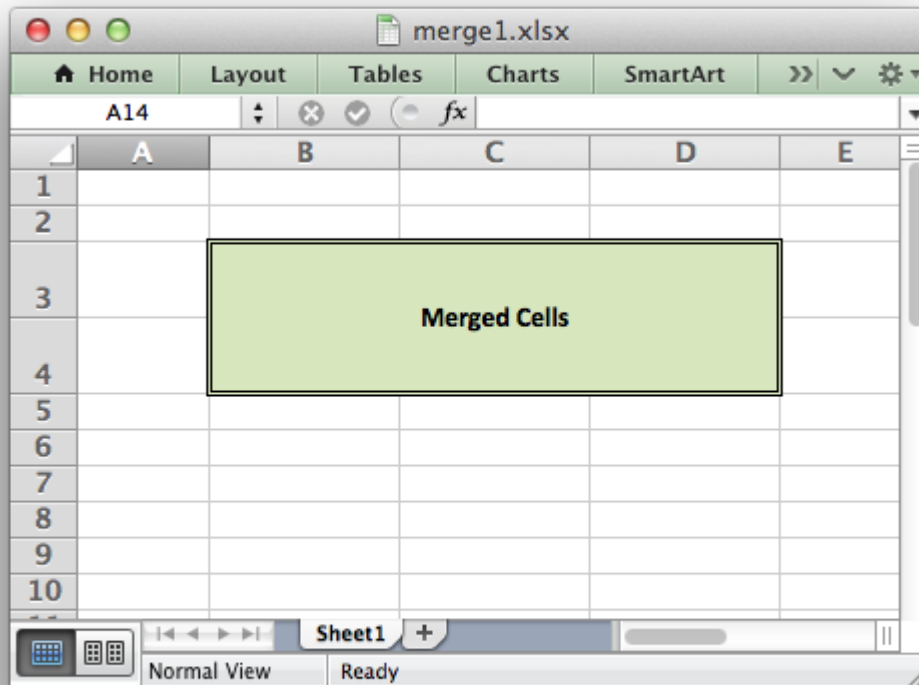

See *Example: Merging Cells* for more details.

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.

7.18 worksheet:set_zoom()

set_zoom(*zoom*)
    Set the worksheet zoom factor.

    Parameters zoom – 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()`.

7.19 worksheet:right_to_left()

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

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

>   worksheet:right_to_left()

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

7.20 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()

7.21 worksheet:set_tab_color()

`set_tab_color`()
>   Set the colour of the worksheet tab.

>   >   **Parameters color** – The tab color.

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

```
worksheet1:set_tab_color("red")
worksheet2:set_tab_color("#FF9900") -- Orange
```

The colour can be a Html style #RRGGBB string or a limited number named colours, see *Working with Colors* and *Example: Setting Worksheet Tab Colours* for more details.

THE WORKSHEET CLASS (PAGE SETUP)

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

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

8.1 worksheet:set_landscape()

set_landscape()
    Set the page orientation as landscape.

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

```
worksheet:set_landscape()
```

8.2 worksheet:set_portrait()

set_portrait()
    Set the page orientation as portrait.

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

```
worksheet:set_portrait()
```

8.3 worksheet:set_page_view()

set_page_view()
    Set the page view mode.

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

```
worksheet:set_page_view()
```

8.4 worksheet:set_paper()

set_paper(*index*)
Set the paper type.

> **Parameters index** – The Excel paper format index.

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

Index	Paper format	Paper size
0	Printer default	Printer default
1	Letter	8 1/2 x 11 in
2	Letter Small	8 1/2 x 11 in
3	Tabloid	11 x 17 in
4	Ledger	17 x 11 in
5	Legal	8 1/2 x 14 in
6	Statement	5 1/2 x 8 1/2 in
7	Executive	7 1/4 x 10 1/2 in
8	A3	297 x 420 mm
9	A4	210 x 297 mm
10	A4 Small	210 x 297 mm
11	A5	148 x 210 mm
12	B4	250 x 354 mm
13	B5	182 x 257 mm
14	Folio	8 1/2 x 13 in
15	Quarto	215 x 275 mm
16	—	10x14 in
17	—	11x17 in
18	Note	8 1/2 x 11 in
19	Envelope 9	3 7/8 x 8 7/8
20	Envelope 10	4 1/8 x 9 1/2
21	Envelope 11	4 1/2 x 10 3/8
22	Envelope 12	4 3/4 x 11
23	Envelope 14	5 x 11 1/2
24	C size sheet	—
25	D size sheet	—
26	E size sheet	—
27	Envelope DL	110 x 220 mm
28	Envelope C3	324 x 458 mm
29	Envelope C4	229 x 324 mm
30	Envelope C5	162 x 229 mm
31	Envelope C6	114 x 162 mm
32	Envelope C65	114 x 229 mm
Continued on next page		

Table 8.1 – continued from previous page

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

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

```
worksheet:set_paper(1) -- US Letter
worksheet:set_paper(9) -- A4
```

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

8.5 worksheet:center_horizontally()

center_horizontally()
    Center the printed page horizontally.

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

```
worksheet:center_horizontally()
```

8.6 worksheet:center_vertically()

center_vertically()
    Center the printed page vertically.

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

```
worksheet:center_vertically()
```

8.7 worksheet:set_margins()

set_margins(*[left=0.7,] right=0.7,] top=0.75,] bottom=0.75]]]*)
    Set the worksheet margins for the printed page.

        Parameters

- **left** – Left margin in inches. Default 0.7.

- **right** – Right margin in inches. Default 0.7.

- **top** – Top margin in inches. Default 0.75.

- **bottom** – Bottom margin in inches. Default 0.75.

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

8.8 worksheet:set_header()

set_header (*[header="",] margin=0.3]]*)
>   Set the printed page header caption and optional margin.

>>   **Parameters**

>>>   - **header** – Header string with Excel control characters.

>>>   - **margin** – Header margin in inches. Default 0.3.

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

The available control character are:

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

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

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

```
worksheet:set_header("&LHello")
```

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

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

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

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

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

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

```
worksheet:set_header("Hello")
```

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

You can have text in each of the justification regions:

```
worksheet:set_header("&LCiao&CBello&RCielo")
```

```
-----------------------------------------------------------------
|                                                               |
| Ciao                    Bello                           Cielo |
|                                                               |
```

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

```
worksheet:set_header("&CPage &P of &N")
```

```
-----------------------------------------------------------------
|                                                               |
|                          Page 1 of 6                          |
|                                                               |
```

```
worksheet:set_header("&CUpdated at &T")
```

```
-----------------------------------------------------------------
|                                                               |
|                        Updated at 12:30 PM                    |
|                                                               |
```

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

```
worksheet1:set_header("&C&30Hello Big")
worksheet2:set_header("&C&10Hello Small")
```

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

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

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

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

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

```
worksheet1:set_header("&CCuriouser && Curiouser - Attorneys at Law")
```

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

```
worksheet:set_header("&CHello", 0.75)
```

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

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

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

8.9 worksheet:set_footer()

set_footer(*[footer="",] margin=0.3]]*)
> Set the printed page footer caption and optional margin.

> > **Parameters**

> > > • **footer** – Footer string with Excel control characters.

> > > • **margin** – Footer margin in inches. Default 0.3.

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

8.10 worksheet:repeat_rows()

repeat_rows(*first_row*[, *last_row*])
> Set the number of rows to repeat at the top of each printed page.

> > **Parameters**

> > > • **first_row** – First row of repeat range.

> > > • **last_row** – Last row of repeat range. Optional.

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

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

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

8.11 worksheet:repeat_columns()

repeat_columns(*first_col*[, *last_col*])
> Set the columns to repeat at the left hand side of each printed page.

> > **Parameters**

> > > • **first_col** – First column of repeat range.

> > > • **last_col** – Last column of repeat range. Optional.

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

This can be achieved by using the `repeat_columns()` method. The parameters `first_column` and `last_column` are zero based. The `last_column` parameter is optional

if you only wish to specify one column. You can also specify the columns using A1 column notation, see *Working with Cell Notation* for more details.:

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

8.12 worksheet:hide_gridlines()

hide_gridlines([*option=1*])
    Set the option to hide gridlines on the screen and the printed page.

       Parameters option – 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:

1. Don't hide gridlines.

2. Hide printed gridlines only.

3. Hide screen and printed gridlines.

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

8.13 worksheet:print_row_col_headers()

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

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

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

```
worksheet:print_row_col_headers()
```

8.14 worksheet:print_area()

print_area(*first_row*, *first_col*, *last_row*, *last_col*)
    Set the print area in the current worksheet.

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

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

All four parameters must be specified. You can also use A1 notation, see *Working with Cell Notation*:

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

8.15 worksheet:print_across()

print_across()
    Set the order in which pages are printed.

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

```
worksheet:print_across()
```

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

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

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

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

8.16 worksheet:fit_to_pages()

fit_to_pages(*width*, *height*)
    Fit the printed area to a specific number of pages both vertically and horizontally.

Parameters

- **width** – Number of pages horizontally.

- **height** – 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 Excel only one of these options to be active at a time. The last method call made will set the active option.

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

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

8.17 worksheet:set_start_page()

`set_start_page()`
    Set the start page number when printing.

        Parameters start_page – Starting page number.

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

```
-- Start print from page 2.
worksheet:set_start_page(2)
```

8.18 worksheet:set_print_scale()

set_print_scale()
>   Set the scale factor for the printed page.

>   **Parameters scale** – Print scale of worksheet to be printed.

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

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

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

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

8.19 worksheet:set_h_pagebreaks()

set_h_pagebreaks(*breaks*)
>   Set the horizontal page breaks on a worksheet.

>   **Parameters breaks** – Table 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 table of one or more page breaks:

```
worksheet1:set_v_pagebreaks({20})
worksheet2:set_v_pagebreaks({20, 40, 60, 80, 100})
```

To create a page break between rows 20 and 21 you must specify the break at row 21. However in zero index notation this is actually row 20. So you can pretend for a small while that you are using 1 index notation:

```
worksheet:set_h_pagebreaks({20}) -- Break between row 20 and 21.
```

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

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

8.20 worksheet:set_v_pagebreaks()

set_v_pagebreaks(*breaks*)
>   Set the vertical page breaks on a worksheet.

>   >   **Parameters breaks** – Table 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, colours, patterns, borders, alignment and number formatting.

Format objects are created by calling the workbook `add_format()` method as follows:

```
format = workbook:add_format()
```

Format properties can be set by calling any of the methods shown in this section:

```
format = workbook:add_format()
format:set_bold()
format:set_font_color("red")
```

Alternatively the properties can be set by passing a table of properties to the `add_format()` constructor:

```
format = workbook:add_format({bold = true, font_color = "red"})
```

The documentation below shows the property methods but the information is equally applicable when using them in the `add_format()` constructor.
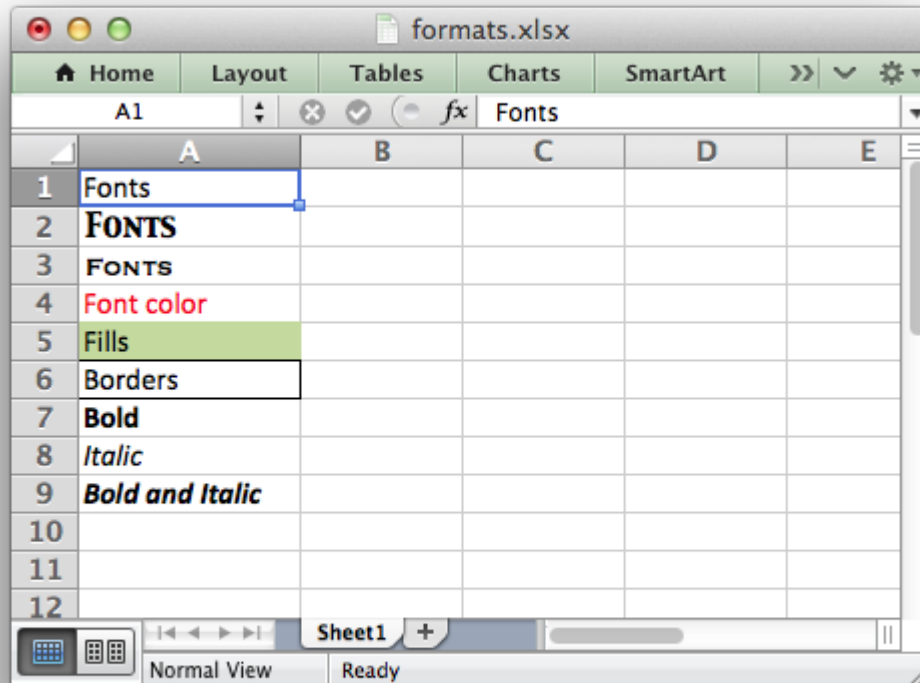
9.1 format:set_font_name()

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

> **Parameters fontname** – Cell font.

Specify the font used used in the cell format:

    cell_format:set_font_name("Times New Roman")

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

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

9.2 format:set_font_size()

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

> **Parameters size** – The cell font size.

Set the font size of the cell format:

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

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

9.3 format:set_font_color()

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

> **Parameters color** – The cell font color.

Set the font colour:

```
format = workbook:add_format()

format:set_font_color("red")

worksheet:write(0, 0, "wheelbarrow", format)
```

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

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

9.4 format:set_bold()

set_bold()
     Turn on bold for the format font.

Set the bold property of the font:

```
format:set_bold()
```

9.5 format:set_italic()

set_italic()
     Turn on italic for the format font.

Set the italic property of the font:

```
format:set_italic()
```

9.6 format:set_underline()

set_underline()
    Turn on underline for the format:

        Parameters style – Underline style.

Set the underline property of the format:

```
format:set_underline()
```

The available underline styles are:

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

9.7 format:set_font_strikeout()

set_font_strikeout()
    Set the strikeout property of the font.

9.8 format:set_font_script()

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

The available options are:

- 1 = Superscript
- 2 = Subscript

9.9 format:set_num_format()

set_num_format(*format_string*)
    Set the number format for a cell.

        Parameters format_string – 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:

```lua
format1 = workbook:add_format()
format2 = workbook:add_format()

format1:set_num_format("d mmm yyyy")   -- Format string.
format2:set_num_format(0x0F)           -- Format index.
```

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

```lua
format01:set_num_format("0.000")
worksheet:write(1, 0, 3.1415926, format01)     --> 3.142

format02:set_num_format("#,##0")
worksheet:write(2, 0, 1234.56, format02)       --> 1,235

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

format04:set_num_format("0.00")
worksheet:write(4, 0, 49.99, format04)         --> 49.99

format05:set_num_format("mm/dd/yy")
worksheet:write(5, 0, 36892.521, format05)     --> 01/01/01

format06:set_num_format("mmm d yyyy")
worksheet:write(6, 0, 36892.521, format06)     --> Jan 1 2001

format07:set_num_format("d mmmm yyyy")
worksheet:write(7, 0, 36892.521, format07)     --> 1 January 2001

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

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

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

-- Zip code.
format11:set_num_format("00000")
worksheet:write(13, 0, 1209, format11)
```
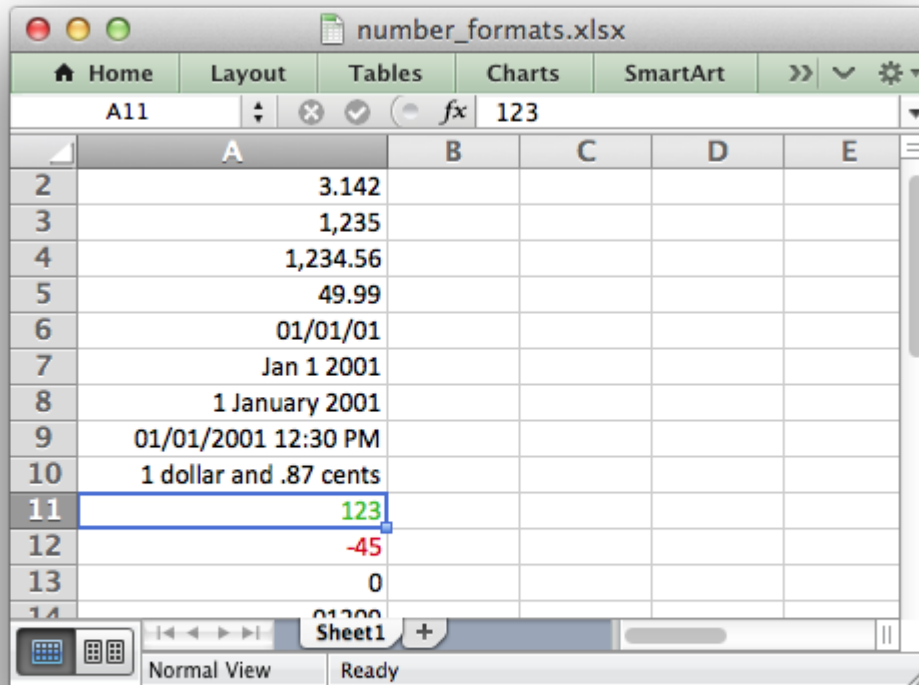
The number system used for dates is described in *Working with Dates and Time*.

The colour format should have one of the following values:

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

For more information refer to the Microsoft documentation on cell formats.

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

Index	Index	Format String
0	0x00	General
1	0x01	0
2	0x02	0.00
3	0x03	#,##0
4	0x04	#,##0.00
5	0x05	($#,##0_);($#,##0)
6	0x06	($#,##0_);[Red]($#,##0)
7	0x07	($#,##0.00_);($#,##0.00)
8	0x08	($#,##0.00_);[Red]($#,##0.00)
9	0x09	0%
10	0x0a	0.00%
11	0x0b	0.00E+00
		Continued on next page

Index	Index	Format String
12	0x0c	# ?/?
13	0x0d	# ??/??
14	0x0e	m/d/yy
15	0x0f	d-mmm-yy
16	0x10	d-mmm
17	0x11	mmm-yy
18	0x12	h:mm AM/PM
19	0x13	h:mm:ss AM/PM
20	0x14	h:mm
21	0x15	h:mm:ss
22	0x16	m/d/yy h:mm
...	...	...
37	0x25	(#,##0_);(#,##0)
38	0x26	(#,##0_);[Red](#,##0)
39	0x27	(#,##0.00_);(#,##0.00)
40	0x28	(#,##0.00_);[Red](#,##0.00)
41	0x29	_(* #,##0_);_(* (#,##0);_(* "-"_);_(@_)
42	0x2a	_($* #,##0_);_($* (#,##0);_($* "-"_);_(@_)
43	0x2b	_(* #,##0.00_);_(* (#,##0.00);_(* "-"??_);_(@_)
44	0x2c	_($* #,##0.00_);_($* (#,##0.00);_($* "-"??_);_(@_)
45	0x2d	mm:ss
46	0x2e	[h]:mm:ss
47	0x2f	mm:ss.0
48	0x30	##0.0E+0
49	0x31	@

Note: Numeric formats 23 to 36 are not documented by Microsoft and may differ in international versions. The listed date and currency formats may also vary depending on system settings.

Note: The dollar sign in the above formats appear as the defined local currency symbol.

9.10 format:set_locked()

set_locked(*state*)
    Set the cell locked state.

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

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

```
locked = workbook:add_format()
locked:set_locked(true)

unlocked = workbook:add_format()
locked:set_locked(false)

-- Enable worksheet protection
worksheet:protect()

-- This cell cannot be edited.
worksheet:write("A1", "=1+2", locked)

-- This cell can be edited.
worksheet:write("A2", "=1+2", unlocked)
```

9.11 format:set_hidden()

set_hidden()
> Hide formulas in a cell.

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

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

-- Enable worksheet protection
worksheet:protect()

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

9.12 format:set_align()

set_align(*alignment*)
> Set the alignment for data in the cell.

> > **Parameters alignment** – The vertical and or horizontal alignment direction.

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

Horizontal alignment
center
right
fill
justify
center_across

The following are the available vertical alignments:

Vertical alignment
top
vcenter
bottom
vjustify

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

```
format = workbook:add_format()

format:set_align("center")
format:set_align("vcenter")

worksheet:set_row(0, 30)
worksheet:write(0, 0, "Some Text", format)
```

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

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

9.13 format:set_center_across()

set_center_across()
>   Centre text across adjacent cells.

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

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

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

worksheet:write(1, 1, "Center across selection", format)
worksheet:write_blank(1, 2, format)
```

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

9.14 format:set_text_wrap()

set_text_wrap()
>   Wrap text in a cell.

Turn text wrapping on for text in a cell:

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

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

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

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

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

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

9.15 format:set_rotation()

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

        Parameters angle – Rotation angle in the range -90 to 90 and 270.

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

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

worksheet:write(0, 0, "This text is rotated", format)
```

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

9.16 format:set_indent()

set_indent(*level*)
    Set the cell text indentation level.

   **Parameters level** – Indentation level.

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

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

format1:set_indent(1)
format2:set_indent(2)

worksheet:write("A1", "This text is indented 1 level",  format1)
worksheet:write("A2", "This text is indented 2 levels", format2)
```
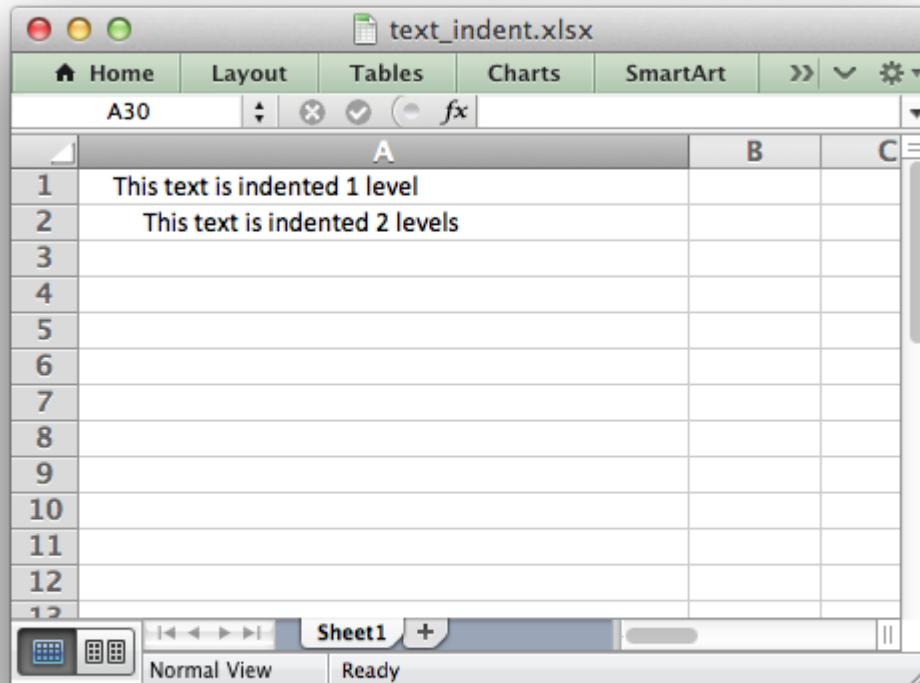
Indentation is a horizontal alignment property. It will override any other horizontal properties but it can be used in conjunction with vertical properties.

9.17 format:set_shrink()

set_shrink()
    Turn on the text "shrink to fit" for a cell.

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

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

worksheet:write(0, 0, "Honey, I shrunk the text!", format)
```

9.18 format:set_text_justlast()

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

Only applies to Far Eastern versions of Excel.

9.19 format:set_pattern()

set_pattern(*index*)

        Parameters index – Pattern index. 0 - 18.

Set the background pattern of a cell.

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

9.20 format:set_bg_color()

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

        Parameters color – The cell font color.

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

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

```
format = workbook:add_format()

format:set_pattern(1)   -- This is optional when using a solid fill.
format:set_bg_color("green")

worksheet:write("A1", "Ray", format)
```
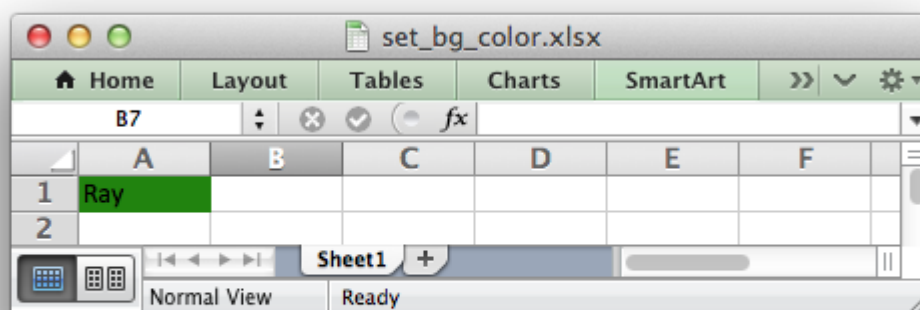

The color can be a Html style #RRGGBB string or a limited number of named colors, see *Working with Colors*.

9.21 format:set_fg_color()

set_fg_color(*color*)
      Set the color of the foreground pattern in a cell.

          Parameters color – The cell font color.

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

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

9.22 format:set_border()

set_border(*style*)
      Set the cell border style.

          Parameters style – 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 Excel index number:

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

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

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

9.23 format:set_bottom()

set_bottom(*style*)
    Set the cell bottom border style.

        Parameters style – Border style index. Default is 1.

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

9.24 format:set_top()

set_top(*style*)
    Set the cell top border style.

        Parameters style – Border style index. Default is 1.

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

9.25 format:set_left()

set_left(*style*)
    Set the cell left border style.

        Parameters style – Border style index. Default is 1.

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

9.26 format:set_right()

set_right(*style*)
    Set the cell right border style.

        Parameters style – Border style index. Default is 1.

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

9.27 format:set_border_color()

set_border_color(*color*)
>   Set the color of the cell border.

>>   **Parameters color** – The cell border color.

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

- set_bottom_color()
- set_top_color()
- set_left_color()
- set_right_color()

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

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

9.28 format:set_bottom_color()

set_bottom_color(*color*)
>   Set the color of the bottom cell border.

>>   **Parameters color** – The cell border color.

See set_border_color() for details on the border colors.

9.29 format:set_top_color()

set_top_color(*color*)
>   Set the color of the top cell border.

>>   **Parameters color** – The cell border color.

See set_border_color() for details on the border colors.

9.30 format:set_left_color()

set_left_color(*color*)
>   Set the color of the left cell border.

>>   **Parameters color** – The cell border color.

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

9.31 format:set_right_color()

set_right_color(*color*)
    Set the color of the right cell border.

> **Parameters color** – The cell border color.

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

WORKING WITH FORMATS

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

10.1 Creating and using a Format object

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

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

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

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

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

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

10.2 Format methods and Format properties

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

Category	Description	Property	Method Name
Font	Font type	font_name	set_font_name()
	Font size	font_size	set_font_size()
	Font color	font_color	set_font_color()
Continued on next page			

Table 10.1 – continued from previous page

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

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

```
format = workbook:add_format()
format:set_bold()
format:set_font_color("red")
```

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

```
format = workbook:add_format({bold = true, font_color = "red"})
```

The object method interface is mainly provided for backward compatibility. The key/value interface has proved to be more flexible in real world programs and is the recommended method for setting format properties.

It is also possible, as with any Lua function that takes a table as its only parameter to use the following shorthand syntax:

format = workbook:add_format{bold = true, font_color = "red"}

10.3 Format Colors

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

```
format1:set_font_color("#FF0000")
format2:set_font_color("red")
```

See *Working with Colors* for more details.

10.4 Format Defaults

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

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

```
format = workbook:add_format()

format:set_bold()   -- Turns bold on.
```

10.5 Modifying Formats

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

```
format = workbook:add_format({bold - true, font_color = "red"})
worksheet:write("A1", "Cell A1", format)

-- Later...
format:set_font_color("green")
worksheet:write("B1", "Cell B1", format)
```

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

WORKING WITH CELL NOTATION

Xlsxwriter.lua 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 = 0, 5 do
    worksheet:write(row, 0, "Hello")
end
```

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

```
worksheet:write("H1", 200)
worksheet:write("H2", "=H1+1")
```

In general when using the `xlsxwriter` module you can use A1 notation anywhere you can use row-column notation.

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

11.1 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.

CELL UTILITY FUNCTIONS

The `xlsxwriter.utility` module contains several helper functions for dealing with A1 notation. These functions can be imported and used as follows:

```
local Utility = require "xlsxwriter.utility"

cell = Utilty.rowcol_to_cell(1, 2) --> C2
```

The available functions are shown below.

12.1 rowcol_to_cell()

rowcol_to_cell(*row*, *col*)
> Convert a zero indexed row and column cell reference to a A1 style string.

>> **Parameters**

>>> • **row** – The cell row.

>>> • **col** – The cell column.

>> **Return type** A1 style string.

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

```
cell = Utilty.rowcol_to_cell(0, 0) --> A1
cell = Utilty.rowcol_to_cell(0, 1) --> B1
cell = Utilty.rowcol_to_cell(1, 0) --> A2
```

12.2 rowcol_to_cell_abs()

rowcol_to_cell_abs(*row*, *col*[, *row_abs*, *col_abs*])
> Convert a zero indexed row and column cell reference to a A1 style string.

>> **Parameters**

>>> • **row** – The cell row.

- **col** – The cell column.

- **row_abs** – Optional flag to make the row absolute.

- **col_abs** – Optional flag to make the column absolute.

> **Return type** A1 style string.

The `rowcol_to_cell_abs()` function is like the `rowcol_to_cell_abs()` function but the optional parameters `row_abs` and `col_abs` can be used to indicate that the row or column is absolute:

```
str = Utilty.rowcol_to_cell_abs(0, 0, false, true) --> $A1
str = Utilty.rowcol_to_cell_abs(0, 0, true        ) --> A$1
str = Utilty.rowcol_to_cell_abs(0, 0, true,  true) --> $A$1
```

12.3 cell_to_rowcol()

cell_to_rowcol(*cell_str*)
> Convert a cell reference in A1 notation to a zero indexed row and column.

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

> **Return type** row, col.

The `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 = Utilty.cell_to_rowcol("A1")   --> (0, 0)
row, col = Utilty.cell_to_rowcol("B1")   --> (0, 1)
row, col = Utilty.cell_to_rowcol("C2")   --> (1, 2)
row, col = Utilty.cell_to_rowcol("$C2")  --> (1, 2)
row, col = Utilty.cell_to_rowcol("C$2")  --> (1, 2)
row, col = Utilty.cell_to_rowcol("$C$2") --> (1, 2)
```

12.4 col_to_name()

col_to_name(*col*[, *col_abs*])
> Convert a zero indexed column cell reference to a string.

> **Parameters**

> - **col** – The cell column.

> - **col_abs** – Optional flag to make the column absolute.

> **Return type** Column style string.

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

```
column = Utilty.col_to_name(0)    --> A
column = Utilty.col_to_name(1)    --> B
column = Utilty.col_to_name(702)  --> AAA
```

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

```
column = Utilty.col_to_name(0, false)  --> A
column = Utilty.col_to_name(0, true)   --> $A
column = Utilty.col_to_name(1, true)   --> $B
```

12.5 range()

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** – The first cell row.
>
> - **first_col** – The first cell column.
>
> - **last_row** – The last cell row.
>
> - **last_col** – The last cell column.
>
> **Return type** A1:B1 style range string.

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

```
cell_range = Utilty.range(0, 0, 9, 0) --> A1:A10
cell_range = Utilty.range(1, 2, 8, 2) --> C2:C9
cell_range = Utilty.range(0, 0, 3, 4) --> A1:E4
```

12.6 range_abs()

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

```
cell_range = Utilty.range_abs(0, 0, 9, 0) --> $A$1:$A$10
cell_range = Utilty.range_abs(1, 2, 8, 2) --> $C$2:$C$9
cell_range = Utilty.range_abs(0, 0, 3, 4) --> $A$1:$E$4
```

WORKING WITH DATES AND TIME

Dates and times in Excel are represented by real numbers. For example a date that is displayed in Excel as "Jan 1 2013 12:00 PM" is stored as the number 41275.5.

The integer part of the number stores the number of days since the epoch, which is generally 1900, 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:

```lua
local Workbook = require "xlsxwriter.workbook"

local workbook  = Workbook:new("date_examples.xlsx")
local worksheet = workbook:add_worksheet()

-- Widen the first column or extra visibility.
worksheet:set_column("A:A", 30)

-- A number to convert to a date.
local number = 41333.5

-- Write it as a number without formatting.
worksheet:write("A1", number)          --> 41333.5

local format2 = workbook:add_format({num_format = "dd/mm/yy"})
worksheet:write("A2", number, format2) --> 28/02/13

local format3 = workbook:add_format({num_format = "mm/dd/yy"})
worksheet:write("A3", number, format3) --> 02/28/13

local format4 = workbook:add_format({num_format = "d-m-yyyy"})
worksheet:write("A4", number, format4) --> 28-2-2013

local format5 = workbook:add_format({num_format = "dd/mm/yy hh:mm"})
worksheet:write("A5", number, format5) --> 28/02/13 12:00

local format6 = workbook:add_format({num_format = "d mmm yyyy"})
worksheet:write("A6", number, format6) --> 28 Feb 2013

local format7 = workbook:add_format({num_format = "mmm d yyyy hh:mm AM/PM"})
worksheet:write("A7", number, format7) --> Feb 28 2008 12:00 PM
```

```
workbook:close()
```
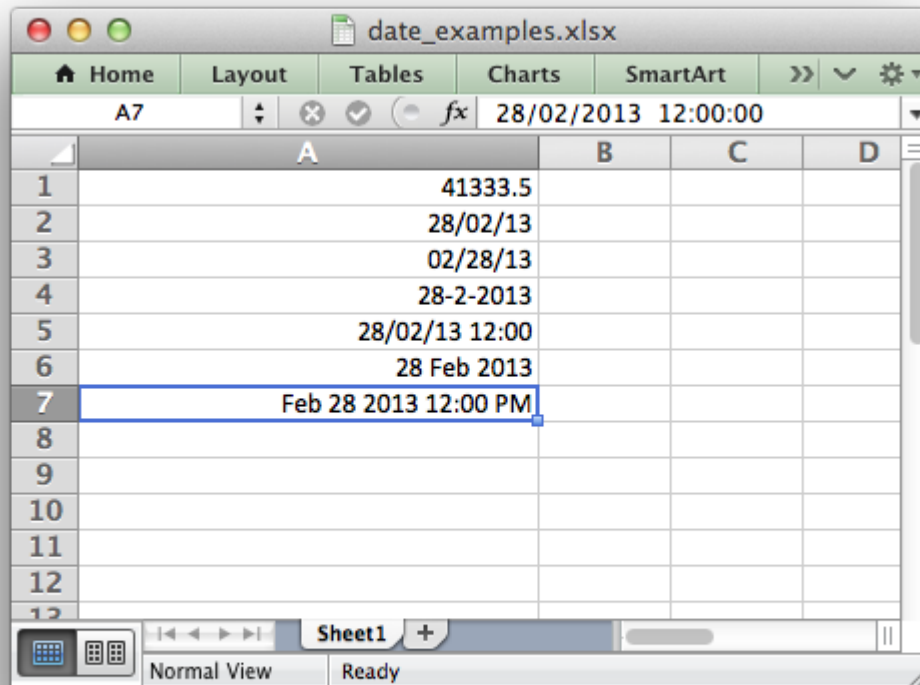

To make working with dates and times a little easier the `xlsxwriter` module provides two date handling methods: `write_date_time()` and `write_date_string()`.

The `write_date_time()` method takes a table of values like those used for os.time()

```
date_format = workbook:add_format({num_format = "d mmmm yyyy"})

worksheet:write_date_time("A1", {year = 2014, month = 3, day = 17}, date_format)
```

The allowable table keys and values are:

Key	Value
year	4 digit year
month	1 - 12
day	1 - 31
hour	0 - 23
min	0 - 59
sec	0 - 59.999

The `write_date_string()` method takes a string in an ISO8601 format:

```
yyyy-mm-ddThh:mm:ss.sss
```

This conforms to an ISO8601 date but it should be noted that the full range of ISO8601 formats are not supported. The following variations are permitted:

```
yyyy-mm-ddThh:mm:ss.sss          -- Standard format.
yyyy-mm-ddThh:mm:ss.sssZ         -- Additional Z (but not time zones).
yyyy-mm-dd                       -- Date only, no time.
          hh:mm:ss.sss           -- Time only, no date.
          hh:mm:ss               -- No fractional seconds.
```

Note that the T is required for cases with both date, and time and seconds are required for all times.

Here is an example using write_date_string():

```
date_format = workbook:add_format({num_format = "d mmmm yyyy"})

worksheet:write_date_string("A1", "2014-03-17", date_format)
```

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

```lua
local Workbook = require "xlsxwriter.workbook"

local workbook  = Workbook:new("datetimes.xlsx")
local worksheet = workbook:add_worksheet()
local bold      = workbook:add_format({bold = true})

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

-- Write the column headers.
worksheet:write("A1", "Formatted date", bold)
worksheet:write("B1", "Format",         bold)

-- Create an ISO8601 style date string to use in the examples.
local date_string = "2013-01-23T12:30:05.123"

-- Examples date and time formats. In the output file compare how changing
-- the format codes change the appearance of the date.
local 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",
```

```lua
    "hh:mm:ss.000",
}

-- Write the same date and time using each of the above formats.
for row, date_format_str in ipairs(date_formats) do

  -- Create a format for the date or time.
  local date_format = workbook:add_format({num_format = date_format_str,
                                            align = "left"})

  -- Write the same date using different formats.
  worksheet:write_date_string(row, 0, date_string, date_format)

  -- Also write the format string for comparison.
  worksheet:write_string(row, 1, date_format_str)

end

workbook:close()
```
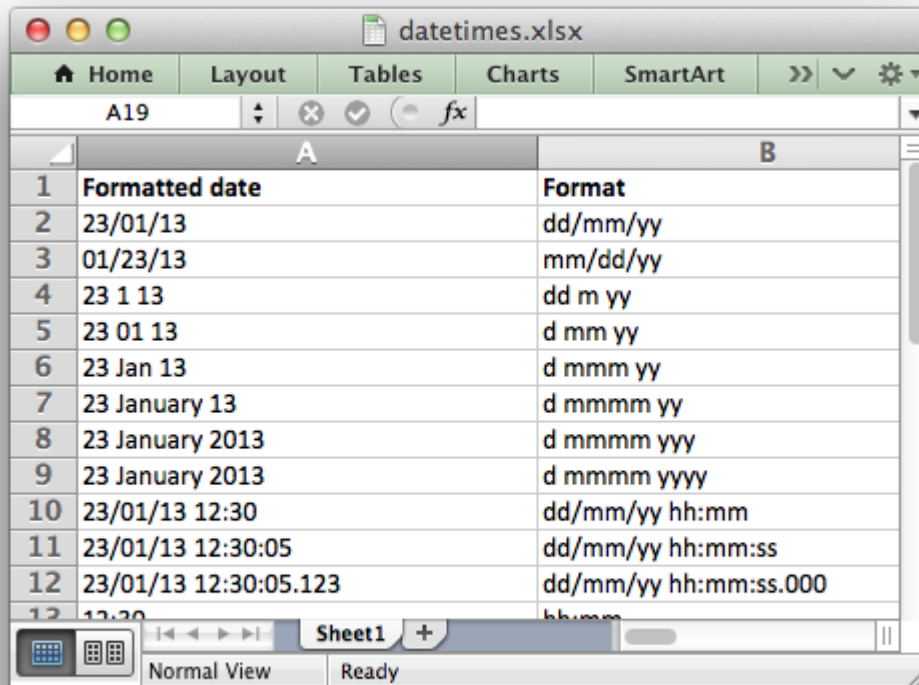
WORKING WITH COLORS

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

```
format:set_font_color('#FF0000')
```

For convenience a limited number of color names are supported:

```
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 MEMORY AND PERFORMANCE

By default `xlsxwriter` holds all cell data in memory. This is to allow future features where formatting is applied separately from the data.

The effect of this is that for large files `xlsxwriter` can consume a lot of memory and it is even possible to run out of memory.

Fortunately, this memory usage can be reduced almost completely by setting the *Workbook:new()* `'constant_memory'` property:

```
workbook = Workbook:new(filename, {constant_memory = true})
```

The optimisation works by flushing each row after a subsequent row is written. In this way the largest amount of data held in memory for a worksheet is the amount of memory required to hold a single row of data.

Since each new row flushes the previous row, data must be written in sequential row order when `'constant_memory'` mode is on:

```
-- With 'constant_memory' you must write data in row column order.
for row = 0, row_max do
  for col = 0, col_max do
    worksheet:write(row, col, some_data)
  end
end

-- With 'constant_memory' the following would only write the first column.
for col = 0, col_max do  -- !!
  for row = 0, row_max do
    worksheet:write(row, col, some_data)
  end
end
```

Another optimisation that is used to reduce memory usage is that cell strings aren't stored in an Excel structure call "shared strings" and instead are written "in-line". This is a documented Excel feature that is supported by most spreadsheet applications. One known exception is Apple Numbers for Mac where the string data isn't displayed.

The trade-off when using `'constant_memory'` mode is that you won't be able to take advantage of any features that manipulate cell data after it is written. Currently there aren't any such features.

For larger files `'constant_memory'` mode also gives an increase in execution speed, see below.

15.1 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	0.20	2071819
400	50	0.40	4149803
800	50	0.86	8305771
1600	50	1.87	16617707
3200	50	3.84	33271579
6400	50	8.02	66599323
12800	50	16.54	133254811

Xlsxwriter in `constant_memory` mode: the execution time still increases linearly with the number of rows but the memory usage remains small and mainly constant:

Rows	Columns	Time (s)	Memory (bytes)
200	50	0.18	41119
400	50	0.36	24735
800	50	0.69	24735
1600	50	1.41	24735
3200	50	2.83	41119
6400	50	5.83	41119
12800	50	11.29	24735

These figures were generated using the `perf_tester.lua` program in the `examples` directory of the xlsxwriter repo.

Note, there will be further optimisation in both modes in later releases.

SIXTEEN

EXAMPLES

The following are some of the examples included in the examples directory of the xlsxwriter distribution.

16.1 Example: Hello World

The simplest possible spreadsheet. This is a good place to start to see if the xlsxwriter module is installed correctly.
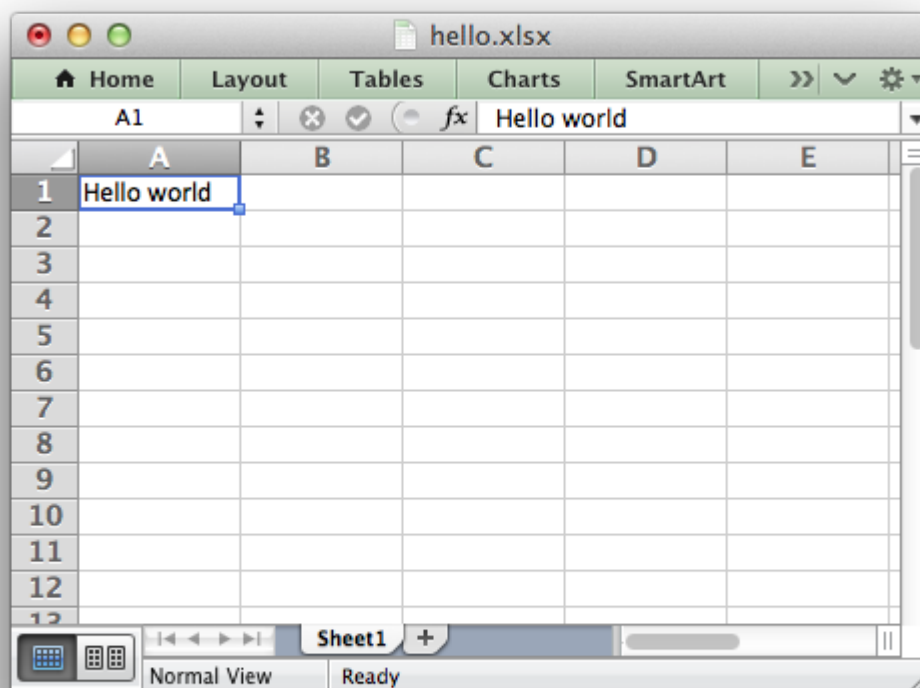
```
----
--
-- A hello world spreadsheet using the xlsxwriter.lua module.
--
-- Copyright 2014, John McNamara, jmcnamara@cpan.org
--

local Workbook = require "xlsxwriter.workbook"

local workbook  = Workbook:new("hello_world.xlsx")
local worksheet = workbook:add_worksheet()

worksheet:write("A1", "Hello world")

workbook:close()
```

16.2 Example: Simple Feature Demonstration

This program is an example of writing some of the features of the `xlsxwriter` module.


```
----
--
```
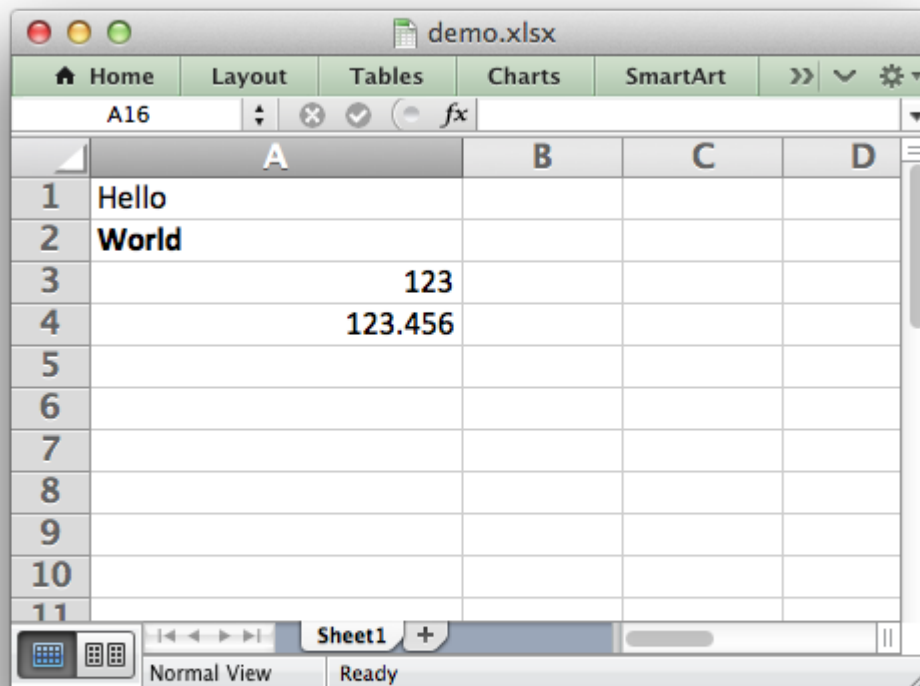
```lua
-- A simple example of some of the features of the xlsxwriter.lua module.
--
-- Copyright 2014, John McNamara, jmcnamara@cpan.org
--

local Workbook = require "xlsxwriter.workbook"

local workbook  = Workbook:new("demo.xlsx")
local 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.
local 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)

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*.

16.3 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 the xlsxwriter.lua module to write
-- simple array formulas.
--
-- Copyright 2014, John McNamara, jmcnamara@cpan.org
--

local Workbook = require "xlsxwriter.workbook"

-- Create a new workbook and add a worksheet
local workbook  = Workbook:new("array_formula.xlsx")
local 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 explicit.
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()
```

16.4 Example: Merging Cells

This program is an example of merging cells in a worksheet. See the `merge_range()` method for more details.
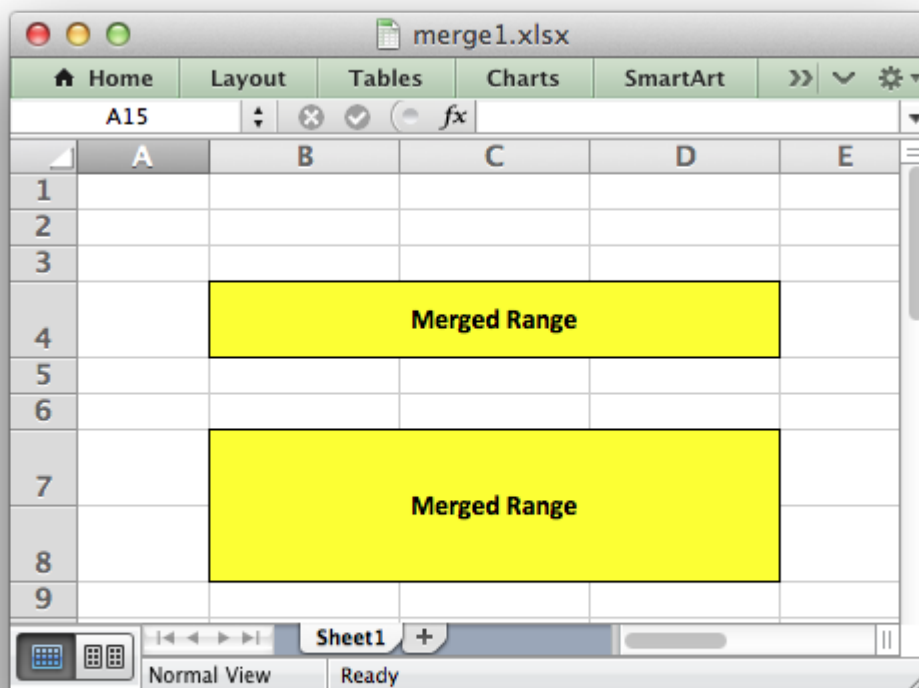

```
----
--
-- A simple example of merging cells with the xlsxwriter Lua module.
--
-- Copyright 2014, John McNamara, jmcnamara@cpan.org
--
```

```lua
local Workbook = require "xlsxwriter.workbook"

local workbook  = Workbook:new("merge1.xlsx")
local 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      = true,
    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()
```

16.5 Example: Adding Defined Names

Example of how to addiing defined names to a workbook.

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. These can then be used in formulas or anywhere a range is used.
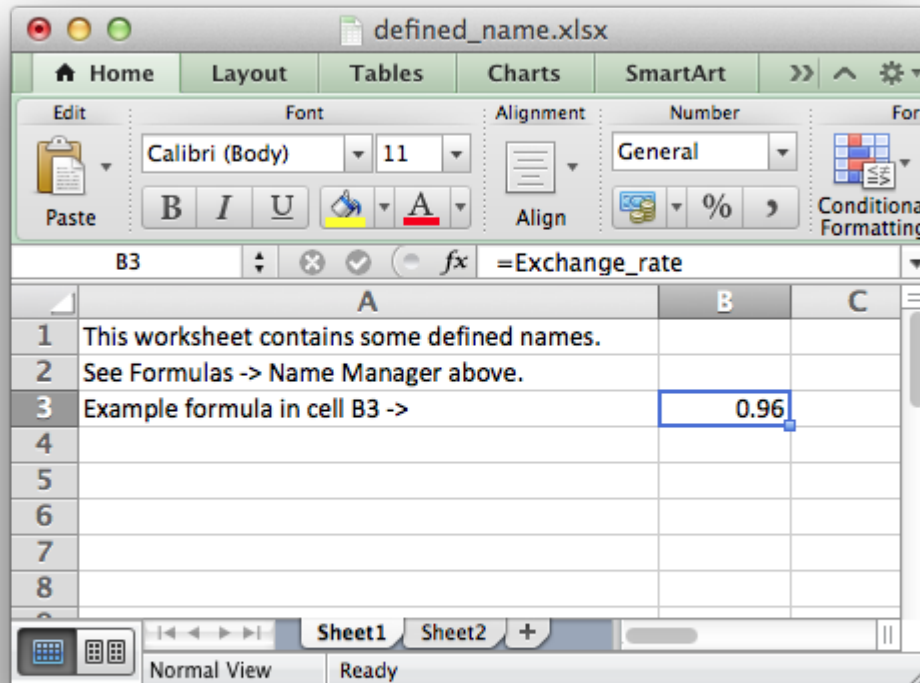
```
----
--
-- Example of how to create defined names with the xlsxwriter.lua module.
--
-- This method is used to define a user friendly name to represent a value,
-- a single cell or a range of cells in a workbook.
--
-- Copyright 2014, John McNamara, jmcnamara@cpan.org
--

local Workbook = require "xlsxwriter.workbook"

local workbook   = Workbook:new("defined_name.xlsx")
local worksheet1 = workbook:add_worksheet()
local 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 ipairs(workbook:worksheets()) do
```

```
    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")
end

workbook:close()
```

16.6 Example: Write UTF-8 Strings

An example of writing simple UTF-8 strings to a worksheet.

Unicode strings in Excel must be UTF-8 encoded. With `xlsxwriter` all that is required is that the source file is UTF-8 encoded and Lua will handle the UTF-8 strings like any other strings:
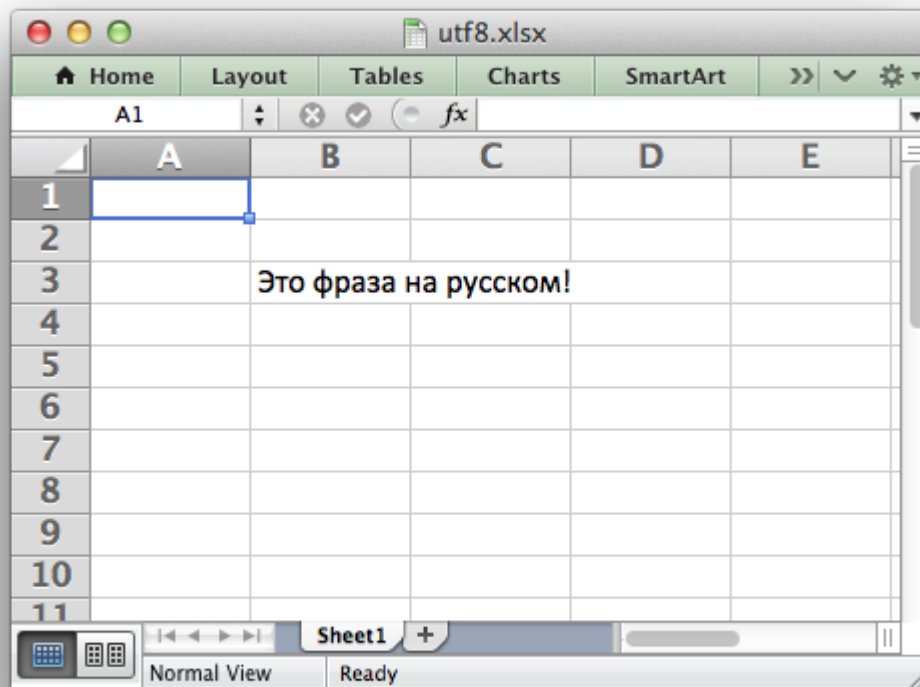
16.7 Example: Convert a UTF-8 file to a Worksheet

This program is an example of reading in data from a UTF-8 encoded text file and converting it to a worksheet.
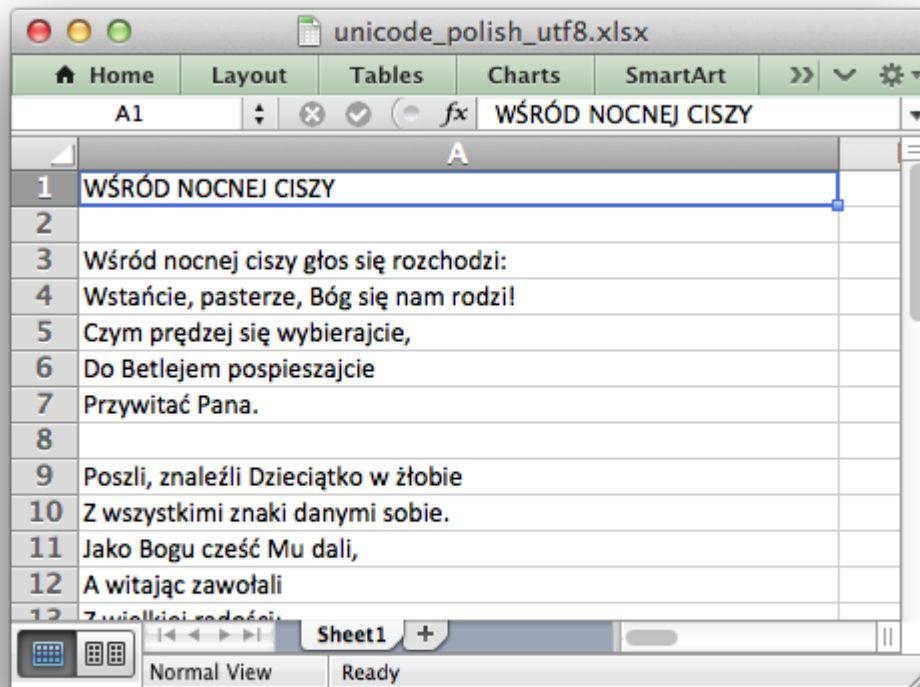

```
----
--
-- A simple example of converting some Unicode text to an Excel file using
-- the xlsxwriter.lua module.
--
-- This example generates a spreadsheet with some Polish text from a file
-- with UTF-8 encoded text.
--
-- Copyright 2014, John McNamara, jmcnamara@cpan.org
--

local Workbook = require "xlsxwriter.workbook"

local workbook  = Workbook:new("utf8_polish.xlsx")
local worksheet = workbook:add_worksheet()

-- Widen the first column to make the text clearer.
worksheet:set_column("A:A", 50)
```

```lua
    -- Open a source of UTF-8 data.
    local file = assert(io.open("utf8_polish.txt", "r"))

    -- Read the text file and write it to the worksheet.
    local line = file:read("*l")
    local row = 0

    while line do
      -- Ignore comments in the text file.
      if not string.match(line, "^#") then
        worksheet:write(row, 0, line)
        row = row + 1
      end
      line = file:read("*l")
    end

    workbook:close()
```

16.8 Example: Setting Document Properties

This program is an example setting document properties. See the `set_properties()` workbook method for more details.
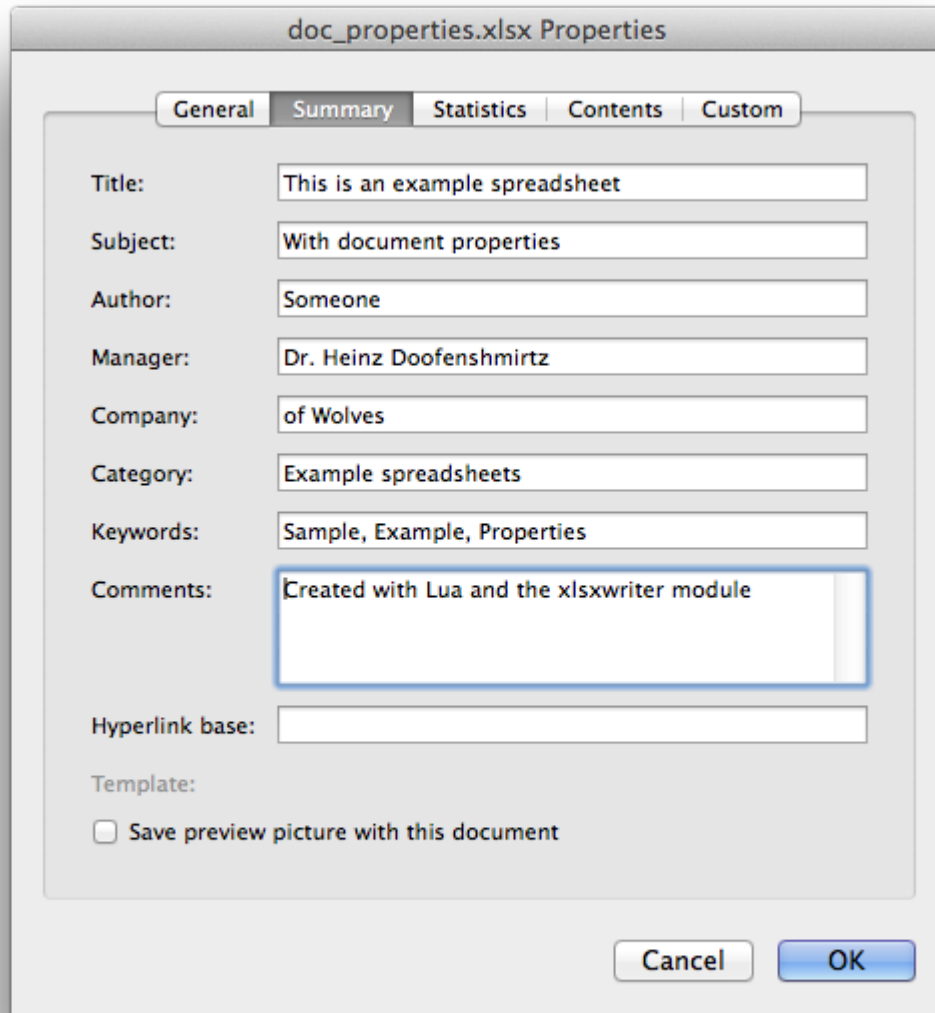
```
----
--
-- An example of adding document properites to a xlsxwriter.lua file.
--
-- Copyright 2014, John McNamara, jmcnamara@cpan.org
--

local Workbook = require "xlsxwriter.workbook"

local workbook  = Workbook:new("doc_properties.xlsx")
local worksheet = workbook:add_worksheet()

workbook:set_properties({
    title    = "This is an example spreadsheet",
```

```
        subject   = "With document properties",
        author    = "Someone",
        manager   = "Dr. Heinz Doofenshmirtz",
        company   = "of Wolves",
        category  = "Example spreadsheets",
        keywords  = "Sample, Example, Properties",
        comments  = "Created with Lua and the xlsxwriter module",
        status    = "Quo",
    })

    worksheet:set_column("A:A", 70)
    worksheet:write("A1", "Select 'Workbook Properties' to see properties.")

    workbook:close()
```

16.9 Example: Setting Worksheet Tab Colours

This program is an example of setting worksheet tab colours. See the set_tab_color() method for more details.
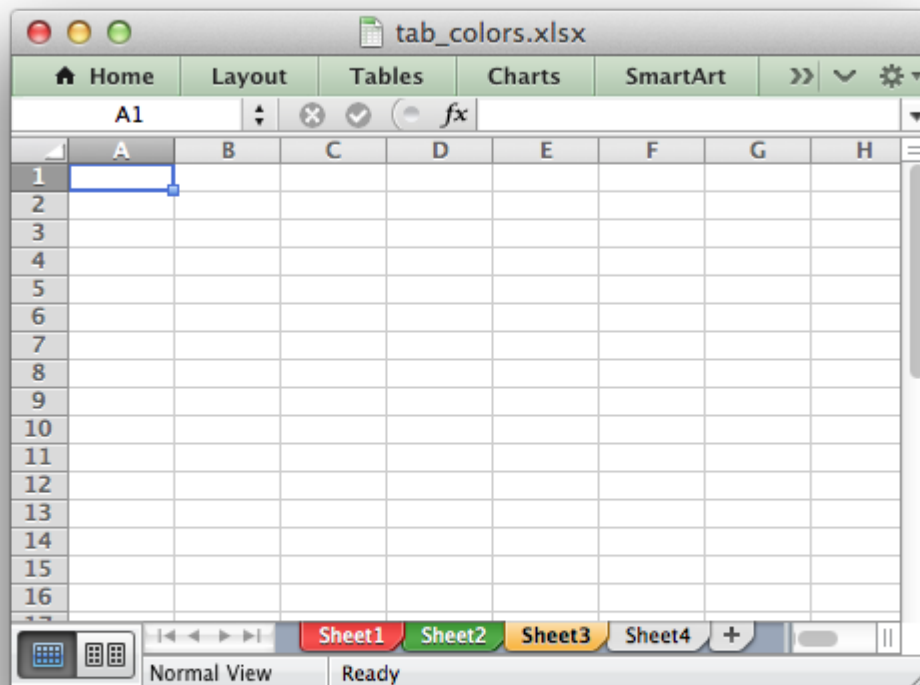
```lua
----
--
-- Example of how to set Excel worksheet tab colours using
-- the Xlsxwriter.lua module.
--
-- Copyright 2014, John McNamara, jmcnamara@cpan.org
--

local Workbook = require "xlsxwriter.workbook"

local workbook = Workbook:new("tab_colors.xlsx")

-- Set up some worksheets.
local worksheet1 = workbook:add_worksheet()
local worksheet2 = workbook:add_worksheet()
local worksheet3 = workbook:add_worksheet()
local worksheet4 = workbook:add_worksheet()

-- Set tab colours, worksheet4 will have the default colour.
worksheet1:set_tab_color("red")
worksheet2:set_tab_color("green")
worksheet3:set_tab_color("#FF9900")

workbook:close()
```

16.10 Example: Hiding Worksheets

This program is an example of how to hide a worksheet using the `hide()` method.
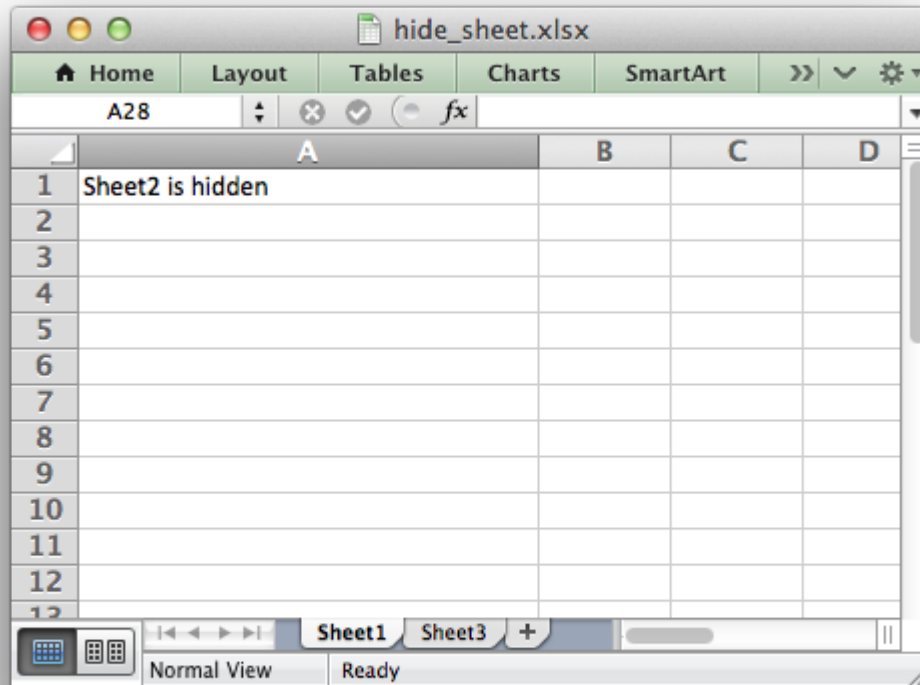
```
----
--
-- Example of how to hide a worksheet with xlsxwriter.lua.
--
-- Copyright 2014, John McNamara, jmcnamara@cpan.org
--

local Workbook = require "xlsxwriter.workbook"

local workbook = Workbook:new("hide_sheet.xlsx")

local worksheet1 = workbook:add_worksheet()
local worksheet2 = workbook:add_worksheet()
local worksheet3 = workbook:add_worksheet()

worksheet1:set_column("A:A", 30)
worksheet2:set_column("A:A", 30)
worksheet3:set_column("A:A", 30)

-- Hide Sheet2. It won't be visible until it is unhidden in Excel.
worksheet2:hide()

worksheet1:write("A1", "Sheet2 is hidden")
worksheet2:write("A1", "Now it's my turn to find you!")
worksheet3:write("A1", "Sheet2 is hidden")
```

```
    workbook:close()
```

16.11 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.
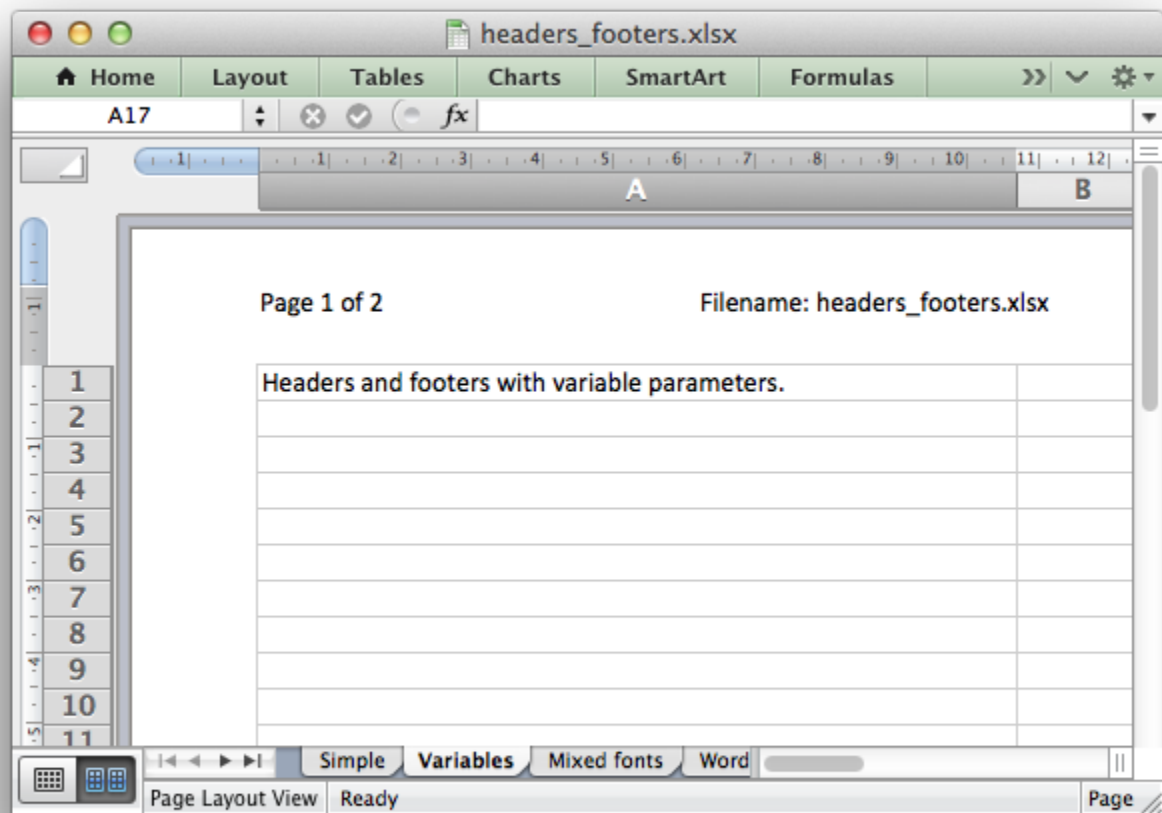

```
----
--
-- This program shows several examples of how to set up headers and
-- footers with xlsxwriter.
--
-- The control characters used in the header/footer strings are:
--
--      Control             Category            Description
--      =======             ========            ===========
--      &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
--
--      &&                     Miscellaneous       Literal ampersand &
--
-- See the main XlsxWriter documentation for more information.
--
-- Copyright 2014, John McNamara, jmcnamara@cpan.org
--

local Workbook = require "xlsxwriter.workbook"

local workbook  = Workbook:new("headers_footers.xlsx")

----
--
-- A simple example to start
--
local worksheet1 = workbook:add_worksheet("Simple")
local header1    = "&CHere is some centred text."
local footer1    = "&LHere is some left aligned text."

worksheet1:set_header(header1)
worksheet1:set_footer(footer1)
worksheet1:set_page_view()

worksheet1:set_column("A:A", 50)
worksheet1:write("A1", "Headers and footers added.")

----
--
-- This is an example of some of the header/footer variables.
--
local worksheet2 = workbook:add_worksheet("Variables")
local header2    = "&LPage &P of &N" .. "&CFilename: &F" .. "&RSheetname: &A"
local footer2    = "&LCurrent date: &D" .. "&RCurrent time: &T"

worksheet2:set_header(header2)
```

```lua
worksheet2:set_footer(footer2)
worksheet2:set_page_view()

worksheet2:set_column("A:A", 50)
worksheet2:write("A1", "Headers and footers with variable parameters.")
worksheet2:write("A20", "Page break inserted here.")
worksheet2:write("A21", "Next sheet")
worksheet2:set_h_pagebreaks({20})

----
--
-- This example shows how to use more than one font
--
local worksheet3 = workbook:add_worksheet("Mixed fonts")
local header3    = '&C&"Courier New,Bold"Hello &"Arial,Italic"World'
local footer3    = '&C&"Symbol"e&"Arial" = mc&X2'

worksheet3:set_header(header3)
worksheet3:set_footer(footer3)
worksheet3:set_page_view()

worksheet3:set_column("A:A", 50)
worksheet3:write("A1", "Headers and footers with mixed fonts.")

----
--
-- Example of line wrapping
--
local worksheet4 = workbook:add_worksheet("Word wrap")
local header4    = "&CHeading 1\nHeading 2"

worksheet4:set_header(header4)
worksheet4:set_page_view()

worksheet4:set_column("A:A", 50)
worksheet4:write("A1", "Header with wrapped text.")

----
--
-- Example of inserting a literal ampersand &
--
local worksheet5 = workbook:add_worksheet("Ampersand")
local header5    = "&CCuriouser && Curiouser - Attorneys at Law"

worksheet5:set_header(header5)
worksheet5:set_page_view()

worksheet5:set_column("A:A", 50)
worksheet5:write("A1", "Header with an ampersand.")

workbook:close()
```

16.12 Example: Indenting Text in a Cell

This program is an example indenting text in a cell.
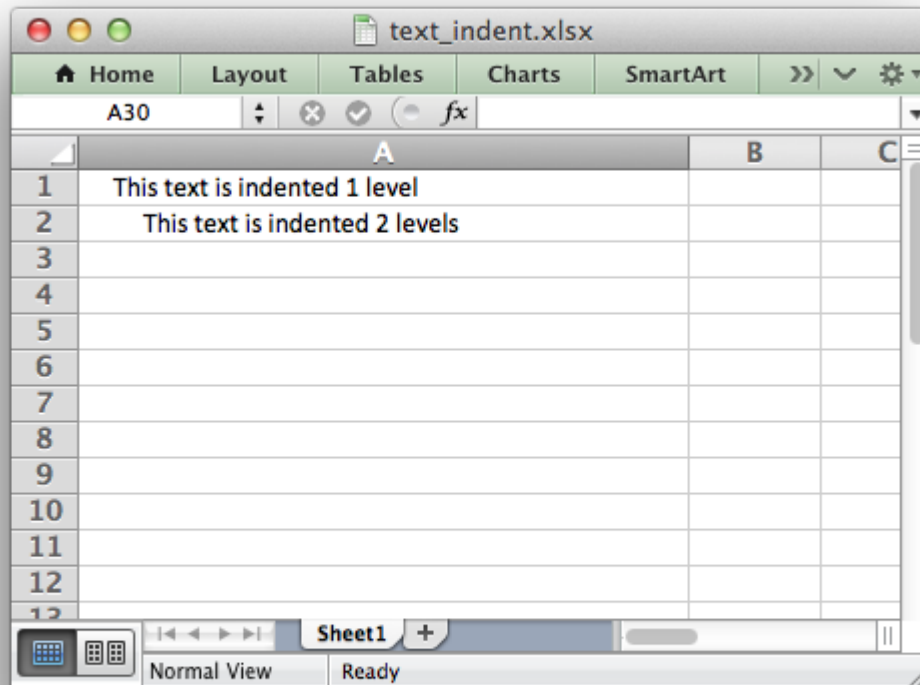

```lua
----
--
-- An example of indenting text in a cell using the xlsxwriter.lua module.
--
-- Copyright 2014, John McNamara, jmcnamara@cpan.org
--

local Workbook = require "xlsxwriter.workbook"

local workbook  = Workbook:new("text_indent.xlsx")
local worksheet = workbook:add_worksheet()

local indent1 = workbook:add_format({indent = 1})
local indent2 = workbook:add_format({indent = 2})

worksheet:set_column("A:A", 40)

worksheet:write("A1", "This text is indented 1 level",  indent1)
worksheet:write("A2", "This text is indented 2 levels", indent2)
```

```
workbook:close()
```

KNOWN ISSUES AND BUGS

This section lists known issues and bugs and gives some information on how to submit bug reports.

17.1 Content is Unreadable. Open and Repair

Very, very occasionally you may see an Excel warning when opening an `xlsxwriter` file like:

> Excel could not open file.xlsx because some content is unreadable. Do you want to open and repair this workbook.

This ominous sounding message is Excel's default warning for any validation error in the XML used for the components of the XLSX file.

If you encounter an issue like this you should open an issue on GitHub with a program to replicate the issue (see below) or send one of the failing output files to the *Author*.

17.2 Formulas displayed as #NAME? until edited

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

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

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

17.3 Formula results displaying as zero in non-Excel applications

Due to wide range of possible formulas and interdependencies between them, `xlsxwriter` doesn't, and realistically cannot, calculate the result of a formula when it is written to an XLSX file. Instead, it stores the value 0 as the formula result. It then sets a global flag in the XLSX file to say that all formulas and functions should be recalculated when the file is opened.

This is the method recommended in the Excel documentation and in general it works fine with spreadsheet applications. However, applications that don't have a facility to calculate formulas, such as Excel Viewer, or several mobile applications, will only display the 0 results.

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

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

17.4 Strings aren't displayed in Apple Numbers in 'constant_memory' mode

In `Workbook()` `'constant_memory'` mode `xlsxwriter` uses an optimisation where cell strings aren't stored in an Excel structure call "shared strings" and instead are written "in-line".

This is a documented Excel feature that is supported by most spreadsheet applications. One known exception is Apple Numbers for Mac where the string data isn't displayed.

17.5 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.

REPORTING BUGS

Here are some tips on reporting bugs in `xlsxwriter`.

18.1 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:

```
lua -e 'W = require "xlsxwriter.workbook"; print(W.version)'
```

Check the *Changes in XlsxWriter* section to see what has changed in the latest versions.

18.2 Read the documentation

Read or search the `xlsxwriter` documentation to see if the issue you are encountering is already explained.

18.3 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.

18.4 Use the xlsxwriter Issue tracker on GitHub

The xlsxwriter issue tracker is on GitHub.

18.5 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 analyse and respond to the bug report more quickly.

Issue with SOMETHING

I am using xlsxwriter to do SOMETHING but it appears to do SOMETHING ELSE.

I am using Lua version X.Y and xlsxwriter x.y.z.

Here is some code that demonstrates the problem:

```lua
local Workbook = require "xlsxwriter.workbook"

local workbook  = Workbook:new("hello_world.xlsx")
local worksheet = workbook:add_worksheet()

worksheet:write("A1", "Hello world")

workbook:close()
```

FREQUENTLY ASKED QUESTIONS

The section outlines some answers to frequently asked questions.

19.1 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.

19.2 Q. Why do my formulas show a zero result in some, non-Excel applications?

Due to wide range of possible formulas and interdependencies between them `xlsxwriter` doesn't, and realistically cannot, calculate the result of a formula when it is written to an XLSX file. Instead, it stores the value 0 as the formula result. It then sets a global flag in the XLSX file to say that all formulas and functions should be recalculated when the file is opened.

This is the method recommended in the Excel documentation and in general it works fine with spreadsheet applications. However, applications that don't have a facility to calculate formulas, such as Excel Viewer, or several mobile applications, will only display the 0 results.

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

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

19.3 Q. Can I apply a format to a range of cells in one go?

Currently no. However, it is a planned features to allow cell formats and data to be written separately.

19.4 Q. Is feature X supported or will it be supported?

All supported features are documented. In time the feature set should expand to be the same as the Python XlsxWriter module.

19.5 Q. Is there an "AutoFit" option for columns?

Unfortunately, there is no way to specify "AutoFit" for a column in the Excel file format. This feature is only available at runtime from within Excel. It is possible to simulate "AutoFit" by tracking the width of the data in the column as your write it.

19.6 Q. Do people actually ask these questions frequently, or at all?

Apart from this question, yes.

CHANGES IN XLSXWRITER

This section shows changes and bug fixes in the XlsxWriter module.

20.1 Release 0.0.4 - April 14 2014

- Added the `define_name()` method to create defined names and ranges in a workbook or worksheet.
- Added the `set_properties()` workbook method for setting document properties.
- Added the `worksheets()` method as an accessor for the worksheets in a workbook.

20.2 Release 0.0.3 - April 9 2014

- Added the `merge_range()` method to merge worksheet cells.

20.3 Release 0.0.2 - April 6 2014

- Added *The Worksheet Class (Page Setup)* methods.

20.4 Release 0.0.1 - March 29 2014

- First public release.

AUTHOR

XlsxWriter was written by John McNamara.

- GitHub
- Twitter @jmcnamara13
- Coderwall
- Ohloh

You can contact me at jmcnamara@cpan.org.

21.1 Donations

If you would like to donate to the xlsxwriter project to keep it active or to pay for the PDF copy of the documentation you can do so via PayPal.

LICENSE

Copyright (c) 2014, John McNamara <jmcnamara@cpan.org>

The MIT/X11 License.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.