openpyxl Documentation

Release 2.4.9

See AUTHORS

Nov 18, 2017

Contents

Author Eric Gazoni, Charlie Clark

Source code http://bitbucket.org/openpyxl/openpyxl/src

Issues http://bitbucket.org/openpyxl/openpyxl/issues

Generated Nov 18, 2017

Version 2.4.9

CHAPTER 1

Introduction

Openpyxl is a Python library for reading and writing Excel 2010 xlsx/xlsm/xltx/xltm files.

It was born from lack of existing library to read/write natively from Python the Office Open XML format.

All kudos to the PHPExcel team as openpyxl was initially based on PHPExcel.

1.1 Support

This is an open source project, maintained by volunteers in their spare time. This may well mean that particular features or functions that you would like are missing. But things don't have to stay that way. You can contribute the project *Development* yourself or contract a developer for particular features.

Professional support for openpyxl is available from Clark Consulting & Research and Adimian. Donations to the project to support further development and maintenance are welcome.

Bug reports and feature requests should be submitted using the issue tracker. Please provide a full traceback of any error you see and if possible a sample file. If for reasons of confidentiality you are unable to make a file publicly available then contact of one the developers.

1.2 Sample code:

```
from openpyxl import Workbook
wb = Workbook()

# grab the active worksheet
ws = wb.active

# Data can be assigned directly to cells
ws['A1'] = 42

# Rows can also be appended
```

```
ws.append([1, 2, 3])

# Python types will automatically be converted
import datetime
ws['A2'] = datetime.datetime.now()

# Save the file
wb.save("sample.xlsx")
```

CHAPTER 2

User List

Official user list can be found on http://groups.google.com/group/openpyxl-users

CHAPTER 3

How to Contribute Code

Any help will be greatly appreciated, just follow those steps:

1. Please start a new fork (https://bitbucket.org/openpyxl/openpyxl/fork) for each independent feature, don't try to fix all problems at the same time, it's easier for those who will review and merge your changes ;-)

2. Hack hack hack

3. Don't forget to add unit tests for your changes! (YES, even if it's a one-liner, changes without tests will **not** be accepted.) There are plenty of examples in the source if you lack know-how or inspiration.

4. If you added a whole new feature, or just improved something, you can be proud of it, so add yourself to the AUTHORS file :-)

5. Let people know about the shiny thing you just implemented, update the docs!

6. When it's done, just issue a pull request (click on the large "pull request" button on *your* repository) and wait for your code to be reviewed, and, if you followed all theses steps, merged into the main repository.

For further information see *Development*

CHAPTER 4

Other ways to help

There are several ways to contribute, even if you can't code (or can't code well):

- triaging bugs on the bug tracker: closing bugs that have already been closed, are not relevant, cannot be reproduced, ...
- updating documentation in virtually every area: many large features have been added (mainly about charts and images at the moment) but without any documentation, it's pretty hard to do anything with it
- proposing compatibility fixes for different versions of Python: we support 2.6 to 3.5, so if it does not work on your environment, let us know :-)

CHAPTER 5

Installation

Install openpyxl using pip. It is advisable to do this in a Python virtualenv without system packages:

```
$ pip install openpyxl
```

Note: There is support for the popular lxml library which will be used if it is installed. This is particular useful when creating large files.

Warning: To be able to include images (jpeg, png, bmp,...) into an openpyxl file, you will also need the "pillow" library that can be installed with:

```
$ pip install pillow
```

or browse https://pypi.python.org/pypi/Pillow/, pick the latest version and head to the bottom of the page for Windows binaries.

CHAPTER 6

Working with a checkout

Sometimes you might want to work with the checkout of a particular version. This may be the case if bugs have been fixed but a release has not yet been made.

```
$ pip install -e hg+https://bitbucket.org/openpyxl/openpyxl@2.4#egg=openpyxl
```

CHAPTER 7

Usage examples

7.1 Tutorial

7.1.1 Manipulating a workbook in memory

Create a workbook

There is no need to create a file on the filesystem to get started with openpyxl. Just import the Workbook class and start using it

```
>>> from openpyxl import Workbook
>>> wb = Workbook()
```

A workbook is always created with at least one worksheet. You can get it by using the `openpyxl.workbook.Workbook.active()` property

```
>>> ws = wb.active
```

Note: This function uses the *_active_sheet_index* property, set to 0 by default. Unless you modify its value, you will always get the first worksheet by using this method.

You can also create new worksheets by using the `openpyxl.workbook.Workbook.create_sheet()` method

```
>>> ws1 = wb.create_sheet("Mysheet") # insert at the end (default)
# or
>>> ws2 = wb.create_sheet("Mysheet", 0) # insert at first position
```

Sheets are given a name automatically when they are created. They are numbered in sequence (Sheet, Sheet1, Sheet2, ...). You can change this name at any time with the *title* property:

```
ws.title = "New Title"
```

The background color of the tab holding this title is white by default. You can change this providing an RRGGBB color code to the sheet_properties.tabColor property:

```
ws.sheet_properties.tabColor = "1072BA"
```

Once you gave a worksheet a name, you can get it as a key of the workbook:

```
>>> ws3 = wb["New Title"]
```

You can review the names of all worksheets of the workbook with the `openpyxl.workbook.Workbook.sheetnames()` property

```
>>> print(wb.sheetnames)
['Sheet2', 'New Title', 'Sheet1']
```

You can loop through worksheets

```
>>> for sheet in wb:
...     print(sheet.title)
```

You can create copies of worksheets *within a single workbook*:

`openpyxl.workbook.Workbook.copy_worksheet()` method:

```
>>> source = wb.active
>>> target = wb.copy_worksheet(source)
```

Note: Only cells (including values, styles, hyperlinks and comments) and certain worksheet attribues (including dimensions, format and properties) are copied. All other workbook / worksheet attributes are not copied - e.g. Images, Charts.

Note: You cannot copy worksheets between workbooks. You also cannot copy a worksheet if the workbook is open in *read-only* or *write-only* mode.

Playing with data

Accessing one cell

Now we know how to access a worksheet, we can start modifying cells content.

Cells can be accessed directly as keys of the worksheet

```
>>> c = ws['A4']
```

This will return the cell at A4 or create one if it does not exist yet. Values can be directly assigned

```
>>> ws['A4'] = 4
```

There is also the `openpyxl.worksheet.Worksheet.cell()` method.

This provides access to cells using row and column notation:

```
>>> d = ws.cell(row=4, column=2, value=10)
```

Note: When a worksheet is created in memory, it contains no *cells*. They are created when first accessed.

Warning: Because of this feature, scrolling through cells instead of accessing them directly will create them all in memory, even if you don't assign them a value.

Something like

```
>>> for i in range(1,101):
...         for j in range(1,101):
...             ws.cell(row=i, column=j)
```

will create 100x100 cells in memory, for nothing.

Accessing many cells

Ranges of cells can be accessed using slicing

```
>>> cell_range = ws['A1':'C2']
```

Ranges of rows or columns can be obtained similarly:

```
>>> colC = ws['C']
>>> col_range = ws['C:D']
>>> row10 = ws[10]
>>> row_range = ws[5:10]
```

You can also use the `openpyxl.worksheet.Worksheet.iter_rows()` method:

```
>>> for row in ws.iter_rows(min_row=1, max_col=3, max_row=2):
...     for cell in row:
...         print(cell)
<Cell Sheet1.A1>
<Cell Sheet1.B1>
<Cell Sheet1.C1>
<Cell Sheet1.A2>
<Cell Sheet1.B2>
<Cell Sheet1.C2>
```

Likewise the `openpyxl.worksheet.Worksheet.iter_cols()` method will return columns:

```
>>> for col in ws.iter_cols(min_row=1, max_col=3, max_row=2):
...     for cell in col:
...         print(cell)
<Cell Sheet1.A1>
<Cell Sheet1.A2>
<Cell Sheet1.B1>
<Cell Sheet1.B2>
<Cell Sheet1.C1>
<Cell Sheet1.C2>
```

If you need to iterate through all the rows or columns of a file, you can instead use the `openpyxl.worksheet.Worksheet.rows()` property:

```
>>> ws = wb.active
>>> ws['C9'] = 'hello world'
>>> tuple(ws.rows)
((<Cell Sheet.A1>, <Cell Sheet.B1>, <Cell Sheet.C1>),
(<Cell Sheet.A2>, <Cell Sheet.B2>, <Cell Sheet.C2>),
(<Cell Sheet.A3>, <Cell Sheet.B3>, <Cell Sheet.C3>),
(<Cell Sheet.A4>, <Cell Sheet.B4>, <Cell Sheet.C4>),
(<Cell Sheet.A5>, <Cell Sheet.B5>, <Cell Sheet.C5>),
(<Cell Sheet.A6>, <Cell Sheet.B6>, <Cell Sheet.C6>),
(<Cell Sheet.A7>, <Cell Sheet.B7>, <Cell Sheet.C7>),
(<Cell Sheet.A8>, <Cell Sheet.B8>, <Cell Sheet.C8>),
(<Cell Sheet.A9>, <Cell Sheet.B9>, <Cell Sheet.C9>))
```

or the `openpyxl.worksheet.Worksheet.columns()` property:

```
>>> tuple(ws.columns)
((<Cell Sheet.A1>,
<Cell Sheet.A2>,
<Cell Sheet.A3>,
<Cell Sheet.A4>,
<Cell Sheet.A5>,
<Cell Sheet.A6>,
...
<Cell Sheet.B7>,
<Cell Sheet.B8>,
<Cell Sheet.B9>),
(<Cell Sheet.C1>,
<Cell Sheet.C2>,
<Cell Sheet.C3>,
<Cell Sheet.C4>,
<Cell Sheet.C5>,
<Cell Sheet.C6>,
<Cell Sheet.C7>,
<Cell Sheet.C8>,
<Cell Sheet.C9>))
```

Data storage

Once we have a `openpyxl.cell.Cell`, we can assign it a value:

```
>>> c.value = 'hello, world'
>>> print(c.value)
'hello, world'

>>> d.value = 3.14
>>> print(d.value)
3.14
```

You can also enable type and format inference:

```
>>> wb = Workbook(guess_types=True)
>>> c.value = '12%'
>>> print(c.value)
0.12

>>> import datetime
```

```
>>> d.value = datetime.datetime.now()
>>> print d.value
datetime.datetime(2010, 9, 10, 22, 25, 18)

>>> c.value = '31.50'
>>> print(c.value)
31.5
```

7.1.2 Saving to a file

The simplest and safest way to save a workbook is by using the `openpyxl.workbook.Workbook.save()` method of the `openpyxl.workbook.Workbook` object:

```
>>> wb = Workbook()
>>> wb.save('balances.xlsx')
```

Warning: This operation will overwrite existing files without warning.

Note: Extension is not forced to be xlsx or xlsm, although you might have some trouble opening it directly with another application if you don't use an official extension.

As OOXML files are basically ZIP files, you can also end the filename with .zip and open it with your favourite ZIP archive manager.

You can specify the attribute *template=True*, to save a workbook as a template:

```
>>> wb = load_workbook('document.xlsx')
>>> wb.template = True
>>> wb.save('document_template.xltx')
```

or set this attribute to *False* (default), to save as a document:

```
>>> wb = load_workbook('document_template.xltx')
>>> wb.template = False
>>> wb.save('document.xlsx', as_template=False)
```

Warning: You should monitor the data attributes and document extensions for saving documents in the document templates and vice versa, otherwise the result table engine can not open the document.

Note: The following will fail:

```
>>> wb = load_workbook('document.xlsx')
>>> # Need to save with the extension *.xlsx
>>> wb.save('new_document.xlsm')
>>> # MS Excel can't open the document
>>>
>>> # or
>>>
>>> # Need specify attribute keep_vba=True
```

```
>>> wb = load_workbook('document.xlsm')
>>> wb.save('new_document.xlsm')
>>> # MS Excel will not open the document
>>>
>>> # or
>>>
>>> wb = load_workbook('document.xltm', keep_vba=True)
>>> # If we need a template document, then we must specify extension as *.xltm.
>>> wb.save('new_document.xlsm')
>>> # MS Excel will not open the document
```

7.1.3 Loading from a file

The same way as writing, you can import `openpyxl.load_workbook()` to open an existing workbook:

```
>>> from openpyxl import load_workbook
>>> wb2 = load_workbook('test.xlsx')
>>> print wb2.get_sheet_names()
['Sheet2', 'New Title', 'Sheet1']
```

This ends the tutorial for now, you can proceed to the *Simple usage* section

7.2 Cookbook

7.2.1 Simple usage

Write a workbook

```
>>> from openpyxl import Workbook
>>> from openpyxl.compat import range
>>> from openpyxl.utils import get_column_letter
>>>
>>> wb = Workbook()
>>>
>>> dest_filename = 'empty_book.xlsx'
>>>
>>> ws1 = wb.active
>>> ws1.title = "range names"
>>>
>>> for row in range(1, 40):
...     ws1.append(range(600))
>>>
>>> ws2 = wb.create_sheet(title="Pi")
>>>
>>> ws2['F5'] = 3.14
>>>
>>> ws3 = wb.create_sheet(title="Data")
>>> for row in range(10, 20):
...     for col in range(27, 54):
...         _ = ws3.cell(column=col, row=row, value="{0}".format(get_column_
↪letter(col)))
>>> print(ws3['AA10'].value)
```

```
AA
>>> wb.save(filename = dest_filename)
```

Read an existing workbook

```
>>> from openpyxl import load_workbook
>>> wb = load_workbook(filename = 'empty_book.xlsx')
>>> sheet_ranges = wb['range names']
>>> print(sheet_ranges['D18'].value)
3
```

Note: There are several flags that can be used in load_workbook.

- *guess_types* will enable or disable (default) type inference when reading cells.
- *data_only* controls whether cells with formulae have either the formula (default) or the value stored the last time Excel read the sheet.
- *keep_vba* controls whether any Visual Basic elements are preserved or not (default). If they are preserved they are still not editable.

Warning: openpyxl does currently not read all possible items in an Excel file so images and charts will be lost from existing files if they are opened and saved with the same name.

Using number formats

```
>>> import datetime
>>> from openpyxl import Workbook
>>> wb = Workbook()
>>> ws = wb.active
>>> # set date using a Python datetime
>>> ws['A1'] = datetime.datetime(2010, 7, 21)
>>>
>>> ws['A1'].number_format
'yyyy-mm-dd h:mm:ss'
>>> # You can enable type inference on a case-by-case basis
>>> wb.guess_types = True
>>> # set percentage using a string followed by the percent sign
>>> ws['B1'] = '3.14%'
>>> wb.guess_types = False
>>> ws['B1'].value
0.031400000000000004
>>>
>>> ws['B1'].number_format
'0%'
```

Using formulae

```
>>> from openpyxl import Workbook
>>> wb = Workbook()
>>> ws = wb.active
>>> # add a simple formula
>>> ws["A1"] = "=SUM(1, 1)"
>>> wb.save("formula.xlsx")
```

Warning: NB you must use the English name for a function and function arguments *must* be separated by commas and not other punctuation such as semi-colons.

openpyxl never evaluates formula but it is possible to check the name of a formula:

```
>>> from openpyxl.utils import FORMULAE
>>> "HEX2DEC" in FORMULAE
True
```

If you're trying to use a formula that isn't known this could be because you're using a formula that was not included in the initial specification. Such formulae must be prefixed with *_xlfn.* to work.

Merge / Unmerge cells

When you merge cells all cells but the top-left one are **removed** from the worksheet. See *Styling Merged Cells* for information on formatting merged cells.

```
>>> from openpyxl.workbook import Workbook
>>>
>>> wb = Workbook()
>>> ws = wb.active
>>>
>>> ws.merge_cells('A2:D2')
>>> ws.unmerge_cells('A2:D2')
>>>
>>> # or equivalently
>>> ws.merge_cells(start_row=2,start_column=1,end_row=2,end_column=4)
>>> ws.unmerge_cells(start_row=2,start_column=1,end_row=2,end_column=4)
```

Inserting an image

```
>>> from openpyxl import Workbook
>>> from openpyxl.drawing.image import Image
>>>
>>> wb = Workbook()
>>> ws = wb.active
>>> ws['A1'] = 'You should see three logos below'
```

```
>>> # create an image
>>> img = Image('logo.png')
```

```
>>> # add to worksheet and anchor next to cells
>>> ws.add_image(img, 'A1')
>>> wb.save('logo.xlsx')
```

Fold columns (outline)

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> ws = wb.create_sheet()
>>> ws.column_dimensions.group('A','D', hidden=True)
>>> wb.save('group.xlsx')
```

7.3 Pandas and NumPy

7.3.1 Working with Pandas and NumPy

openpyxl is able to work with the popular libraries Pandas and NumPy

NumPy Support

openpyxl has builtin support for the NumPy types float, integer and boolean. DateTimes are supported using the Pandas' Timestamp type.

Working with Pandas Dataframes

The `openpyxl.utils.dataframe.dataframe_to_rows()` function provides a simple way to work with Pandas Dataframes:

```
from openpyxl.utils.dataframe import dataframe_to_rows
wb = Workbook()
ws = wb.active

for r in dataframe_to_rows(df, index=True, header=True):
    ws.append(r)
```

While Pandas itself supports conversion to Excel, this gives client code additional flexibility including the ability to stream dataframes straight to files.

To convert a dataframe into a worksheet highlighting the header and index:

```
wb = Workbook()
ws = wb.active

for r in dataframe_to_rows(df, index=True, header=True):
    ws.append(r)

for cell in ws['A'] + ws[1]:
    cell.style = 'Pandas'

wb.save("pandas_openpyxl.xlsx")
```

Alternatively, if you just want to convert the data you can use write-only mode:

```
from openpyxl.cell.cell import WriteOnlyCell
wb = Workbook(write_only=True)
ws = wb.create_sheet()
```

```
cell = WriteOnlyCell(ws)
cell.style = 'Pandas'

 def format_first_row(row, cell):

    for c in row:
        cell.value = c
        yield cell

rows = dataframe_to_rows(df)
first_row = format_first_row(next(rows), cell)
ws.append(first_row)

for row in rows:
    row = list(row)
    cell.value = row[0]
    row[0] = cell
    ws.append(row)

wb.save("openpyxl_stream.xlsx")
```

This code will work just as well with a standard workbook.

Converting a worksheet to a Dataframe

To convert a worksheet to a Dataframe you can use the *values* property. This is very easy if the worksheet has no headers or indices:

```
df = DataFrame(ws.values)
```

If the worksheet does have headers or indices, such as one created by Pandas, then a little more work is required:

```
data = ws.values
cols = next(data)[1:]
data = list(data)
idx = [r[0] for r in data]
data = (islice(r, 1, None) for r in data)
df = DataFrame(data, index=idx, columns=cols)
```

7.4 Charts

7.4.1 Charts

Warning: Openpyxl currently supports chart creation within a worksheet only. Charts in existing workbooks will be lost.

Chart types

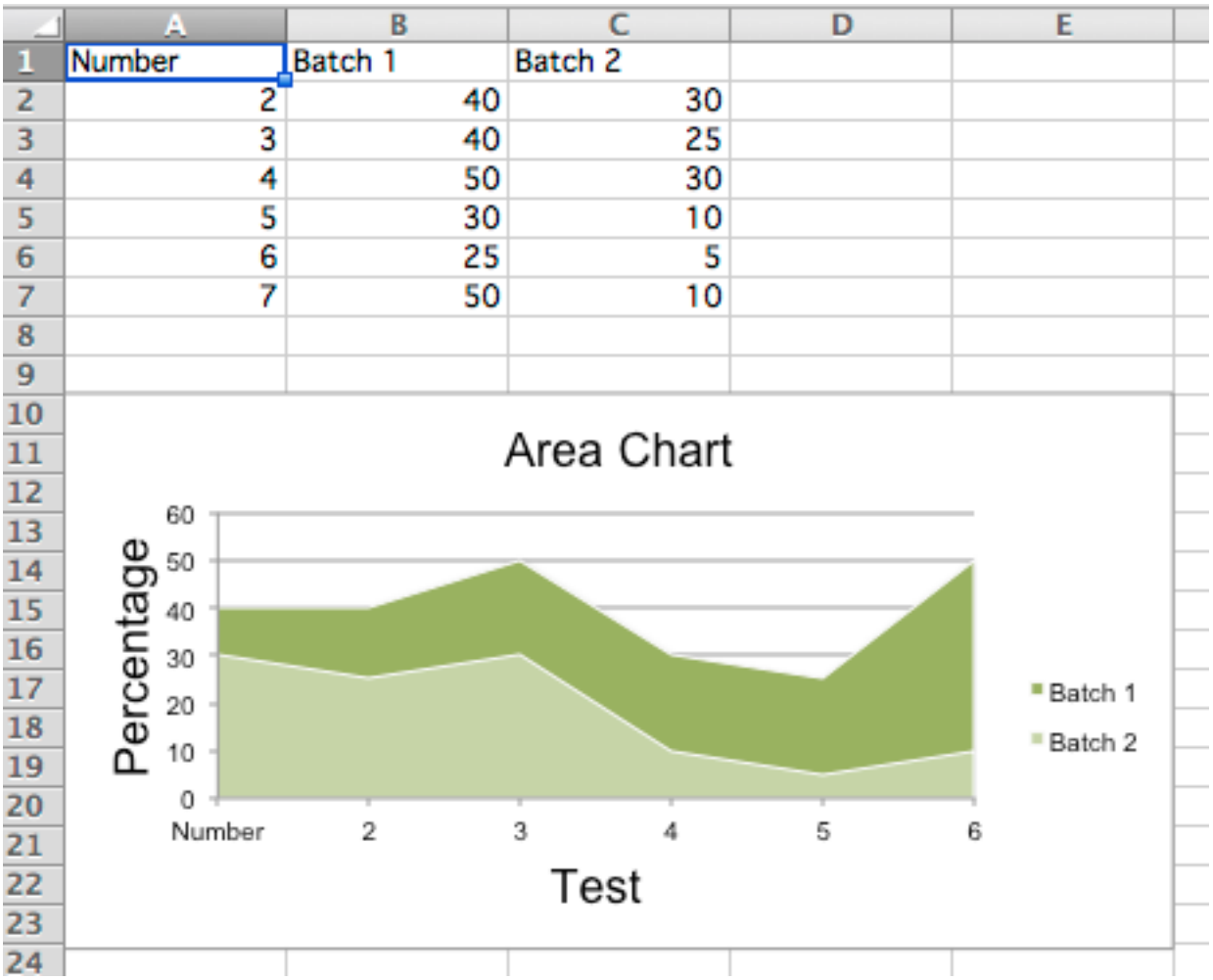
The following charts are available:

Area Charts

2D Area Charts

Area charts are similar to line charts with the addition that the area underneath the plotted line is filled. Different variants are available by setting the grouping to “standard”, “stacked” or “percentStacked”; “standard” is the default.

```
from openpyxl import Workbook
from openpyxl.chart import (
    AreaChart,
    Reference,
    Series,
)

wb = Workbook()
ws = wb.active

rows = [
    ['Number', 'Batch 1', 'Batch 2'],
    [2, 40, 30],
    [3, 40, 25],
    [4, 50, 30],
    [5, 30, 10],
    [6, 25, 5],
    [7, 50, 10],
]

for row in rows:
    ws.append(row)

chart = AreaChart()
chart.title = "Area Chart"
chart.style = 13
chart.x_axis.title = 'Test'
chart.y_axis.title = 'Percentage'

cats = Reference(ws, min_col=1, min_row=1, max_row=7)
data = Reference(ws, min_col=2, min_row=1, max_col=3, max_row=7)
chart.add_data(data, titles_from_data=True)
chart.set_categories(cats)

ws.add_chart(chart, "A10")

wb.save("area.xlsx")
```



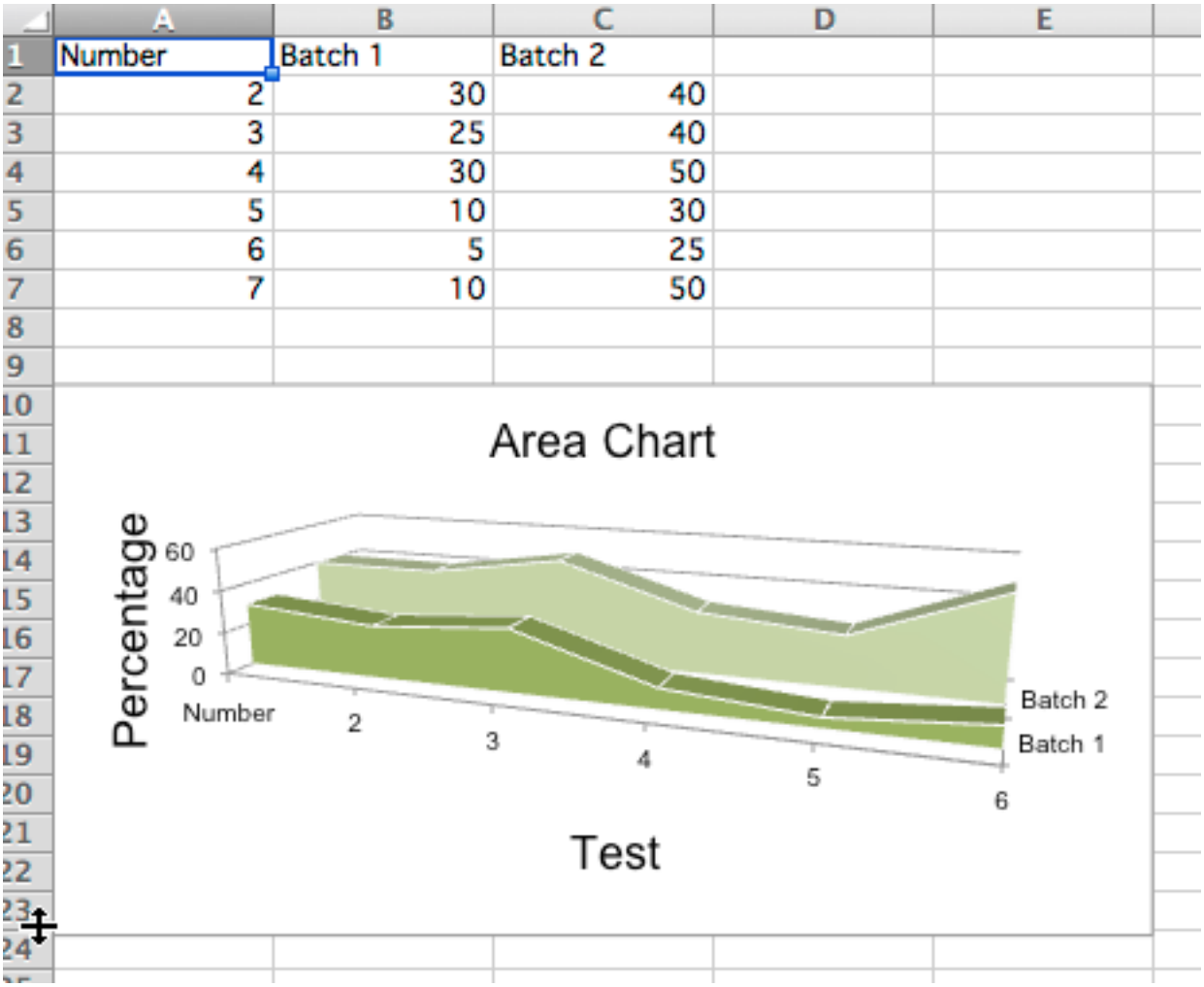
3D Area Charts

You can also create 3D area charts

```
from openpyxl import Workbook
from openpyxl.chart import (
    AreaChart3D,
    Reference,
    Series,
)

wb = Workbook()
ws = wb.active

rows = [
    ['Number', 'Batch 1', 'Batch 2'],
    [2, 30, 40],
    [3, 25, 40],
    [4 ,30, 50],
    [5 ,10, 30],
    [6,  5, 25],
    [7 ,10, 50],
```

```
]

for row in rows:
    ws.append(row)

chart = AreaChart3D()
chart.title = "Area Chart"
chart.style = 13
chart.x_axis.title = 'Test'
chart.y_axis.title = 'Percentage'
chart.legend = None

cats = Reference(ws, min_col=1, min_row=1, max_row=7)
data = Reference(ws, min_col=2, min_row=1, max_col=3, max_row=7)
chart.add_data(data, titles_from_data=True)
chart.set_categories(cats)

ws.add_chart(chart, "A10")

wb.save("area3D.xlsx")
```

This produces a simple 3D area chart where third axis can be used to replace the legend:



Bar and Column Charts

In bar charts values are plotted as either horizontal bars or vertical columns.

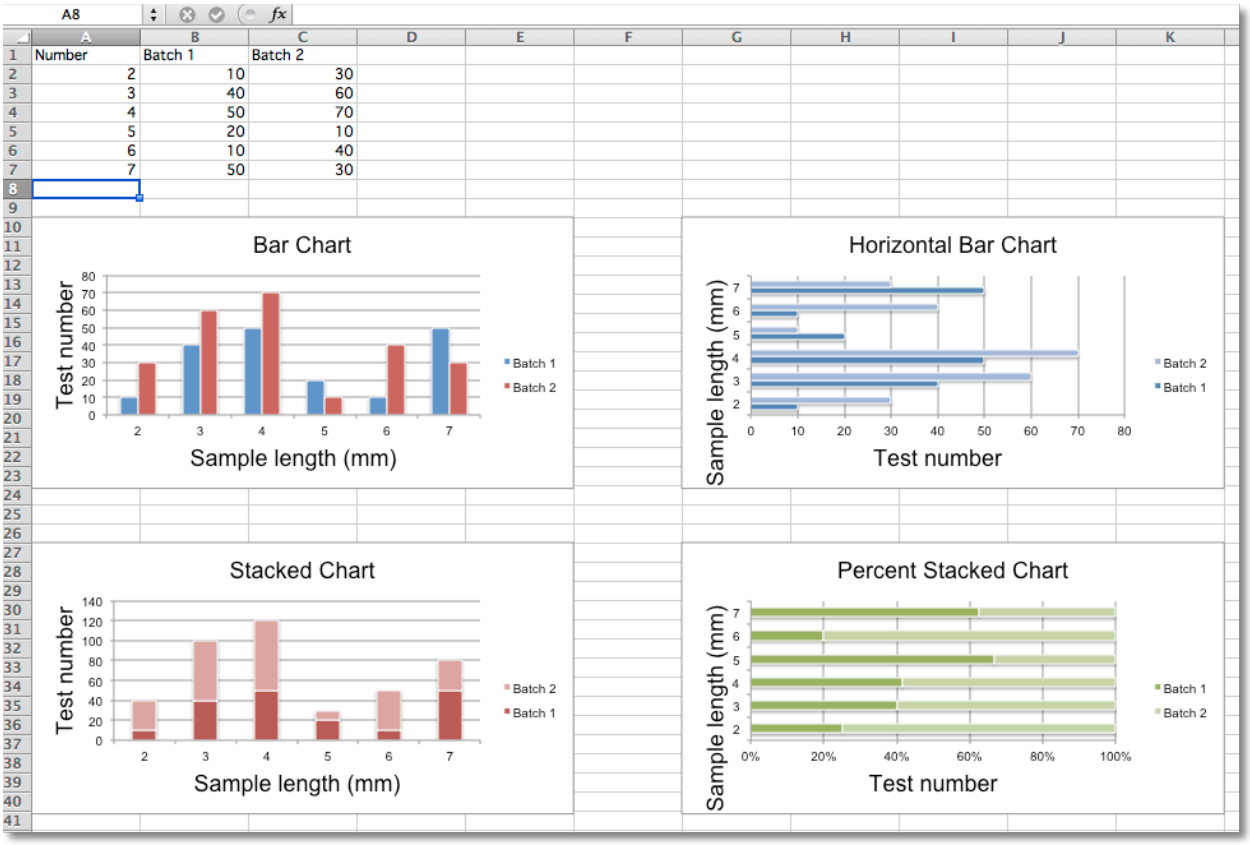
Vertical, Horizontal and Stacked Bar Charts

Note: The following settings affect the different chart types.

Switch between vertical and horizontal bar charts by setting *type* to *col* or *bar* respectively.

When using stacked charts the *overlap* needs to be set to 100.

If bars are horizontal, x and y axes are revesed.




```
from openpyxl import Workbook
from openpyxl.chart import BarChart, Series, Reference

wb = Workbook(write_only=True)
ws = wb.create_sheet()

rows = [
    ('Number', 'Batch 1', 'Batch 2'),
    (2, 10, 30),
    (3, 40, 60),
    (4, 50, 70),
    (5, 20, 10),
```

```
    (6, 10, 40),
    (7, 50, 30),
]


for row in rows:
    ws.append(row)


chart1 = BarChart()
chart1.type = "col"
chart1.style = 10
chart1.title = "Bar Chart"
chart1.y_axis.title = 'Test number'
chart1.x_axis.title = 'Sample length (mm)'

data = Reference(ws, min_col=2, min_row=1, max_row=7, max_col=3)
cats = Reference(ws, min_col=1, min_row=2, max_row=7)
chart1.add_data(data, titles_from_data=True)
chart1.set_categories(cats)
chart1.shape = 4
ws.add_chart(chart1, "A10")

from copy import deepcopy

chart2 = deepcopy(chart1)
chart2.style = 11
chart2.type = "bar"
chart2.title = "Horizontal Bar Chart"

ws.add_chart(chart2, "G10")


chart3 = deepcopy(chart1)
chart3.type = "col"
chart3.style = 12
chart3.grouping = "stacked"
chart3.overlap = 100
chart3.title = 'Stacked Chart'

ws.add_chart(chart3, "A27")


chart4 = deepcopy(chart1)
chart4.type = "bar"
chart4.style = 13
chart4.grouping = "percentStacked"
chart4.overlap = 100
chart4.title = 'Percent Stacked Chart'

ws.add_chart(chart4, "G27")

wb.save("bar.xlsx")
```

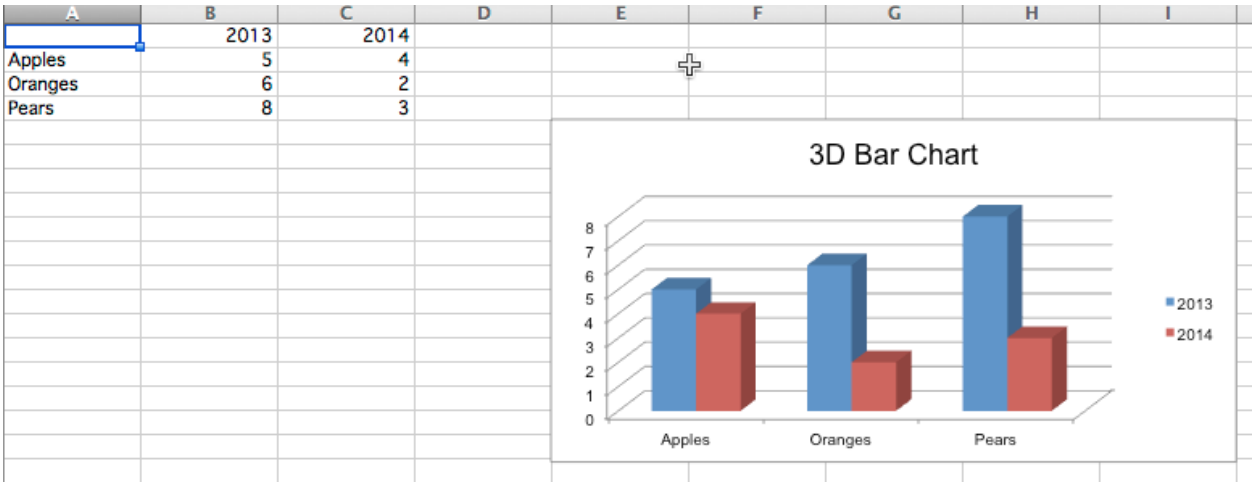
This will produce four charts illustrating the various possibilities.

3D Bar Charts

You can also create 3D bar charts

```
from openpyxl import Workbook
from openpyxl.chart import (
    Reference,
    Series,
    BarChart3D,
)

wb = Workbook()
ws = wb.active

rows = [
    (None, 2013, 2014),
    ("Apples", 5, 4),
    ("Oranges", 6, 2),
    ("Pears", 8, 3)
]

for row in rows:
    ws.append(row)

data = Reference(ws, min_col=2, min_row=1, max_col=3, max_row=4)
titles = Reference(ws, min_col=1, min_row=2, max_row=4)
chart = BarChart3D()
chart.title = "3D Bar Chart"
chart.add_data(data=data, titles_from_data=True)
chart.set_categories(titles)

ws.add_chart(chart, "E5")
wb.save("bar3d.xlsx")
```

This produces a simple 3D bar chart

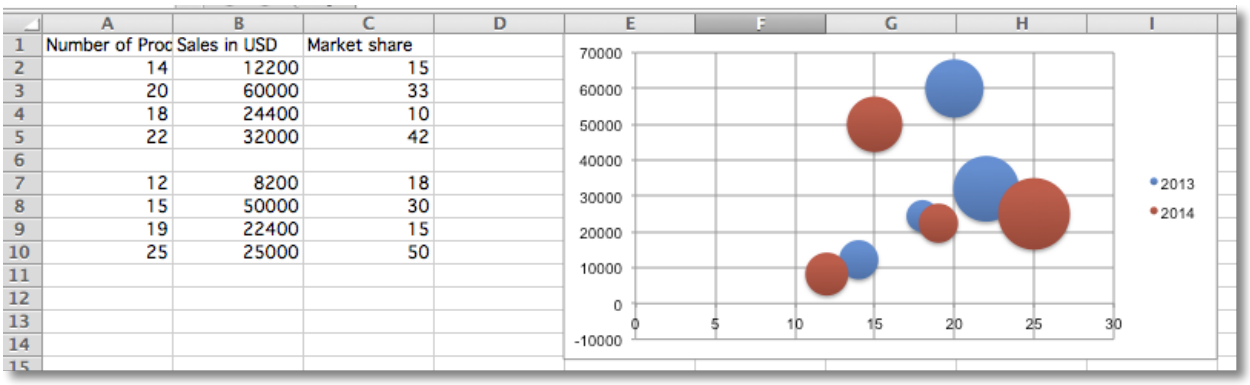


Bubble Charts

Bubble charts are similar to scatter charts but use a third dimension to determine the size of the bubbles. Charts can include multiple series.

```
"""
Sample bubble chart
"""

from openpyxl import Workbook
from openpyxl.chart import Series, Reference, BubbleChart

wb = Workbook()
ws = wb.active

rows = [
    ("Number of Products", "Sales in USD", "Market share"),
    (14, 12200, 15),
    (20, 60000, 33),
    (18, 24400, 10),
    (22, 32000, 42),
    (),
    (12, 8200, 18),
    (15, 50000, 30),
    (19, 22400, 15),
    (25, 25000, 50),
]

for row in rows:
    ws.append(row)

chart = BubbleChart()
chart.style = 18 # use a preset style

# add the first series of data
xvalues = Reference(ws, min_col=1, min_row=2, max_row=5)
yvalues = Reference(ws, min_col=2, min_row=2, max_row=5)
size = Reference(ws, min_col=3, min_row=2, max_row=5)
series = Series(values=yvalues, xvalues=xvalues, zvalues=size, title="2013")
chart.series.append(series)

# add the second
xvalues = Reference(ws, min_col=1, min_row=7, max_row=10)
yvalues = Reference(ws, min_col=2, min_row=7, max_row=10)
size = Reference(ws, min_col=3, min_row=7, max_row=10)
series = Series(values=yvalues, xvalues=xvalues, zvalues=size, title="2014")
chart.series.append(series)

# place the chart starting in cell E1
ws.add_chart(chart, "E1")
wb.save("bubble.xlsx")
```

This will produce bubble chart with two series and should look something like this



	A	B	C
1	Number of Proc	Sales in USD	Market share
2	14	12200	15
3	20	60000	33
4	18	24400	10
5	22	32000	42
6			
7	12	8200	18
8	15	50000	30
9	19	22400	15
10	25	25000	50

Line Charts

Line Charts

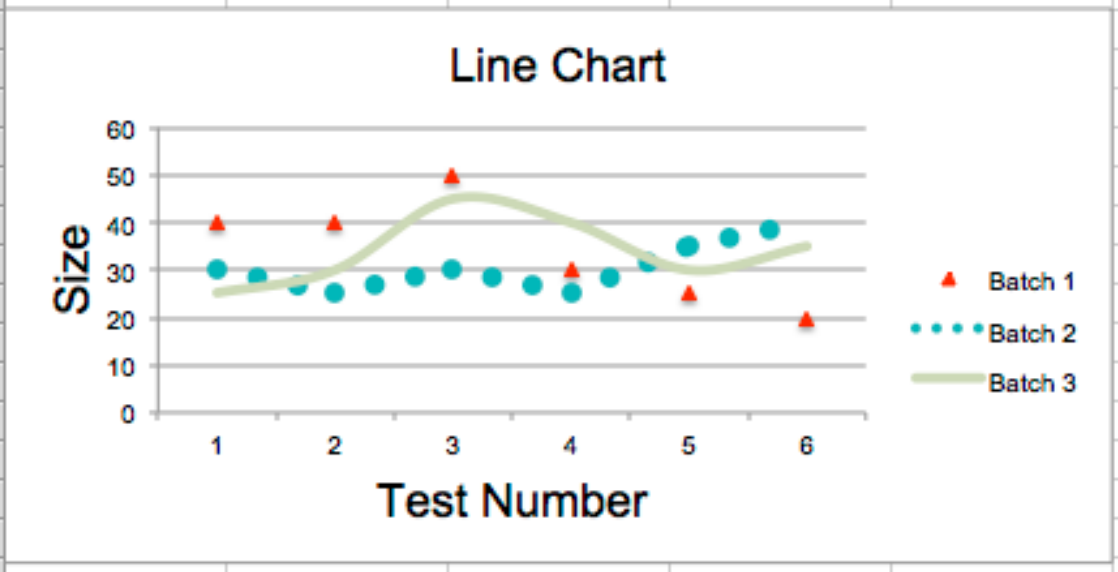
Line charts allow data to be plotted against a fixed axis. They are similar to scatter charts, the main difference is that with line charts each data series is plotted against the same values. Different kinds of axes can be used for the secondary axes.

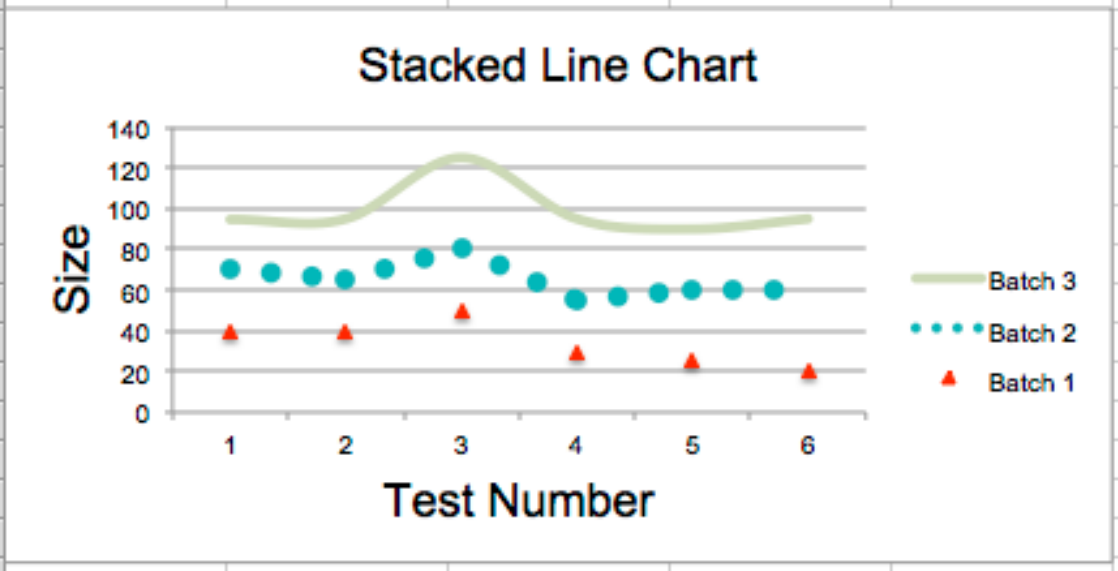
Similar to bar charts there are three kinds of line charts: standard, stacked and percentStacked.

```
from datetime import date

from openpyxl import Workbook
from openpyxl.chart import (
    LineChart,
    Reference,
)
from openpyxl.chart.axis import DateAxis

wb = Workbook()
ws = wb.active

rows = [
    ['Date', 'Batch 1', 'Batch 2', 'Batch 3'],
    [date(2015,9, 1), 40, 30, 25],
    [date(2015,9, 2), 40, 25, 30],
    [date(2015,9, 3), 50, 30, 45],
    [date(2015,9, 4), 30, 25, 40],
    [date(2015,9, 5), 25, 35, 30],
    [date(2015,9, 6), 20, 40, 35],
]

for row in rows:
    ws.append(row)

c1 = LineChart()
c1.title = "Line Chart"
c1.style = 13
c1.y_axis.title = 'Size'
c1.x_axis.title = 'Test Number'

data = Reference(ws, min_col=2, min_row=1, max_col=4, max_row=7)
c1.add_data(data, titles_from_data=True)
```

```
# Style the lines
s1 = c1.series[0]
s1.marker.symbol = "triangle"
s1.marker.graphicalProperties.solidFill = "FF0000" # Marker filling
s1.marker.graphicalProperties.line.solidFill = "FF0000" # Marker outline

s1.graphicalProperties.line.noFill = True

s2 = c1.series[1]
s2.graphicalProperties.line.solidFill = "00AAAA"
s2.graphicalProperties.line.dashStyle = "sysDot"
s2.graphicalProperties.line.width = 100050 # width in EMUs

s2 = c1.series[2]
s2.smooth = True # Make the line smooth

ws.add_chart(c1, "A10")

from copy import deepcopy
stacked = deepcopy(c1)
stacked.grouping = "stacked"
stacked.title = "Stacked Line Chart"
ws.add_chart(stacked, "A27")

percent_stacked = deepcopy(c1)
percent_stacked.grouping = "percentStacked"
percent_stacked.title = "Percent Stacked Line Chart"
ws.add_chart(percent_stacked, "A44")

# Chart with date axis
c2 = LineChart()
c2.title = "Date Axis"
c2.style = 12
c2.y_axis.title = "Size"
c2.y_axis.crossAx = 500
c2.x_axis = DateAxis(crossAx=100)
c2.x_axis.number_format = 'd-mmm'
c2.x_axis.majorTimeUnit = "days"
c2.x_axis.title = "Date"

c2.add_data(data, titles_from_data=True)
dates = Reference(ws, min_col=1, min_row=2, max_row=7)
c2.set_categories(dates)

ws.add_chart(c2, "A61")

wb.save("line.xlsx")
```

	A	B	C	D	E
1	Date	Batch 1	Batch 2	Batch 3	
2	2015-09-01	40	30	25	
3	2015-09-02	40	25	30	
4	2015-09-03	50	30	45	
5	2015-09-04	30	25	40	
6	2015-09-05	25	35	30	
7	2015-09-06	20	40	35	

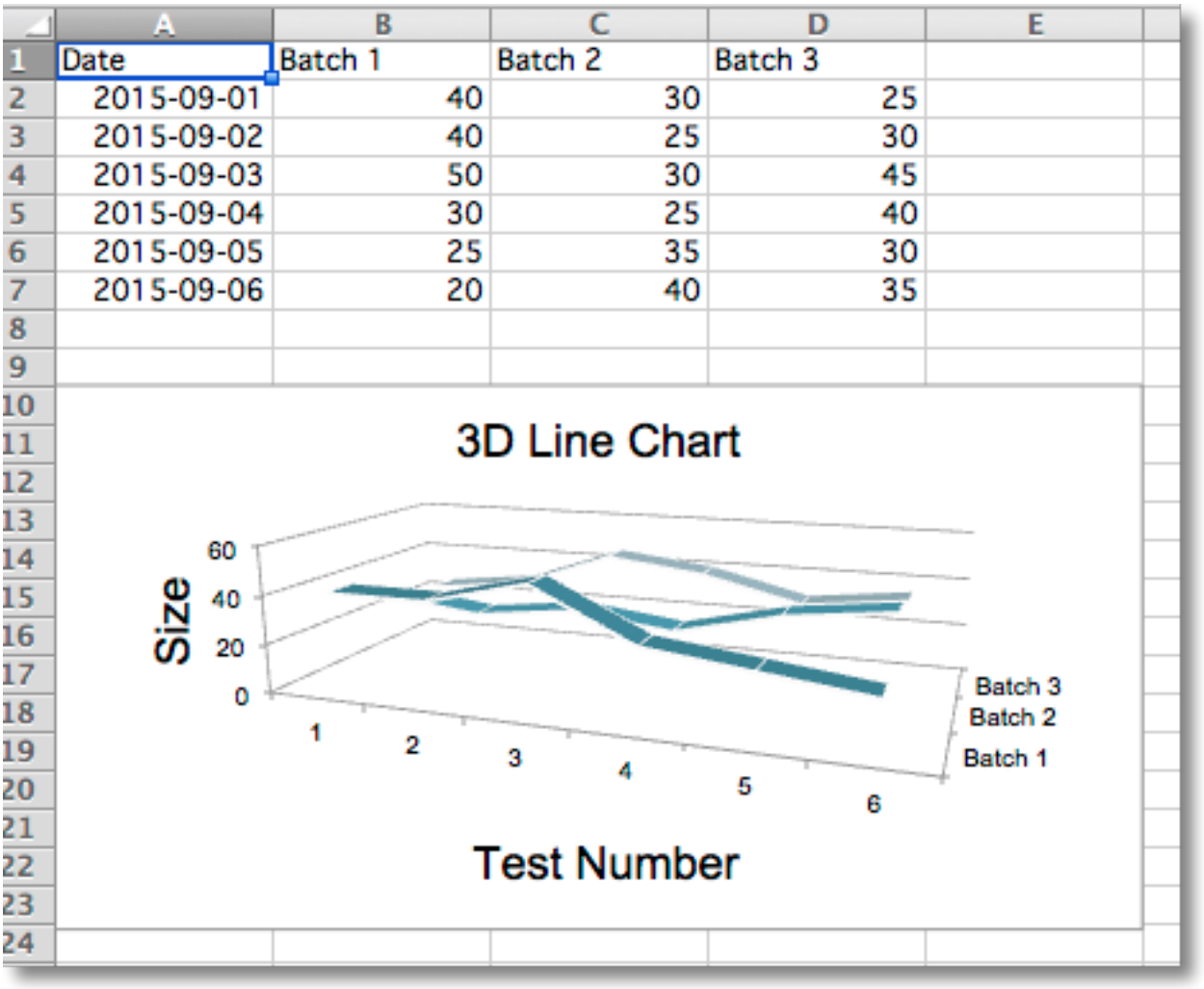




3D Line Charts

In 3D line charts the third axis is the same as the legend for the series.

```
from datetime import date

from openpyxl import Workbook
from openpyxl.chart import (
    LineChart3D,
    Reference,
)
from openpyxl.chart.axis import DateAxis

wb = Workbook()
ws = wb.active

rows = [
    ['Date', 'Batch 1', 'Batch 2', 'Batch 3'],
    [date(2015,9, 1), 40, 30, 25],
    [date(2015,9, 2), 40, 25, 30],
    [date(2015,9, 3), 50, 30, 45],
    [date(2015,9, 4), 30, 25, 40],
    [date(2015,9, 5), 25, 35, 30],
    [date(2015,9, 6), 20, 40, 35],
]

for row in rows:
    ws.append(row)

c1 = LineChart3D()
c1.title = "3D Line Chart"
c1.legend = None
c1.style = 15
c1.y_axis.title = 'Size'
c1.x_axis.title = 'Test Number'

data = Reference(ws, min_col=2, min_row=1, max_col=4, max_row=7)
c1.add_data(data, titles_from_data=True)

ws.add_chart(c1, "A10")

wb.save("line3D.xlsx")
```

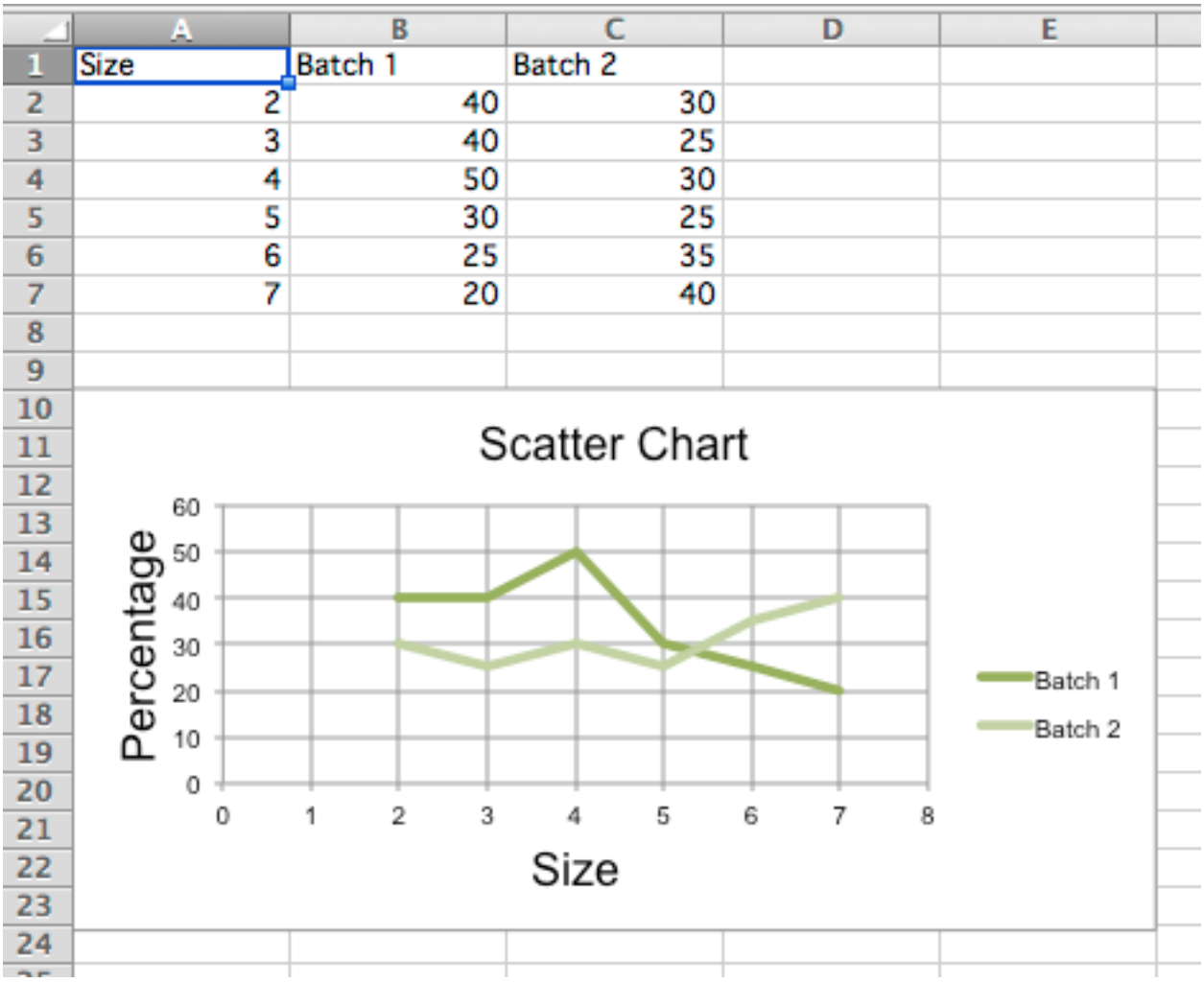


Scatter Charts

Scatter, or xy, charts are similar to some line charts. The main difference is that one series of values is plotted against another. This is useful where values are unordered.

```
from openpyxl import Workbook
from openpyxl.chart import (
    ScatterChart,
    Reference,
    Series,
)

wb = Workbook()
ws = wb.active

rows = [
    ['Size', 'Batch 1', 'Batch 2'],
    [2, 40, 30],
    [3, 40, 25],
    [4, 50, 30],
    [5, 30, 25],
```

```
    [6, 25, 35],
    [7, 20, 40],
]

for row in rows:
    ws.append(row)

chart = ScatterChart()
chart.title = "Scatter Chart"
chart.style = 13
chart.x_axis.title = 'Size'
chart.y_axis.title = 'Percentage'

xvalues = Reference(ws, min_col=1, min_row=2, max_row=7)
for i in range(2, 4):
    values = Reference(ws, min_col=i, min_row=1, max_row=7)
    series = Series(values, xvalues, title_from_data=True)
    chart.series.append(series)

ws.add_chart(chart, "A10")

wb.save("scatter.xlsx")
```



Note: The specification says that there are the following types of scatter charts: 'line', 'lineMarker', 'marker', 'smooth', 'smoothMarker'. However, at least in Microsoft Excel, this is just a shortcut for other settings that otherwise no effect. For consistency with line charts, the style for each series should be set manually.

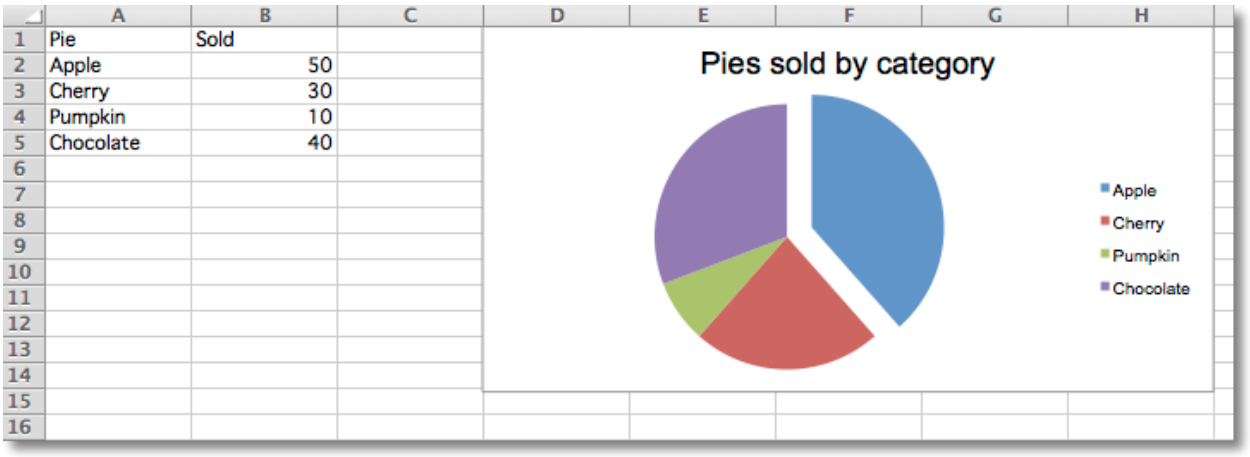
Pie Charts

Pie Charts

Pie charts plot data as slices of a circle with each slice representing the percentage of the whole. Slices are plotted in a clockwise direction with 0° being at the top of the circle. Pie charts can only take a single series of data. The title of the chart will default to being the title of the series.

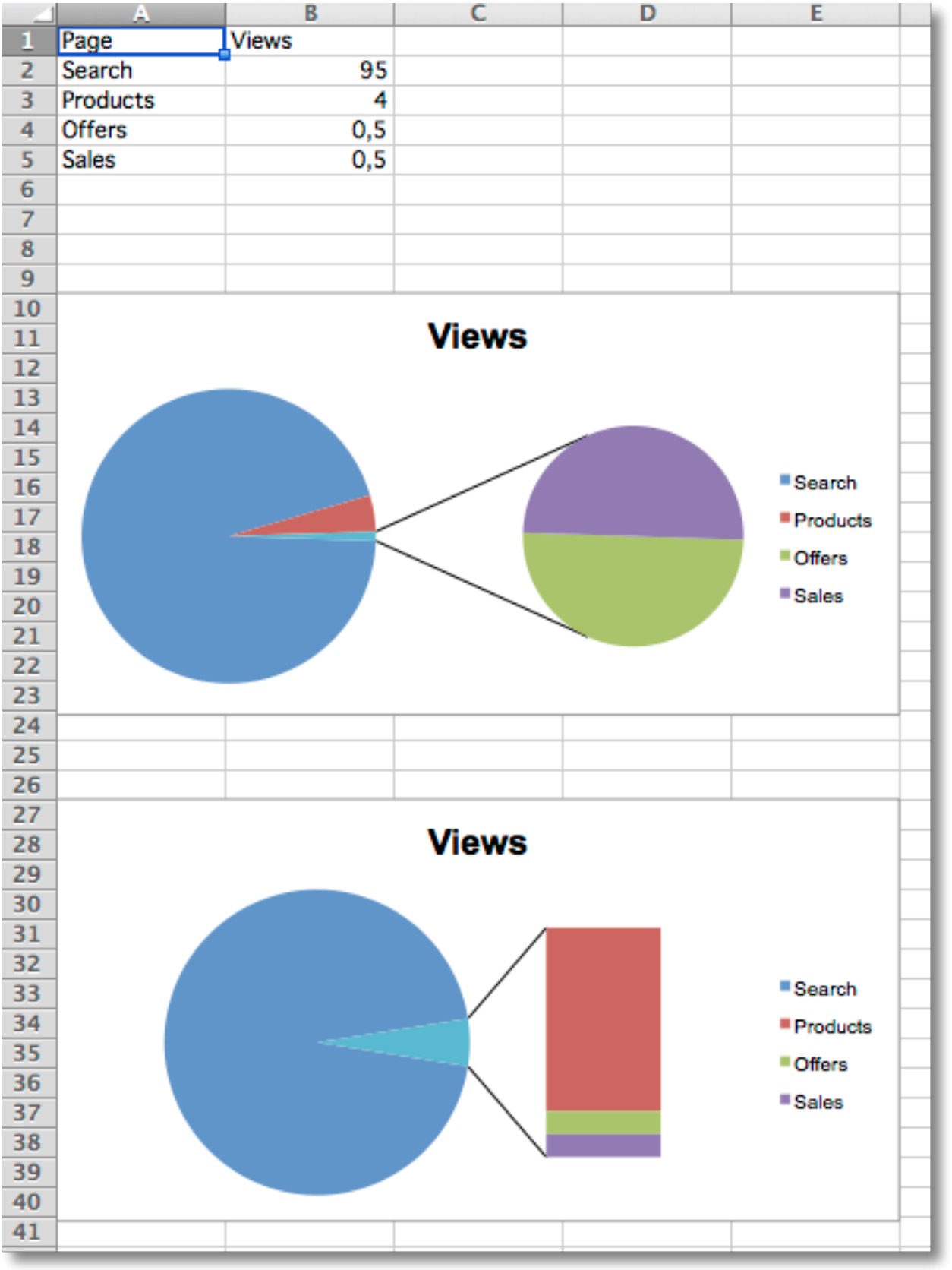
```
from openpyxl import Workbook

from openpyxl.chart import (
    PieChart,
    ProjectedPieChart,
    Reference
)
from openpyxl.chart.series import DataPoint

data = [
    ['Pie', 'Sold'],
    ['Apple', 50],
    ['Cherry', 30],
    ['Pumpkin', 10],
    ['Chocolate', 40],
]

wb = Workbook()
ws = wb.active

for row in data:
    ws.append(row)

pie = PieChart()
labels = Reference(ws, min_col=1, min_row=2, max_row=5)
data = Reference(ws, min_col=2, min_row=1, max_row=5)
pie.add_data(data, titles_from_data=True)
pie.set_categories(labels)
pie.title = "Pies sold by category"

# Cut the first slice out of the pie
slice = DataPoint(idx=0, explosion=20)
pie.series[0].data_points = [slice]

ws.add_chart(pie, "D1")


ws = wb.create_sheet(title="Projection")

data = [
    ['Page', 'Views'],
    ['Search', 95],
```

```
    ['Products', 4],
    ['Offers', 0.5],
    ['Sales', 0.5],
]

for row in data:
    ws.append(row)

projected_pie = ProjectedPieChart()
projected_pie.type = "pie"
projected_pie.splitType = "val" # split by value
labels = Reference(ws, min_col=1, min_row=2, max_row=5)
data = Reference(ws, min_col=2, min_row=1, max_row=5)
projected_pie.add_data(data, titles_from_data=True)
projected_pie.set_categories(labels)

ws.add_chart(projected_pie, "A10")

from copy import deepcopy
projected_bar = deepcopy(projected_pie)
projected_bar.type = "bar"
projected_bar.splitType = 'pos' # split by position

ws.add_chart(projected_bar, "A27")

wb.save("pie.xlsx")
```



Projected Pie Charts

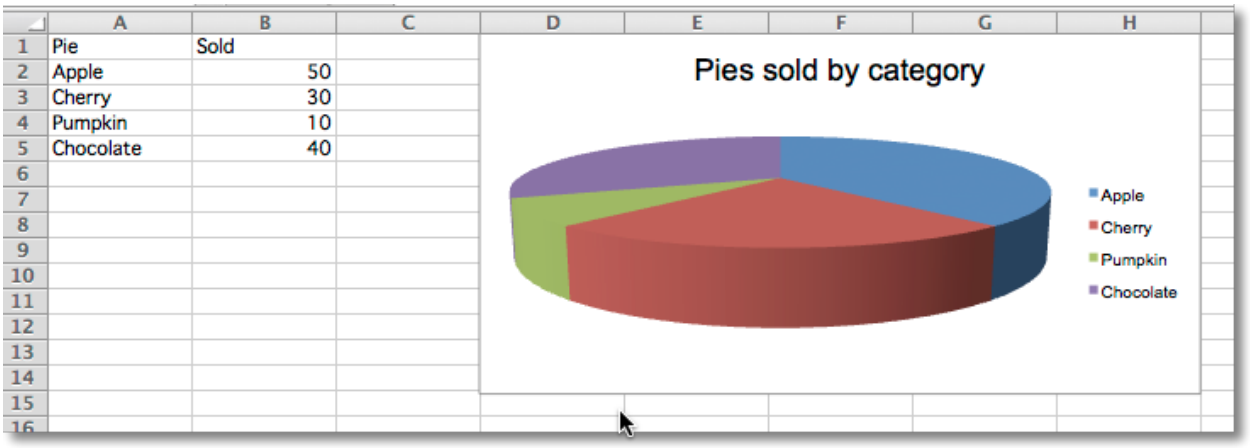
Projected pie charts extract some slices from a pie chart and project them into a second pie or bar chart. This is useful when there are several smaller items in the data series. The chart can be split according percent, val(ue) or pos(ition). If nothing is set then the application decides which to use. In addition custom splits can be defined.


A
B
C
D
E
Page
Views
Search
95
Products
4
Offers
0,5
Sales
0,5
Views
Search
Products
Offers
Sales
Views
Search
Products
Offers
Sales

3D Pie Charts

Pie charts can also be created with a 3D effect.

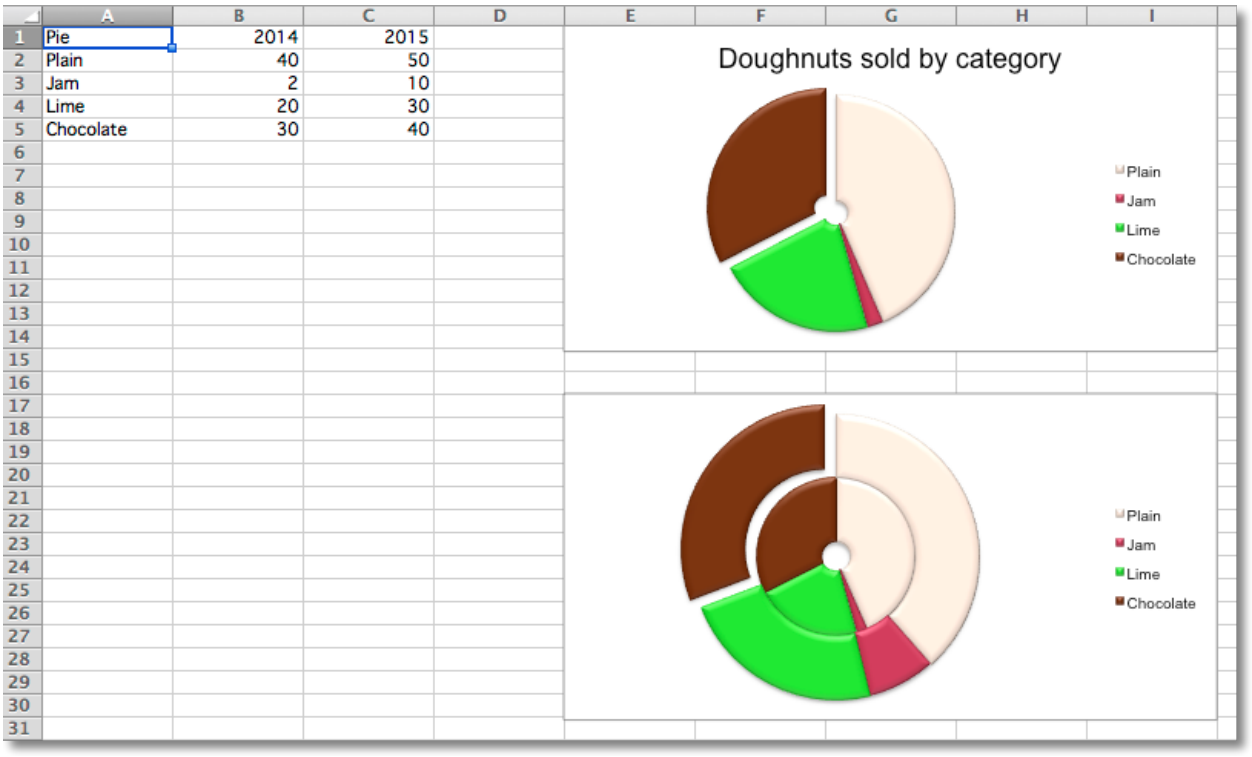
```
from openpyxl import Workbook

from openpyxl.chart import (
    PieChart3D,
    Reference
)

data = [
    ['Pie', 'Sold'],
    ['Apple', 50],
    ['Cherry', 30],
    ['Pumpkin', 10],
    ['Chocolate', 40],
]

wb = Workbook()
ws = wb.active

for row in data:
    ws.append(row)

pie = PieChart3D()
labels = Reference(ws, min_col=1, min_row=2, max_row=5)
data = Reference(ws, min_col=2, min_row=1, max_row=5)
pie.add_data(data, titles_from_data=True)
pie.set_categories(labels)
pie.title = "Pies sold by category"


ws.add_chart(pie, "D1")

wb.save("pie3D.xlsx")
```



Doughnut Charts

Doughnut charts are similar to pie charts except that they use a ring instead of a circle. They can also plot several series of data as concentric rings.

```
from openpyxl import Workbook

from openpyxl.chart import (
    DoughnutChart,
    Reference,
    Series,
)
from openpyxl.chart.series import DataPoint

data = [
    ['Pie', 2014, 2015],
    ['Plain', 40, 50],
    ['Jam', 2, 10],
    ['Lime', 20, 30],
    ['Chocolate', 30, 40],
]

wb = Workbook()
ws = wb.active

for row in data:
    ws.append(row)

chart = DoughnutChart()
labels = Reference(ws, min_col=1, min_row=2, max_row=5)
data = Reference(ws, min_col=2, min_row=1, max_row=5)
chart.add_data(data, titles_from_data=True)
chart.set_categories(labels)
chart.title = "Doughnuts sold by category"
chart.style = 26

# Cut the first slice out of the doughnut
slices = [DataPoint(idx=i) for i in range(4)]
plain, jam, lime, chocolate = slices
chart.series[0].data_points = slices
plain.graphicalProperties.solidFill = "FAE1D0"
jam.graphicalProperties.solidFill = "BB2244"
lime.graphicalProperties.solidFill = "22DD22"
chocolate.graphicalProperties.solidFill = "61210B"
chocolate.explosion = 10

ws.add_chart(chart, "E1")

from copy import deepcopy

chart2 = deepcopy(chart)
chart2.title = None
data = Reference(ws, min_col=3, min_row=1, max_row=5)
series2 = Series(data, title_from_data=True)
series2.data_points = slices
chart2.series.append(series2)

ws.add_chart(chart2, "E17")

wb.save("doughnut.xlsx")
```



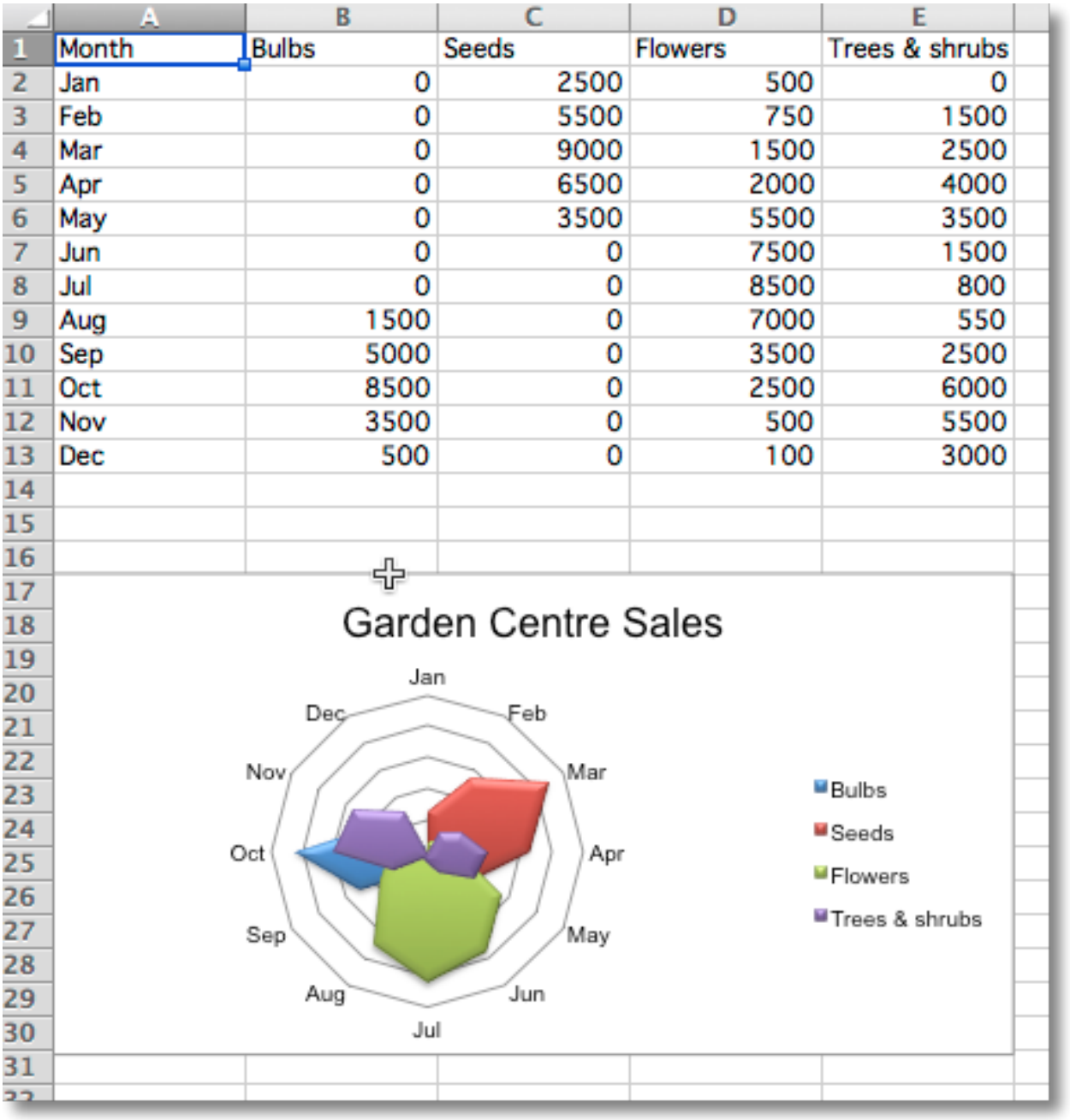
Radar Charts

Data that is arranged in columns or rows on a worksheet can be plotted in a radar chart. Radar charts compare the aggregate values of multiple data series. It is effectively a projection of an area chart on a circular x-axis.

There are two types of radar chart: standard, where the area is marked with a line; and filled where the where the whole area is filled. The additional type "marker" has no effect. If markers are desired these can be set for the relevant series.

```
from openpyxl import Workbook
from openpyxl.chart import (
    RadarChart,
    Reference,
)

wb = Workbook()
ws = wb.active

rows = [
    ['Month', "Bulbs", "Seeds", "Flowers", "Trees & shrubs"],
    ['Jan', 0, 2500, 500, 0,],
    ['Feb', 0, 5500, 750, 1500],
    ['Mar', 0, 9000, 1500, 2500],
    ['Apr', 0, 6500, 2000, 4000],
    ['May', 0, 3500, 5500, 3500],
    ['Jun', 0, 0, 7500, 1500],
    ['Jul', 0, 0, 8500, 800],
    ['Aug', 1500, 0, 7000, 550],
    ['Sep', 5000, 0, 3500, 2500],
    ['Oct', 8500, 0, 2500, 6000],
    ['Nov', 3500, 0, 500, 5500],
```

```
    ['Dec', 500, 0, 100, 3000 ],
]

for row in rows:
    ws.append(row)

chart = RadarChart()
chart.type = "filled"
labels = Reference(ws, min_col=1, min_row=2, max_row=13)
data = Reference(ws, min_col=2, max_col=5, min_row=1, max_row=13)
chart.add_data(data, titles_from_data=True)
chart.set_categories(labels)
chart.style = 26
chart.title = "Garden Centre Sales"
chart.y_axis.delete = True

ws.add_chart(chart, "A17")

wb.save("radar.xlsx")
```



	A	B	C	D	E
1	Month	Bulbs	Seeds	Flowers	Trees & shrubs
2	Jan	0	2500	500	0
3	Feb	0	5500	750	1500
4	Mar	0	9000	1500	2500
5	Apr	0	6500	2000	4000
6	May	0	3500	5500	3500
7	Jun	0	0	7500	1500
8	Jul	0	0	8500	800
9	Aug	1500	0	7000	550
10	Sep	5000	0	3500	2500
11	Oct	8500	0	2500	6000
12	Nov	3500	0	500	5500
13	Dec	500	0	100	3000

Stock Charts

Data that is arranged in columns or rows in a specific order on a worksheet can be plotted in a stock chart. As its name implies, a stock chart is most often used to illustrate the fluctuation of stock prices. However, this chart may also be used for scientific data. For example, you could use a stock chart to indicate the fluctuation of daily or annual temperatures. You must organize your data in the correct order to create stock charts.

The way stock chart data is organized in the worksheet is very important. For example, to create a simple high-low-close stock chart, you should arrange your data with High, Low, and Close entered as column headings, in that order.

Although stock charts are a distinct type, the various types are just shortcuts for particular formatting options:

- high-low-close is essentially a line chart with no lines and the marker set to XYZ. It also sets hiLoLines to True
- open-high-low-close is the as a high-low-close chart with the marker for each data point set to XZZ and upDownLines.

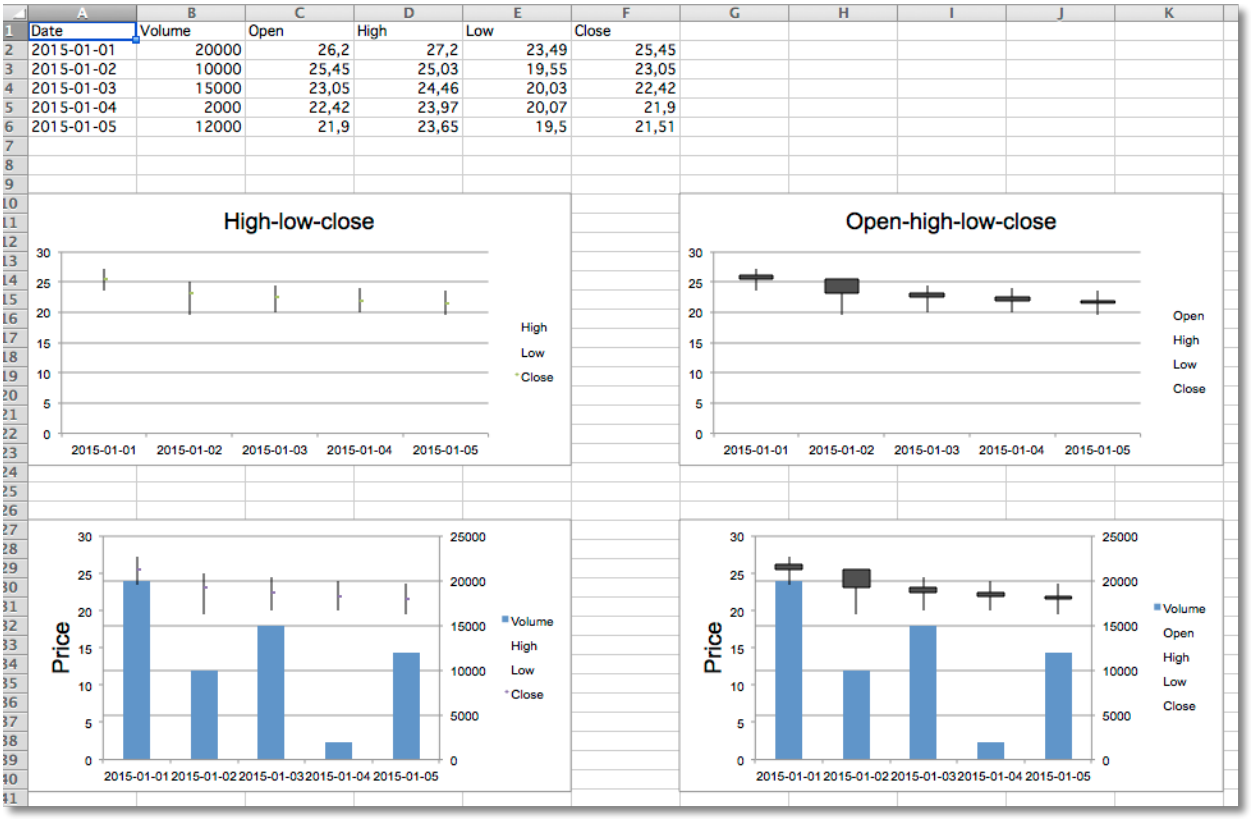
Volume can be added by combining the stock chart with a bar chart for the volume.

```
from datetime import date

from openpyxl import Workbook

from openpyxl.chart import (
    BarChart,
    StockChart,
    Reference,
    Series,
)
from openpyxl.chart.axis import DateAxis, ChartLines
from openpyxl.chart.updown_bars import UpDownBars

wb = Workbook()
ws = wb.active

rows = [
   ['Date',      'Volume','Open', 'High', 'Low', 'Close'],
   ['2015-01-01', 20000,    26.2, 27.20, 23.49, 25.45,  ],
   ['2015-01-02', 10000,    25.45, 25.03, 19.55, 23.05, ],
   ['2015-01-03', 15000,    23.05, 24.46, 20.03, 22.42, ],
   ['2015-01-04', 2000,     22.42, 23.97, 20.07, 21.90, ],
   ['2015-01-05', 12000,    21.9, 23.65, 19.50, 21.51,  ],
]

for row in rows:
    ws.append(row)

# High-low-close
c1 = StockChart()
labels = Reference(ws, min_col=1, min_row=2, max_row=6)
data = Reference(ws, min_col=4, max_col=6, min_row=1, max_row=6)
c1.add_data(data, titles_from_data=True)
c1.set_categories(labels)
for s in c1.series:
    s.graphicalProperties.line.noFill = True
# marker for close
s.marker.symbol = "dot"
s.marker.size = 5
c1.title = "High-low-close"
c1.hiLowLines = ChartLines()

# Excel is broken and needs a cache of values in order to display hiLoLines :-/
from openpyxl.chart.data_source import NumData, NumVal
pts = [NumVal(idx=i) for i in range(len(data) - 1)]
cache = NumData(pt=pts)
c1.series[-1].val.numRef.numCache = cache

ws.add_chart(c1, "A10")

# Open-high-low-close
```

```
c2 = StockChart()
data = Reference(ws, min_col=3, max_col=6, min_row=1, max_row=6)
c2.add_data(data, titles_from_data=True)
c2.set_categories(labels)
for s in c2.series:
    s.graphicalProperties.line.noFill = True
c2.hiLowLines = ChartLines()
c2.upDownBars = UpDownBars()
c2.title = "Open-high-low-close"

# add dummy cache
c2.series[-1].val.numRef.numCache = cache

ws.add_chart(c2, "G10")

# Create bar chart for volume

bar = BarChart()
data =  Reference(ws, min_col=2, min_row=1, max_row=6)
bar.add_data(data, titles_from_data=True)
bar.set_categories(labels)

from copy import deepcopy

# Volume-high-low-close
b1 = deepcopy(bar)
c3 = deepcopy(c1)
c3.y_axis.majorGridlines = None
c3.y_axis.title = "Price"
b1.y_axis.axId = 20
b1.z_axis = c3.y_axis
b1.y_axis.crosses = "max"
b1 += c3

c3.title = "High low close volume"

ws.add_chart(b1, "A27")

## Volume-open-high-low-close
b2 = deepcopy(bar)
c4 = deepcopy(c2)
c4.y_axis.majorGridlines = None
c4.y_axis.title = "Price"
b2.y_axis.axId = 20
b2.z_axis = c4.y_axis
b2.y_axis.crosses = "max"
b2 += c4

ws.add_chart(b2, "G27")

wb.save("stock.xlsx")
```

Warning: Due to a bug in Excel high-low lines will only be shown if at least one of the data series has some dummy values. This can be done with the following hack:

```
from openpyxl.chart.data_source import NumData, NumVal
pts = [NumVal(idx=i) for i in range(len(data) - 1)]
cache = NumData(pt=pts)
c1.series[-1].val.numRef.numCache = cache
```



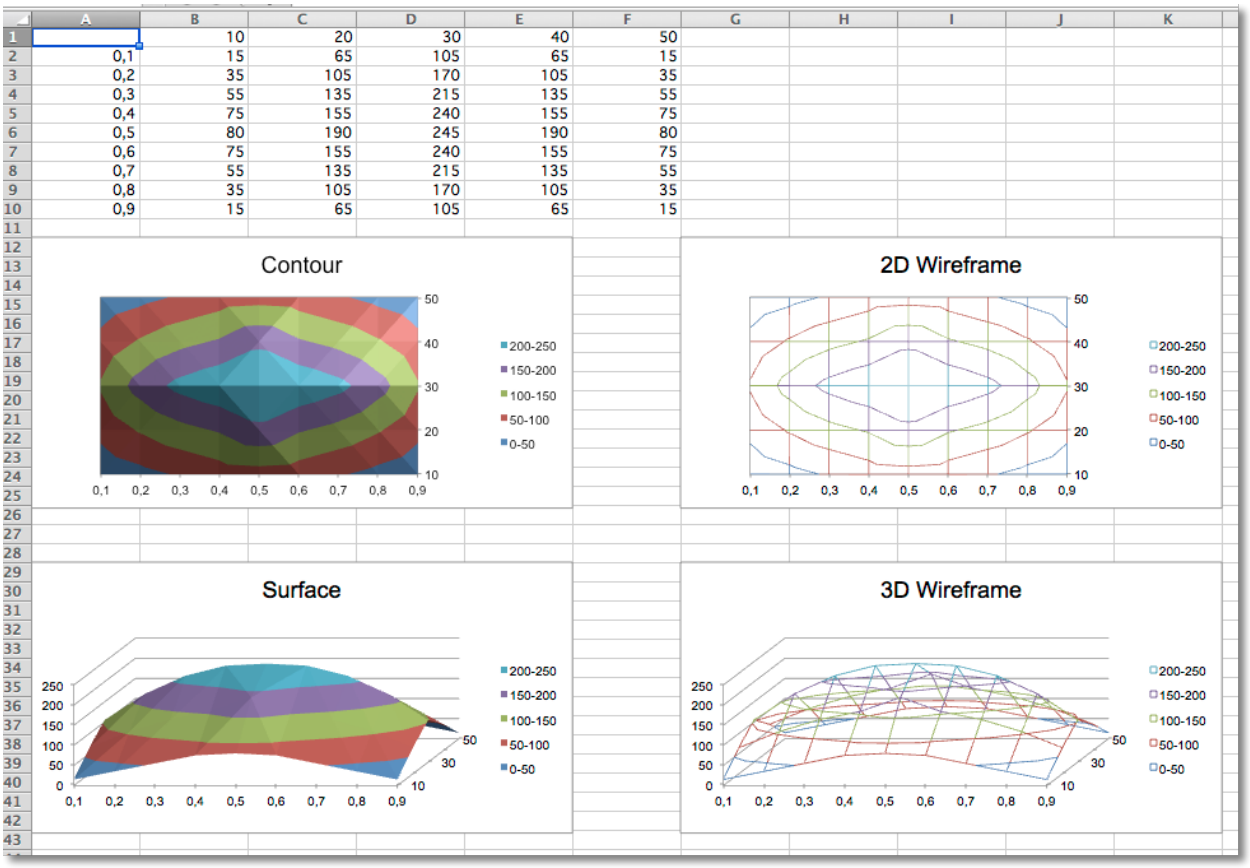
Surface charts

Data that is arranged in columns or rows on a worksheet can be plotted in a surface chart. A surface chart is useful when you want to find optimum combinations between two sets of data. As in a topographic map, colors and patterns indicate areas that are in the same range of values.

By default all surface charts are 3D. 2D wireframe and contour charts are created by setting the rotation and perspective.

```
from openpyxl import Workbook
from openpyxl.chart import (
    SurfaceChart,
    SurfaceChart3D,
    Reference,
    Series,
)
from openpyxl.chart.axis import SeriesAxis

wb = Workbook()
ws = wb.active
```

```
data = [
    [None, 10, 20, 30, 40, 50,],
    [0.1, 15, 65, 105, 65, 15,],
    [0.2, 35, 105, 170, 105, 35,],
    [0.3, 55, 135, 215, 135, 55,],
    [0.4, 75, 155, 240, 155, 75,],
    [0.5, 80, 190, 245, 190, 80,],
    [0.6, 75, 155, 240, 155, 75,],
    [0.7, 55, 135, 215, 135, 55,],
    [0.8, 35, 105, 170, 105, 35,],
    [0.9, 15, 65, 105, 65, 15],
]

for row in data:
    ws.append(row)


c1 = SurfaceChart()
ref = Reference(ws, min_col=2, max_col=6, min_row=1, max_row=10)
labels = Reference(ws, min_col=1, min_row=2, max_row=10)
c1.add_data(ref, titles_from_data=True)
c1.set_categories(labels)
c1.title = "Contour"

ws.add_chart(c1, "A12")

from copy import deepcopy

# wireframe
c2 = deepcopy(c1)
c2.wireframe = True
c2.title = "2D Wireframe"

ws.add_chart(c2, "G12")

# 3D Surface
c3 = SurfaceChart3D()
c3.add_data(ref, titles_from_data=True)
c3.set_categories(labels)
c3.title = "Surface"

ws.add_chart(c3, "A29")

c4 = deepcopy(c3)
c4.wireframe = True
c4.title = "3D Wireframe"

ws.add_chart(c4, "G29")

wb.save("surface.xlsx")
```



Creating a chart

Charts are composed of at least one series of one or more data points. Series themselves are comprised of references to cell ranges.

```
>>> from openpyxl import Workbook
>>> wb = Workbook()
>>> ws = wb.active
>>> for i in range(10):
...     ws.append([i])
>>>
>>> from openpyxl.chart import BarChart, Reference, Series
>>> values = Reference(ws, min_col=1, min_row=1, max_col=1, max_row=10)
>>> chart = BarChart()
>>> chart.add_data(values)
>>> ws.add_chart(chart, "E15")
>>> wb.save("SampleChart.xlsx")
```

By default the top-left corner of a chart is anchored to cell E15 and the size is 15 x 7.5 cm (approximately 5 columns by 14 rows). This can be changed by setting the *anchor*, *width* and *height* properties of the chart. The actual size will depend on operating system and device. Other anchors are possible see `openpyxl.drawing.spreadsheet_drawing` for further information.

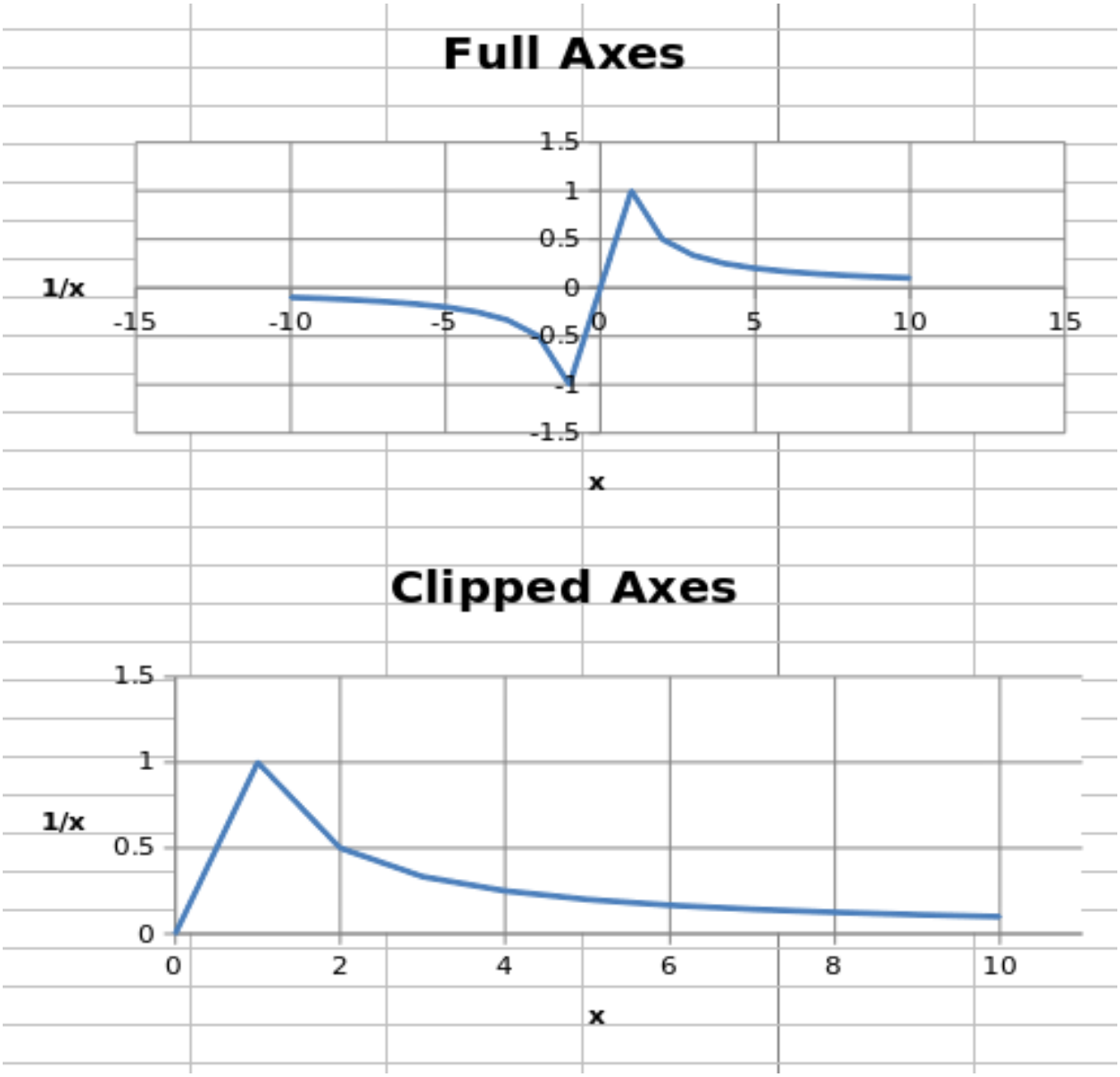
Working with axes

Axis Limits and Scale

Minima and Maxima

Axis minimum and maximum values can be set manually to display specific regions on a chart.

```
from openpyxl import Workbook
from openpyxl.chart import (
    ScatterChart,
    Reference,
    Series,
)

wb = Workbook()
ws = wb.active

ws.append(['X', '1/X'])
for x in range(-10, 11):
    if x:
        ws.append([x, 1.0 / x])

chart1 = ScatterChart()
chart1.title = "Full Axes"
chart1.x_axis.title = 'x'
chart1.y_axis.title = '1/x'
chart1.legend = None

chart2 = ScatterChart()
chart2.title = "Clipped Axes"
chart2.x_axis.title = 'x'
chart2.y_axis.title = '1/x'
chart2.legend = None

chart2.x_axis.scaling.min = 0
chart2.y_axis.scaling.min = 0
chart2.x_axis.scaling.max = 11
chart2.y_axis.scaling.max = 1.5

x = Reference(ws, min_col=1, min_row=2, max_row=22)
y = Reference(ws, min_col=2, min_row=2, max_row=22)
s = Series(y, xvalues=x)
chart1.append(s)
chart2.append(s)

ws.add_chart(chart1, "C1")
ws.add_chart(chart2, "C15")

wb.save("minmax.xlsx")
```



Note: In some cases such as the one shown, setting the axis limits is effectively equivalent to displaying a sub-range of the data. For large datasets, rendering of scatter plots (and possibly others) will be much faster when using subsets of the data rather than axis limits in both Excel and Open/Libre Office.

Logarithmic Scaling

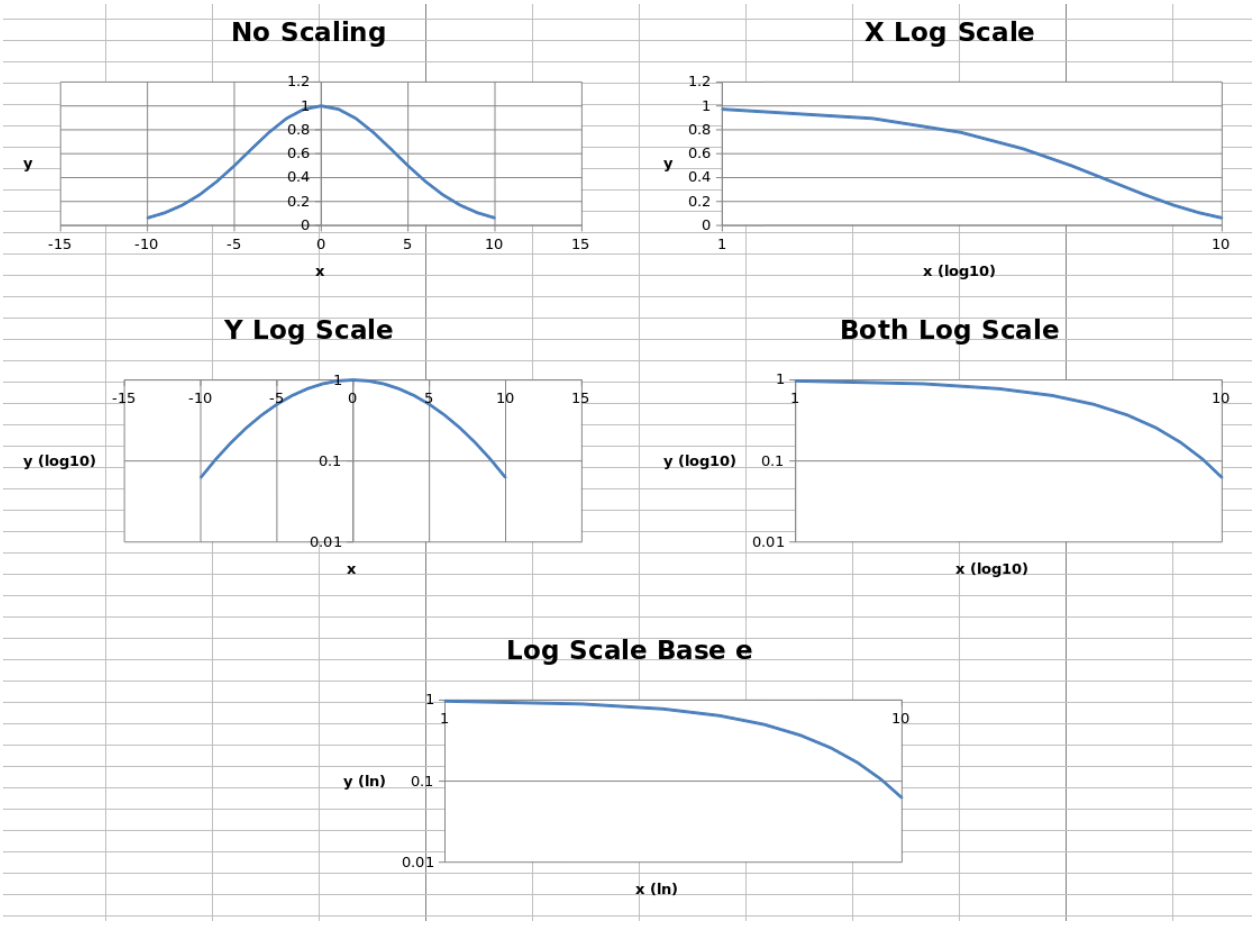
Both the x- and y-axes can be scaled logarithmically. The base of the logarithm can be set to any valid float. If the x-axis is scaled logarithmically, negative values in the domain will be discarded.

```
from openpyxl import Workbook
from openpyxl.chart import (
    ScatterChart,
    Reference,
```

```
    Series,
)
import math

wb = Workbook()
ws = wb.active

ws.append(['X', 'Gaussian'])
for i, x in enumerate(range(-10, 11)):
    ws.append([x, "=EXP(-(($A${row}/6)^2))".format(row = i + 2)])

chart1 = ScatterChart()
chart1.title = "No Scaling"
chart1.x_axis.title = 'x'
chart1.y_axis.title = 'y'
chart1.legend = None

chart2 = ScatterChart()
chart2.title = "X Log Scale"
chart2.x_axis.title = 'x (log10)'
chart2.y_axis.title = 'y'
chart2.legend = None
chart2.x_axis.scaling.logBase = 10

chart3 = ScatterChart()
chart3.title = "Y Log Scale"
chart3.x_axis.title = 'x'
chart3.y_axis.title = 'y (log10)'
chart3.legend = None
chart3.y_axis.scaling.logBase = 10

chart4 = ScatterChart()
chart4.title = "Both Log Scale"
chart4.x_axis.title = 'x (log10)'
chart4.y_axis.title = 'y (log10)'
chart4.legend = None
chart4.x_axis.scaling.logBase = 10
chart4.y_axis.scaling.logBase = 10

chart5 = ScatterChart()
chart5.title = "Log Scale Base e"
chart5.x_axis.title = 'x (ln)'
chart5.y_axis.title = 'y (ln)'
chart5.legend = None
chart5.x_axis.scaling.logBase = math.e
chart5.y_axis.scaling.logBase = math.e

x = Reference(ws, min_col=1, min_row=2, max_row=22)
y = Reference(ws, min_col=2, min_row=2, max_row=22)
s = Series(y, xvalues=x)
chart1.append(s)
chart2.append(s)
chart3.append(s)
chart4.append(s)
chart5.append(s)

ws.add_chart(chart1, "C1")
ws.add_chart(chart2, "I1")
```

```
ws.add_chart(chart3, "C15")
ws.add_chart(chart4, "I15")
ws.add_chart(chart5, "F30")

wb.save("log.xlsx")
```

This produces five charts that look something like this:



The first four charts show the same data unscaled, scaled logarithmically in each axis and in both axes, with the logarithm base set to 10. The final chart shows the same data with both axes scaled, but the base of the logarithm set to `e`.

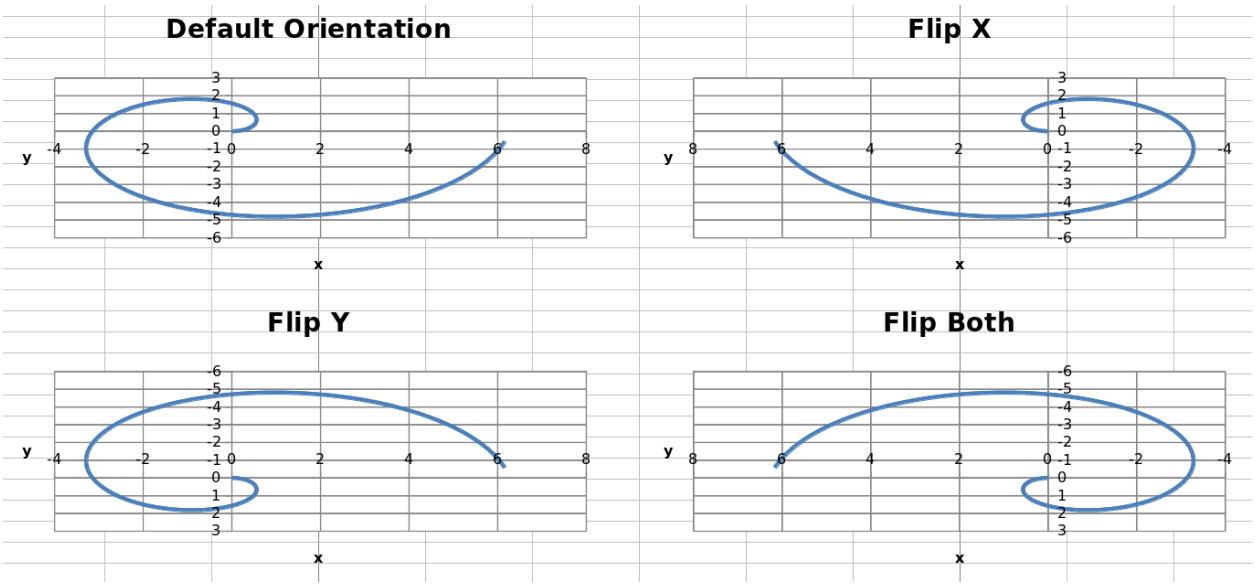
Axis Orientation

Axes can be displayed "normally" or in reverse. Axis orientation is controlled by the scaling `orientation` property, which can have a value of either `'minMax'` for normal orientation or `'maxMin'` for reversed.

```
from openpyxl import Workbook
from openpyxl.chart import (
    ScatterChart,
    Reference,
    Series,
)
```

```
wb = Workbook()
ws = wb.active

ws["A1"] = "Archimedean Spiral"
ws.append(["T", "X", "Y"])
for i, t in enumerate(range(100)):
    ws.append([t / 16.0, "=$A${row}*COS($A${row})".format(row = i + 3),
                         "=$A${row}*SIN($A${row})".format(row = i + 3)])

chart1 = ScatterChart()
chart1.title = "Default Orientation"
chart1.x_axis.title = 'x'
chart1.y_axis.title = 'y'
chart1.legend = None

chart2 = ScatterChart()
chart2.title = "Flip X"
chart2.x_axis.title = 'x'
chart2.y_axis.title = 'y'
chart2.legend = None
chart2.x_axis.scaling.orientation = "maxMin"
chart2.y_axis.scaling.orientation = "minMax"

chart3 = ScatterChart()
chart3.title = "Flip Y"
chart3.x_axis.title = 'x'
chart3.y_axis.title = 'y'
chart3.legend = None
chart3.x_axis.scaling.orientation = "minMax"
chart3.y_axis.scaling.orientation = "maxMin"

chart4 = ScatterChart()
chart4.title = "Flip Both"
chart4.x_axis.title = 'x'
chart4.y_axis.title = 'y'
chart4.legend = None
chart4.x_axis.scaling.orientation = "maxMin"
chart4.y_axis.scaling.orientation = "maxMin"

x = Reference(ws, min_col=2, min_row=2, max_row=102)
y = Reference(ws, min_col=3, min_row=2, max_row=102)
s = Series(y, xvalues=x)
chart1.append(s)
chart2.append(s)
chart3.append(s)
chart4.append(s)

ws.add_chart(chart1, "D1")
ws.add_chart(chart2, "J1")
ws.add_chart(chart3, "D15")
ws.add_chart(chart4, "J15")

wb.save("orientation.xlsx")
```

This produces four charts with the axes in each possible combination of orientations that look something like this:



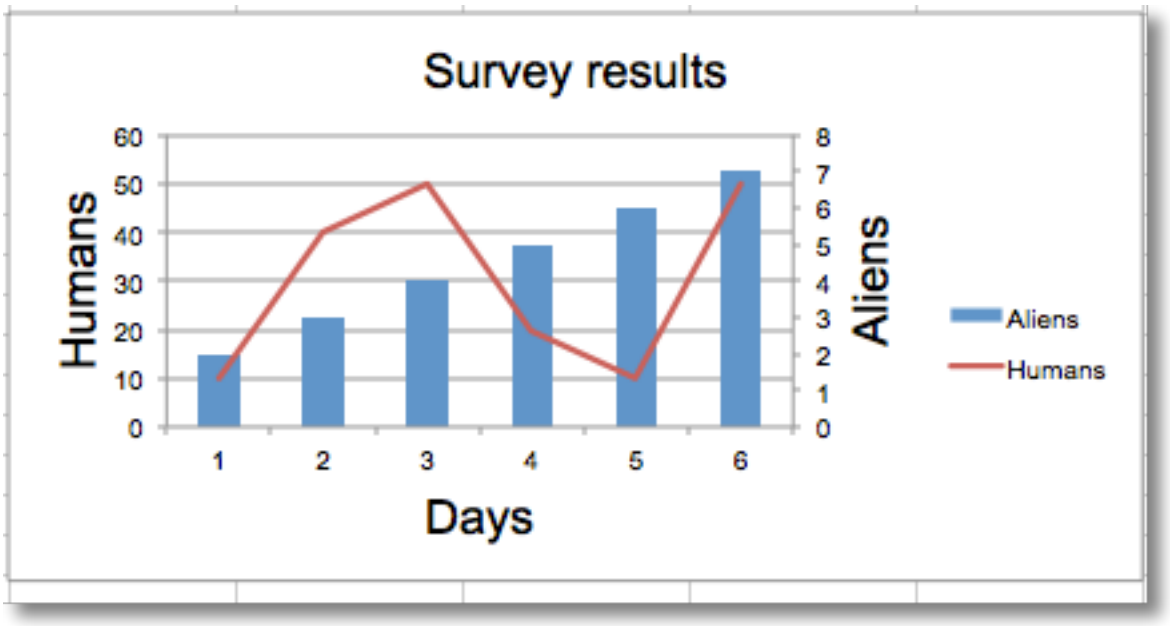
Adding a second axis

Adding a second axis actually involves creating a second chart that shares a common x-axis with the first chart but has a separate y-axis.

```
from openpyxl import Workbook
from openpyxl.chart import (
    LineChart,
    BarChart,
    Reference,
    Series,
)

wb = Workbook()
ws = wb.active

rows = [
    ['Aliens', 2, 3, 4, 5, 6, 7],
    ['Humans', 10, 40, 50, 20, 10, 50],
]

for row in rows:
    ws.append(row)

c1 = BarChart()
v1 = Reference(ws, min_col=1, min_row=1, max_col=7)
c1.add_data(v1, titles_from_data=True, from_rows=True)

c1.x_axis.title = 'Days'
c1.y_axis.title = 'Aliens'
c1.y_axis.majorGridlines = None
c1.title = 'Survey results'


# Create a second chart
c2 = LineChart()
```

```
v2 = Reference(ws, min_col=1, min_row=2, max_col=7)
c2.add_data(v2, titles_from_data=True, from_rows=True)
c2.y_axis.axId = 200
c2.y_axis.title = "Humans"

# Display y-axis of the second chart on the right by setting it to cross the x-axis
↪at its maximum
c1.y_axis.crosses = "max"
c1 += c2

ws.add_chart(c1, "D4")

wb.save("secondary.xlsx")
```

This produces a combined line and bar chart looking something like this:



Change the chart layout

Changing the layout of plot area and legend

The layout of the chart within the canvas can be set by using the layout property an instance of a layout class.

Chart layout

Size and position

The chart can be positioned within its container. `x` and `y` adjust position, `w` and `h` adjust the size . The units are proportions of the container. A chart cannot be positioned outside of its container and the width and height are the dominant constraints: if x + w > 1, then x = 1 - w.

x is the horizontal position from the left y is the vertical position the top h is the height of the chart relative to its container w is the width of the box

Mode

In addition to the size and position the mode for the relevant attribute can also be set to either *factor* or *edge*. Factor is the default:

```
layout.xMode = edge
```

Target

The layoutTarget can be set to `outer` or `inner`. The default is `outer`:

```
layout.layoutTarget = inner
```

Legend layout

The position of the legend can be controlled either by setting its position: `r`, `l`, `t`, `b`, and `tr`, for right, left, top, bottom and top right respectively. The default is `r`.

```
legend.position = 'tr'
```

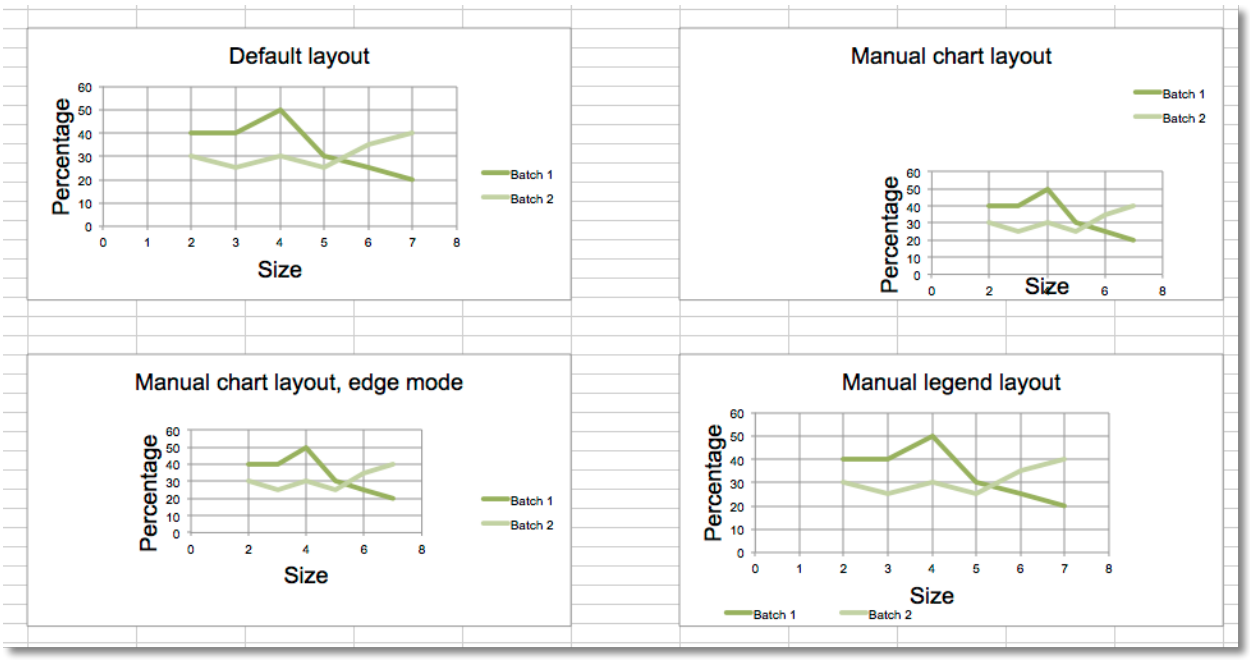
or applying a manual layout:

```
legend.layout = ManualLayout()
```

```
from openpyxl import Workbook, load_workbook
from openpyxl.chart import ScatterChart, Series, Reference
from openpyxl.chart.layout import Layout, ManualLayout

wb = Workbook()
ws = wb.active

rows = [
    ['Size', 'Batch 1', 'Batch 2'],
    [2, 40, 30],
    [3, 40, 25],
    [4, 50, 30],
    [5, 30, 25],
    [6, 25, 35],
    [7, 20, 40],
]

for row in rows:
    ws.append(row)

ch1 = ScatterChart()
xvalues = Reference(ws, min_col=1, min_row=2, max_row=7)
for i in range(2, 4):
    values = Reference(ws, min_col=i, min_row=1, max_row=7)
    series = Series(values, xvalues, title_from_data=True)
    ch1.series.append(series)


ch1.title = "Default layout"
ch1.style = 13
```

```
ch1.x_axis.title = 'Size'
ch1.y_axis.title = 'Percentage'
ch1.legend.position = 'r'

ws.add_chart(ch1, "B10")

from copy import deepcopy

# Half-size chart, bottom right
ch2 = deepcopy(ch1)
ch2.title = "Manual chart layout"
ch2.legend.position = "tr"
ch2.layout=Layout(
    manualLayout=ManualLayout(
        x=0.25, y=0.25,
        h=0.5, w=0.5,
    )
)
ws.add_chart(ch2, "H10")

# Half-size chart, centred
ch3 = deepcopy(ch1)
ch3.layout = Layout(
    ManualLayout(
    x=0.25, y=0.25,
    h=0.5, w=0.5,
    xMode="edge",
    yMode="edge",
    )
)
ch3.title = "Manual chart layout, edge mode"
ws.add_chart(ch3, "B27")

# Manually position the legend bottom left
ch4 = deepcopy(ch1)
ch4.title = "Manual legend layout"
ch4.legend.layout = Layout(
    manualLayout=ManualLayout(
        yMode='edge',
        xMode='edge',
        x=0, y=0.9,
        h=0.1, w=0.5
    )
)

ws.add_chart(ch4, "H27")

wb.save("chart_layout.xlsx")
```

This produces four charts illustrating various possibilities:



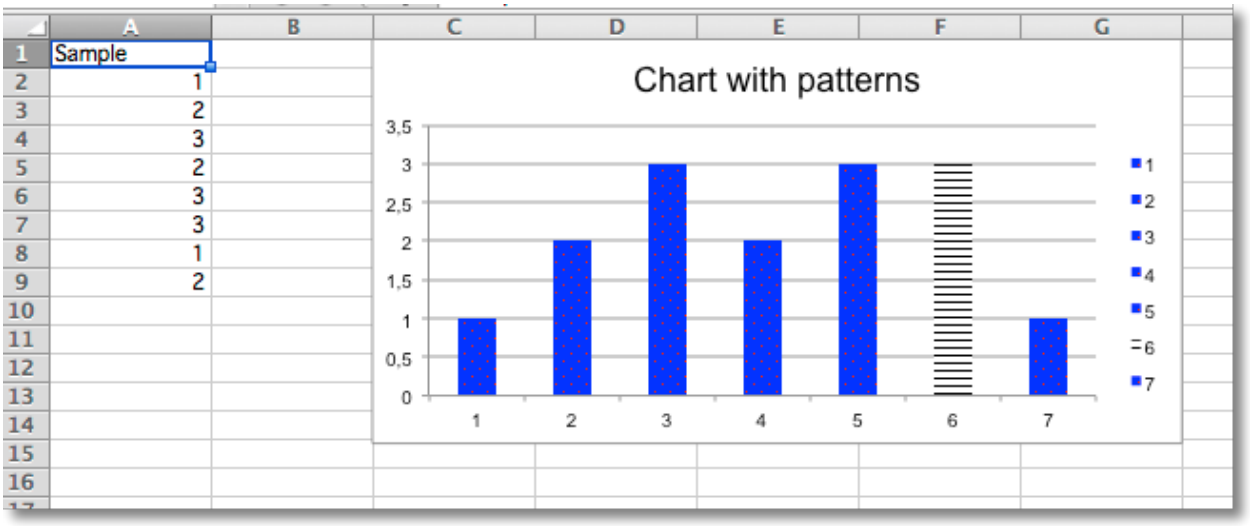
Styling charts

Adding Patterns

Whole data series and individual data points can be extensively styled through the *graphicalProperties*. Getting things just right may take some time.

```
from openpyxl import Workbook
from openpyxl.chart import BarChart, Reference
from openpyxl.chart.marker import DataPoint

from openpyxl.drawing.fill import PatternFillProperties, ColorChoice

wb = Workbook()
ws = wb.active

rows = [
    ("Sample",),
    (1,),
    (2,),
    (3,),
    (2,),
    (3,),
    (3,),
    (1,),
    (2,),
]

for r in rows:
    ws.append(r)


c = BarChart()
```

```
data = Reference(ws, min_col=1, min_row=1, max_row=8)
c.add_data(data, titles_from_data=True)
c.title = "Chart with patterns"

# set a pattern for the whole series
series = c.series[0]
fill =  PatternFillProperties(prst="pct5")
fill.foreground = ColorChoice(prstClr="red")
fill.background = ColorChoice(prstClr="blue")
series.graphicalProperties.pattFill = fill

# set a pattern for a 6th data point (0-indexed)
pt = DataPoint(idx=5)
pt.graphicalProperties.pattFill = PatternFillProperties(prst="ltHorz")
series.dPt.append(pt)

ws.add_chart(c, "C1")

wb.save("pattern.xlsx")
```



Advanced charts

Charts can be combined to create new charts:

Gauge Charts

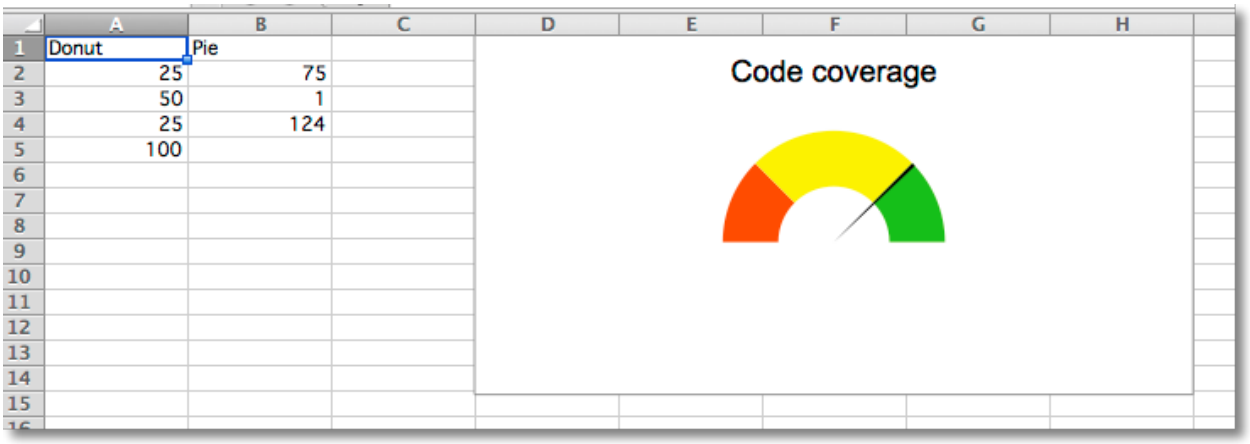
Gauge charts combine a pie chart and a doughnut chart to create a "gauge". The first chart is a doughnut chart with four slices. The first three slices correspond to the colours of the gauge; the fourth slice, which is half of the doughnut, is made invisible.

A pie chart containing three slices is added. The first and third slice are invisible so that the second slice can act as the needle on the gauge.

The effects are done using the graphical properties of individual data points in a data series.

```
from openpyxl import Workbook
```

```
from openpyxl.chart import PieChart, DoughnutChart, Series, Reference
from openpyxl.chart.series import DataPoint


data = [
    ["Donut", "Pie"],
    [25, 75],
    [50, 1],
    [25, 124],
    [100],
]

# based on http://www.excel-easy.com/examples/gauge-chart.html

wb = Workbook()
ws = wb.active
for row in data:
    ws.append(row)

# First chart is a doughnut chart
c1 = DoughnutChart(firstSliceAng=270, holeSize=50)
c1.title = "Code coverage"
c1.legend = None

ref = Reference(ws, min_col=1, min_row=2, max_row=5)
s1 = Series(ref, title_from_data=False)

slices = [DataPoint(idx=i) for i in range(4)]
slices[0].graphicalProperties.solidFill = "FF3300" # red
slices[1].graphicalProperties.solidFill = "FCF305" # yellow
slices[2].graphicalProperties.solidFill = "1FB714" # green
slices[3].graphicalProperties.noFill = True # invisible

s1.data_points = slices
c1.series = [s1]

# Second chart is a pie chart
c2 = PieChart(firstSliceAng=270)
c2.legend = None

ref = Reference(ws, min_col=2, min_row=2, max_col=2, max_row=4)
s2 = Series(ref, title_from_data=False)

slices = [DataPoint(idx=i) for i in range(3)]
slices[0].graphicalProperties.noFill = True # invisible
slices[1].graphicalProperties.solidFill = "000000" # black needle
slices[2].graphicalProperties.noFill = True # invisible
s2.data_points = slices
c2.series = [s2]

c1 += c2 # combine charts

ws.add_chart(c1, "D1")

wb.save("gauge.xlsx")
```



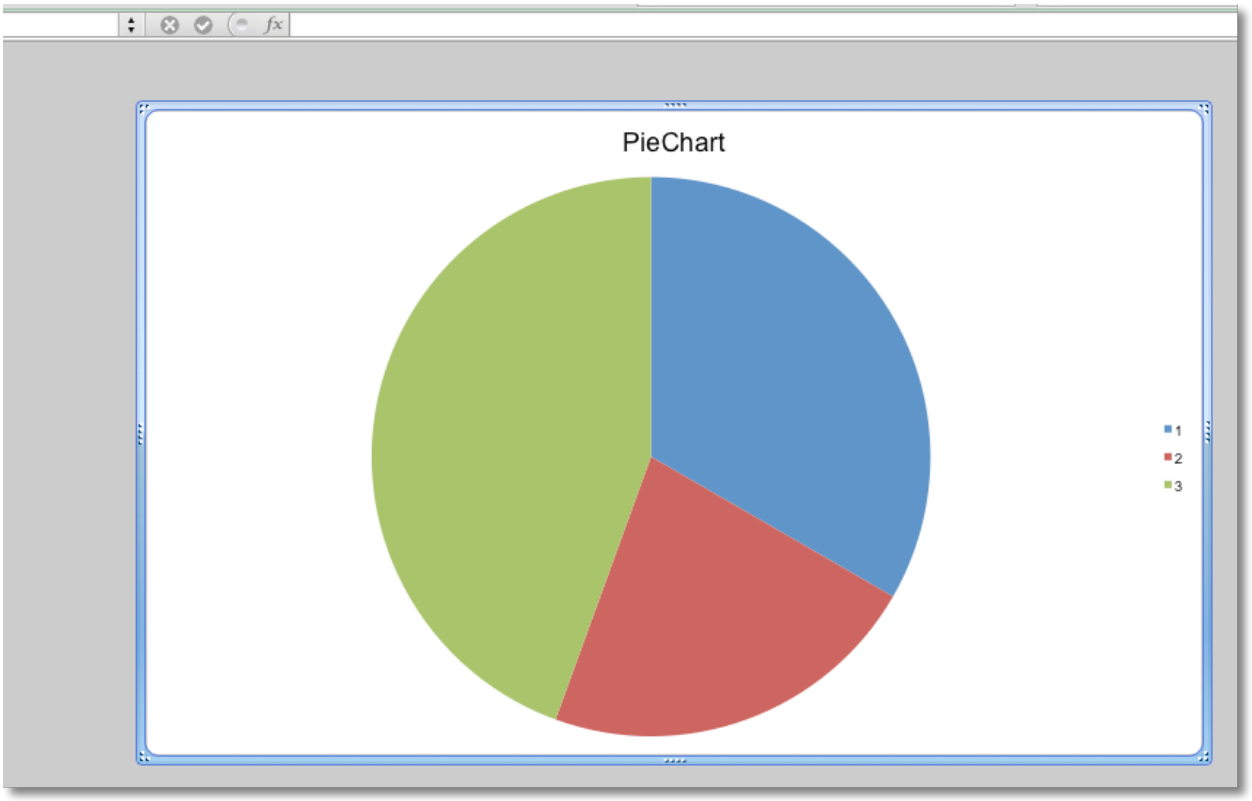
Using chartsheets

Charts can be added to special worksheets called chartsheets:

Chartsheets

Chartsheets are special worksheets which only contain charts. All the data for the chart must be on a different worksheet.

```
from openpyxl import Workbook

from openpyxl.chart import PieChart, Reference, Series

wb = Workbook()
ws = wb.active
cs = wb.create_chartsheet()

rows = [
    ["Bob", 3],
    ["Harry", 2],
    ["James", 4],
]

for row in rows:
    ws.append(row)


chart = PieChart()
labels = Reference(ws, min_col=1, min_row=1, max_row=3)
data = Reference(ws, min_col=2, min_row=1, max_row=3)
chart.series = (Series(data),)
chart.title = "PieChart"

cs.add_chart(chart)

wb.save("demo.xlsx")
```



7.5 Comments

7.5.1 Comments

Warning: Openpyxl currently supports the reading and writing of comment text only. Formatting information is lost. Comments are not currently supported if *use_iterators=True* is used.

Adding a comment to a cell

Comments have a text attribute and an author attribute, which must both be set

```
>>> from openpyxl import Workbook
>>> from openpyxl.comments import Comment
>>> wb = Workbook()
>>> ws = wb.active
>>> comment = ws["A1"].comment
>>> comment = Comment('This is the comment text', 'Comment Author')
>>> comment.text
'This is the comment text'
>>> comment.author
'Comment Author'
```

If you assign the same comment to multiple cells then openpyxl will automatically create copies

```
>>> from openpyxl import Workbook
>>> from openpyxl.comments import Comment
>>> wb=Workbook()
>>> ws=wb.active
>>> comment = Comment("Text", "Author")
>>> ws["A1"].comment = comment
>>> ws["B2"].comment = comment
>>> ws["A1"].comment is comment
True
>>> ws["B2"].comment is comment
False
```

Loading and saving comments

Comments present in a workbook when loaded are stored in the comment attribute of their respective cells automatically. Formatting information such as font size, bold and italics are lost, as are the original dimensions and position of the comment's container box.

Comments remaining in a workbook when it is saved are automatically saved to the workbook file.

7.6 Read/write large files

7.6.1 Read-only mode

Sometimes, you will need to open or write extremely large XLSX files, and the common routines in openpyxl won't be able to handle that load. Fortunately, there are two modes that enable you to read and write unlimited amounts of data with (near) constant memory consumption.

Introducing `openpyxl.worksheet.read_only.ReadOnlyWorksheet`:

```
from openpyxl import load_workbook
wb = load_workbook(filename='large_file.xlsx', read_only=True)
ws = wb['big_data']

for row in ws.rows:
    for cell in row:
        print(cell.value)
```

Warning:

- `openpyxl.worksheet.read_only.ReadOnlyWorksheet` is read-only

Cells returned are not regular `openpyxl.cell.cell.Cell` but `openpyxl.cell.read_only.ReadOnlyCell`.

Worksheet dimensions

Read-only mode relies on applications and libraries that created the file providing correct information about the worksheets, specifically the used part of it, known as the dimensions. Some applications set this incorrectly. You can check the apparent dimensions of a worksheet using *ws.calculate_dimension()*. If this returns a range that you know

is incorrect, say *A1:A1* then simply resetting the max_row and max_column attributes should allow you to work with the file:

```
ws.max_row = ws.max_column = None
```

7.6.2 Write-only mode

Here again, the regular `openpyxl.worksheet.worksheet.Worksheet` has been replaced by a faster alternative, the `openpyxl.writer.write_only.WriteOnlyWorksheet`. When you want to dump large amounts of data make sure you have *lxml* installed.

```
>>> from openpyxl import Workbook
>>> wb = Workbook(write_only=True)
>>> ws = wb.create_sheet()
>>>
>>> # now we'll fill it with 100 rows x 200 columns
>>>
>>> for irow in range(100):
...     ws.append(['%d' % i for i in range(200)])
>>> # save the file
>>> wb.save('new_big_file.xlsx')
```

If you want to have cells with styles or comments then use a `openpyxl.writer.write_only.WriteOnlyCell()`

```
>>> from openpyxl import Workbook
>>> wb = Workbook(write_only = True)
>>> ws = wb.create_sheet()
>>> from openpyxl.writer.write_only import WriteOnlyCell
>>> from openpyxl.comments import Comment
>>> from openpyxl.styles import Font
>>> cell = WriteOnlyCell(ws, value="hello world")
>>> cell.font = Font(name='Courier', size=36)
>>> cell.comment = Comment(text="A comment", author="Author's Name")
>>> ws.append([cell, 3.14, None])
>>> wb.save('write_only_file.xlsx')
```

This will create a write-only workbook with a single sheet, and append a row of 3 cells: one text cell with a custom font and a comment, a floating-point number, and an empty cell (which will be discarded anyway).

Warning:

- Unlike a normal workbook, a newly-created write-only workbook does not contain any worksheets; a worksheet must be specifically created with the `create_sheet()` method.
- In a write-only workbook, rows can only be added with `append()`. It is not possible to write (or read) cells at arbitrary locations with `cell()` or `iter_rows()`.
- It is able to export unlimited amount of data (even more than Excel can handle actually), while keeping memory usage under 10Mb.
- A write-only workbook can only be saved once. After that, every attempt to save the workbook or append() to an existing worksheet will raise an `openpyxl.utils.exceptions.WorkbookAlreadySaved` exception.

- Everything that appears in the file before the actual cell data must be created before cells are added because it must written to the file before then. For example, *freeze_panes* should be set before cells are added.

7.7 Working with styles

7.7.1 Working with styles

Introduction

Styles are used to change the look of your data while displayed on screen. They are also used to determine the formatting for numbers.

Styles can be applied to the following aspects:

- font to set font size, color, underlining, etc.
- fill to set a pattern or color gradient
- border to set borders on a cell
- cell alignment
- protection

The following are the default values

```
>>> from openpyxl.styles import PatternFill, Border, Side, Alignment, Protection, Font
>>> font = Font(name='Calibri',
...                 size=11,
...                 bold=False,
...                 italic=False,
...                 vertAlign=None,
...                 underline='none',
...                 strike=False,
...                 color='FF000000')
>>> fill = PatternFill(fill_type=None,
...                 start_color='FFFFFFFF',
...                 end_color='FF000000')
>>> border = Border(left=Side(border_style=None,
...                           color='FF000000'),
...                 right=Side(border_style=None,
...                            color='FF000000'),
...                 top=Side(border_style=None,
...                          color='FF000000'),
...                 bottom=Side(border_style=None,
...                             color='FF000000'),
...                 diagonal=Side(border_style=None,
...                               color='FF000000'),
...                 diagonal_direction=0,
...                 outline=Side(border_style=None,
...                              color='FF000000'),
...                 vertical=Side(border_style=None,
...                               color='FF000000'),
...                 horizontal=Side(border_style=None,
...                                 color='FF000000')
...                )
```

```
>>> alignment=Alignment(horizontal='general',
...                     vertical='bottom',
...                     text_rotation=0,
...                     wrap_text=False,
...                     shrink_to_fit=False,
...                     indent=0)
>>> number_format = 'General'
>>> protection = Protection(locked=True,
...                         hidden=False)
>>>
```

Cell Styles and Named Styles

There are two types of styles: cell styles and named styles, also known as style templates.

Cell Styles

Cell styles are shared between objects and once they have been assigned they cannot be changed. This stops unwanted side-effects such as changing the style for lots of cells when instead of only one.

```
>>> from openpyxl.styles import colors
>>> from openpyxl.styles import Font, Color
>>> from openpyxl import Workbook
>>> wb = Workbook()
>>> ws = wb.active
>>>
>>> a1 = ws['A1']
>>> d4 = ws['D4']
>>> ft = Font(color=colors.RED)
>>> a1.font = ft
>>> d4.font = ft
>>>
>>> a1.font.italic = True # is not allowed
>>>
>>> # If you want to change the color of a Font, you need to reassign it::
>>>
>>> a1.font = Font(color=colors.RED, italic=True) # the change only affects A1
```

Copying styles

Styles can also be copied

```
>>> from openpyxl.styles import Font
>>> from copy import copy
>>>
>>> ft1 = Font(name='Arial', size=14)
>>> ft2 = copy(ft1)
>>> ft2.name = "Tahoma"
>>> ft1.name
'Arial'
>>> ft2.name
'Tahoma'
>>> ft2.size # copied from the
14.0
```

Basic Font Colors

Colors are usually RGB or aRGB hexvalues. The *colors* module contains some handy constants

```
>>> from openpyxl.styles import Font
>>> from openpyxl.styles.colors import RED
>>> font = Font(color=RED)
>>> font = Font(color="FFBB00")
```

There is also support for legacy indexed colors as well as themes and tints

```
>>> from openpyxl.styles.colors import Color
>>> c = Color(indexed=32)
>>> c = Color(theme=6, tint=0.5)
```

Applying Styles

Styles are applied directly to cells

```
>>> from openpyxl.workbook import Workbook
>>> from openpyxl.styles import Font, Fill
>>> wb = Workbook()
>>> ws = wb.active
>>> c = ws['A1']
>>> c.font = Font(size=12)
```

Styles can also applied to columns and rows but note that this applies only to cells created (in Excel) after the file is closed. If you want to apply styles to entire rows and columns then you must apply the style to each cell yourself. This is a restriction of the file format:

```
>>> col = ws.column_dimensions['A']
>>> col.font = Font(bold=True)
>>> row = ws.row_dimensions[1]
>>> row.font = Font(underline="single")
```

Styling Merged Cells

Sometimes you want to format a range of cells as if they were a single object. Excel pretends that this is possible by merging cells (deleting all but the top-left cell) and then recreating them in order to apply pseudo-styles.

```
from openpyxl.styles import Border, Side, PatternFill, Font, GradientFill, Alignment
from openpyxl import Workbook


def style_range(ws, cell_range, border=Border(), fill=None, font=None,␣
↪alignment=None):
    """
    Apply styles to a range of cells as if they were a single cell.

    :param ws:  Excel worksheet instance
    :param range: An excel range to style (e.g. A1:F20)
    :param border: An openpyxl Border
```

```
    :param fill: An openpyxl PatternFill or GradientFill
    :param font: An openpyxl Font object
    """

    top = Border(top=border.top)
    left = Border(left=border.left)
    right = Border(right=border.right)
    bottom = Border(bottom=border.bottom)

    first_cell = ws[cell_range.split(":")[0]]
    if alignment:
        ws.merge_cells(cell_range)
        first_cell.alignment = alignment

    rows = ws[cell_range]
    if font:
        first_cell.font = font

    for cell in rows[0]:
        cell.border = cell.border + top
    for cell in rows[-1]:
        cell.border = cell.border + bottom

    for row in rows:
        l = row[0]
        r = row[-1]
        l.border = l.border + left
        r.border = r.border + right
        if fill:
            for c in row:
                c.fill = fill

wb = Workbook()
ws = wb.active
my_cell = ws['B2']
my_cell.value = "My Cell"
thin = Side(border_style="thin", color="000000")
double = Side(border_style="double", color="ff0000")

border = Border(top=double, left=thin, right=thin, bottom=double)
fill = PatternFill("solid", fgColor="DDDDDD")
fill = GradientFill(stop=("000000", "FFFFFF"))
font = Font(b=True, color="FF0000")
al = Alignment(horizontal="center", vertical="center")


style_range(ws, 'B2:F4', border=border, fill=fill, font=font, alignment=al)
wb.save("styled.xlsx")
```

Edit Page Setup

```
>>> from openpyxl.workbook import Workbook
>>>
>>> wb = Workbook()
>>> ws = wb.active
>>>
```

```
>>> ws.page_setup.orientation = ws.ORIENTATION_LANDSCAPE
>>> ws.page_setup.paperSize = ws.PAPERSIZE_TABLOID
>>> ws.page_setup.fitToHeight = 0
>>> ws.page_setup.fitToWidth = 1
```

Named Styles

In contrast to Cell Styles, Named Styles are mutable. They make sense when you want to apply formatting to lots of different cells at once. NB. once you have assigned a named style to a cell, additional changes to the style will **not** affect the cell.

Once a named style has been registered with a workbook, it can be referred to simply by name.

Creating a Named Style

```
>>> from openpyxl.styles import NamedStyle, Font, Border, Side
>>> highlight = NamedStyle(name="highlight")
>>> highlight.font = Font(bold=True, size=20)
>>> bd = Side(style='thick', color="000000")
>>> highlight.border = Border(left=bd, top=bd, right=bd, bottom=bd)
```

Once a named style has been created, it can be registered with the workbook:

```
>>> wb.add_named_style(highlight)
```

But named styles will also be registered automatically the first time they are assigned to a cell:

```
>>> ws['A1'].style = highlight
```

Once registered assign the style using just the name:

```
>>> ws['D5'].style = 'highlight'
```

Using builtin styles

The specification includes some builtin styles which can also be used. Unfortunately, the names for these styles are stored in their localised forms. openpyxl will only recognise the English names and only exactly as written here. These are as follows:

- 'Normal' # same as no style

Number formats

- 'Comma'
- 'Comma [0]'
- 'Currency'
- 'Currency [0]'
- 'Percent'

Informative

- 'Calculation'
- 'Total'
- 'Note'
- 'Warning Text'
- 'Explanatory Text'

Text styles

- 'Title'
- 'Headline 1'
- 'Headline 2'
- 'Headline 3'
- 'Headline 4'
- 'Hyperlink'
- 'Followed Hyperlink'
- 'Linked Cell'

Comparisons

- 'Input'
- 'Output'
- 'Check Cell'
- 'Good'
- 'Bad'
- 'Neutral'

Highlights

- 'Accent1'
- '20 % - Accent1'
- '40 % - Accent1'
- '60 % - Accent1'
- 'Accent2'
- '20 % - Accent2'
- '40 % - Accent2'
- '60 % - Accent2'
- 'Accent3'

- '20 % - Accent3'
- '40 % - Accent3'
- '60 % - Accent3'
- 'Accent4'
- '20 % - Accent4'
- '40 % - Accent4'
- '60 % - Accent4'
- 'Accent5'
- '20 % - Accent5'
- '40 % - Accent5'
- '60 % - Accent5'
- 'Accent6'
- '20 % - Accent6'
- '40 % - Accent6'
- '60 % - Accent6'
- 'Pandas'

For more information about the builtin styles please refer to the `openpyxl.styles.builtins`

7.7.2 Additional Worksheet Properties

These are advanced properties for particular behaviours, the most used ones are the "fitTopage" page setup property and the tabColor that define the background color of the worksheet tab.

Available properties for worksheets

- "enableFormatConditionsCalculation"
- "filterMode"
- "published"
- "syncHorizontal"
- "syncRef"
- "syncVertical"
- "transitionEvaluation"
- "transitionEntry"
- "tabColor"

Available fields for page setup properties

"autoPageBreaks" "fitToPage"

Available fields for outlines

- "applyStyles"
- "summaryBelow"
- "summaryRight"
- "showOutlineSymbols"

see http://msdn.microsoft.com/en-us/library/documentformat.openxml.spreadsheet.sheetproperties%28v=office.14%29.aspx_ for details.

Note: By default, outline properties are intitialized so you can directly modify each of their 4 attributes, while page setup properties don't. If you want modify the latter, you should first initialize a `openpyxl.worksheet.properties.PageSetupProperties` object with the required parameters. Once done, they can be directly modified by the routine later if needed.

```
>>> from openpyxl.workbook import Workbook
>>> from openpyxl.worksheet.properties import WorksheetProperties, PageSetupProperties
>>>
>>> wb = Workbook()
>>> ws = wb.active
>>>
>>> wsprops = ws.sheet_properties
>>> wsprops.tabColor = "1072BA"
>>> wsprops.filterMode = False
>>> wsprops.pageSetUpPr = PageSetupProperties(fitToPage=True, autoPageBreaks=False)
>>> wsprops.outlinePr.summaryBelow = False
>>> wsprops.outlinePr.applyStyles = True
>>> wsprops.pageSetUpPr.autoPageBreaks = True
```

7.8 Conditional Formatting

7.8.1 Conditional Formatting

Excel supports three different types of conditional formatting: builtins, standard and custom. Builtins combine specific rules with predefined styles. Standard conditional formats combine specific rules with custom formatting. In additional it is possible to define custom formulae for applying custom formats using differential styles.

Note: The syntax for the different rules varies so much that it is not possible for openpyxl to know whether a rule makes sense or not.

The basic syntax for creating a formatting rule is:

```
>>> from openpyxl.formatting import Rule
>>> from openpyxl.styles import Font, PatternFill, Border
>>> from openpyxl.styles.differential import DifferentialStyle
>>> dxf = DifferentialStyle(font=Font(bold=True), fill=PatternFill(start_color='EE1111
↪', end_color='EE1111'))
>>> rule = Rule(type='cellIs', dxf=dxf, formula=["10"])
```

Because the signatures for some rules can be quite verbose there are also some convenience factories for creating them.

Builtin formats

The builtins conditional formats are:

- ColorScale
- IconSet
- DataBar

Builtin formats contain a sequence of formatting settings which combine a type with an integer for comparison. Possible types are: *'num', 'percent', 'max', 'min', 'formula', 'percentile'*.

ColorScale

You can have color scales with 2 or 3 colors. 2 color scales produce a gradient from one color to another; 3 color scales use an additional color for 2 gradients.

The full syntax for creating a ColorScale rule is:

```
>>> from openpyxl.formatting.rule import ColorScale, FormatObject
>>> from openpyxl.styles import Color
>>> first = FormatObject(type='min')
>>> last = FormatObject(type='max')
>>> # colors match the format objects:
>>> colors = [Color('AA0000'), Color('00AA00')]
>>> cs2 = ColorScale(cfvo=[first, last], color=colors)
>>> # a three color scale would extend the sequences
>>> mid = FormatObject(type='num', val=40)
>>> colors.insert(1, Color('00AA00'))
>>> cs3 = ColorScale(cfvo=[first, mid, last], color=colors)
>>> # create a rule with the color scale
>>> from openpyxl.formatting.rule import Rule
>>> rule = Rule(type='colorScale', colorScale=cs3)
```

There is a convenience function for creating ColorScale rules

```
>>> from openpyxl.formatting.rule import ColorScaleRule
>>> rule = ColorScaleRule(start_type='percentile', start_value=10, start_color=
↪'FFAA0000',
...                       mid_type='percentile', mid_value=50, mid_color='FF0000AA',
...                       end_type='percentile', end_value=90, end_color='FF00AA00')
```

IconSet

Choose from the following set of icons: *'3Arrows', '3ArrowsGray', '3Flags', '3TrafficLights1', '3TrafficLights2', '3Signs', '3Symbols', '3Symbols2', '4Arrows', '4ArrowsGray', '4RedToBlack', '4Rating', '4TrafficLights', '5Arrows', '5ArrowsGray', '5Rating', '5Quarters'*

The full syntax for creating an IconSet rule is:

```
>>> from openpyxl.formatting.rule import IconSet, FormatObject
>>> first = FormatObject(type='percent', val=0)
>>> second = FormatObject(type='percent', val=33)
>>> third = FormatObject(type='percent', val=67)
>>> iconset = IconSet(iconSet='3TrafficLights1', cfvo=[first, second, third],␣
↪showValue=None, percent=None, reverse=None)
```

```
>>> # assign the icon set to a rule
>>> from openpyxl.formatting.rule import Rule
>>> rule = Rule(type='iconSet', iconSet=iconset)
```

There is a convenience function for creating IconSet rules:

```
>>> from openpyxl.formatting.rule import IconSetRule
>>> rule = IconSetRule('5Arrows', 'percent', [10, 20, 30, 40, 50], showValue=None,␣
↪percent=None, reverse=None)
```

DataBar

Currently, openpyxl supports the DataBars as defined in the original specification. Borders and directions were added in a later extension.

The full syntax for creating a DataBar rule is:

```
>>> from openpyxl.formatting.rule import DataBar, FormatObject
>>> first = FormatObject(type='min')
>>> second = FormatObject(type='max')
>>> data_bar = DataBar(cfvo=[first, second], color="638EC6", showValue=None,␣
↪minLength=None, maxLength=None)
>>> # assign the data bar to a rule
>>> from openpyxl.formatting.rule import Rule
>>> rule = Rule(type='dataBar', dataBar=data_bar)
```

There is a convenience function for creating DataBar rules:

```
>>> from openpyxl.formatting.rule import DataBarRule
>>> rule = DataBarRule(start_type='percentile', start_value=10, end_type='percentile',
↪ end_value='90',
...                    color="FF638EC6", showValue="None", minLength=None,␣
↪maxLength=None)
```

Standard conditional formats

The standard conditional formats are:

- Average
- Percent
- Unique or duplicate
- Value
- Rank

```
>>> from openpyxl import Workbook
>>> from openpyxl.styles import Color, PatternFill, Font, Border
>>> from openpyxl.styles.differential import DifferentialStyle
>>> from openpyxl.formatting.rule import ColorScaleRule, CellIsRule, FormulaRule
>>>
>>> wb = Workbook()
>>> ws = wb.active
>>>
>>> # Create fill
```

```
>>> redFill = PatternFill(start_color='EE1111',
...                end_color='EE1111',
...                fill_type='solid')
>>>
>>> # Add a two-color scale
>>> # Takes colors in excel 'RRGGBB' style.
>>> ws.conditional_formatting.add('A1:A10',
...             ColorScaleRule(start_type='min', start_color='AA0000',
...                            end_type='max', end_color='00AA00')
...                            )
>>>
>>> # Add a three-color scale
>>> ws.conditional_formatting.add('B1:B10',
...                ColorScaleRule(start_type='percentile', start_value=10, start_
↪color='AA0000',
...                               mid_type='percentile', mid_value=50, mid_color='0000AA
↪',
...                               end_type='percentile', end_value=90, end_color='00AA00
↪')
...                                 )
>>>
>>> # Add a conditional formatting based on a cell comparison
>>> # addCellIs(range_string, operator, formula, stopIfTrue, wb, font, border, fill)
>>> # Format if cell is less than 'formula'
>>> ws.conditional_formatting.add('C2:C10',
...             CellIsRule(operator='lessThan', formula=['C$1'], stopIfTrue=True,␣
↪fill=redFill))
>>>
>>> # Format if cell is between 'formula'
>>> ws.conditional_formatting.add('D2:D10',
...             CellIsRule(operator='between', formula=['1','5'], stopIfTrue=True,␣
↪fill=redFill))
>>>
>>> # Format using a formula
>>> ws.conditional_formatting.add('E1:E10',
...             FormulaRule(formula=['ISBLANK(E1)'], stopIfTrue=True, fill=redFill))
>>>
>>> # Aside from the 2-color and 3-color scales, format rules take fonts, borders and␣
↪fills for styling:
>>> myFont = Font()
>>> myBorder = Border()
>>> ws.conditional_formatting.add('E1:E10',
...             FormulaRule(formula=['E1=0'], font=myFont, border=myBorder,␣
↪fill=redFill))
>>>
>>> # Highlight cells that contain particular text by using a special formula
>>> red_text = Font(color="9C0006")
>>> red_fill = PatternFill(bgColor="FFC7CE")
>>> dxf = DifferentialStyle(font=red_text, fill=red_fill)
>>> rule = Rule(type="containsText", operator="containsText", text="highlight",␣
↪dxf=dxf)
>>> rule.formula = ['NOT(ISERROR(SEARCH("highlight",A1)))']
>>> ws.conditional_formatting.add('A1:F40', rule)
>>> wb.save("test.xlsx")
```

7.9 Print Settings

7.9.1 Print Settings

openpyxl provides reasonably full support for print settings.

Edit Print Options

```
>>> from openpyxl.workbook import Workbook
>>>
>>> wb = Workbook()
>>> ws = wb.active
>>>
>>> ws.print_options.horizontalCentered = True
>>> ws.print_options.verticalCentered = True
```

Headers and Footers

Headers and footers use their own formatting language. This is fully supported when writing them but, due to the complexity and the possibility of nesting, only partially when reading them. There is support for the font, size and color for a left, centre/center, or right element. Granular control (highlighting individuals words) will require applying control codes manually.

```
>>> from openpyxl.workbook import Workbook
>>>
>>> wb = Workbook()
>>> ws = wb.active
>>>
>>> ws.oddHeader.left.text = "Page &[Page] of &N"
>>> ws.oddHeader.left.size = 14
>>> ws.oddHeader.left.font = "Tahoma,Bold"
>>> ws.oddHeader.left.color = "CC3366"
```

Also supported are *evenHeader* and *evenFooter* as well as *firstHeader* and *firstFooter*.

Add Print Titles

You can print titles on every page to ensure that the data is properly labelled.

```
>>> from openpyxl.workbook import Workbook
>>>
>>> wb = Workbook()
>>> ws = wb.active
>>>
>>> ws.print_title_cols = 'A:B' # the first two cols
>>> ws.print_title_rows = '1:1' # the first row
```

Add a Print Area

You can select a part of a worksheet as the only part that you want to print

```
>>> from openpyxl.workbook import Workbook
>>>
>>> wb = Workbook()
>>> ws = wb.active
>>>
>>> ws.print_area = 'A1:F10'
```

7.10 Filtering and Sorting

7.10.1 Using filters and sorts

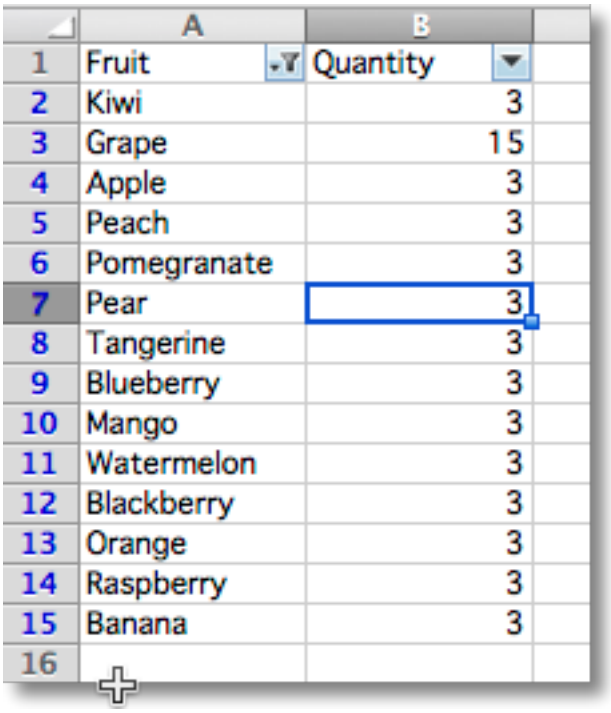
It's possible to add a filter to a worksheet.

Note: Filters and sorts can only be configured by openpyxl but will need to be applied in applications like Excel. This is because they actually rearranges or format cells or rows in the range.

To add a filter you define a range and then add columns and sort conditions:

```
from openpyxl import Workbook

wb = Workbook()
ws = wb.active

data = [
    ["Fruit", "Quantity"],
    ["Kiwi", 3],
    ["Grape", 15],
    ["Apple", 3],
    ["Peach", 3],
    ["Pomegranate", 3],
    ["Pear", 3],
    ["Tangerine", 3],
    ["Blueberry", 3],
    ["Mango", 3],
    ["Watermelon", 3],
    ["Blackberry", 3],
    ["Orange", 3],
    ["Raspberry", 3],
    ["Banana", 3]
]

for r in data:
    ws.append(r)

ws.auto_filter.ref = "A1:B15"
ws.auto_filter.add_filter_column(0, ["Kiwi", "Apple", "Mango"])
ws.auto_filter.add_sort_condition("B2:B15")

wb.save("filtered.xlsx")
```

This will add the relevant instructions to the file but will **neither actually filter nor sort**.



	A	B
1	Fruit	Quantity
2	Kiwi	3
3	Grape	15
4	Apple	3
5	Peach	3
6	Pomegranate	3
7	Pear	3
8	Tangerine	3
9	Blueberry	3
10	Mango	3
11	Watermelon	3
12	Blackberry	3
13	Orange	3
14	Raspberry	3
15	Banana	3
16		

7.11 Worksheet Tables

7.11.1 Worksheet Tables

Worksheet tables are references to groups of cells. This makes certain operations such as styling the cells in a table easier.

Creating a table

```
from openpyxl import Workbook
from openpyxl.worksheet.table import Table, TableStyleInfo

wb = Workbook()
ws = wb.active

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

# add column headings. NB. these must be strings
ws.append(["Fruit", "2011", "2012", "2013", "2014"])
for row in data:
    ws.append(row)

tab = Table(displayName="Table1", ref="A1:E5")
```

```
# Add a default style with striped rows and banded columns
style = TableStyleInfo(name="TableStyleMedium9", showFirstColumn=False,
                       showLastColumn=False, showRowStripes=True,␣
↪showColumnStripes=True)
tab.tableStyleInfo = style
ws.add_table(tab)
wb.save("table.xlsx")
```

By default tables are created with a header from the first row and filters for all the columns.

Styles are managed using the the *TableStyleInfo* object. This allows you to stripe rows or columns and apply the different colour schemes.

Important notes

Table names must be unique within a workbook and table headers and filter ranges must always contain strings. If this is not the case then Excel may consider the file invalid and remove the table.

7.12 Data Validation

7.12.1 Validating cells

You can add data validation to a workbook but currently cannot read existing data validation.

Examples

```
>>> from openpyxl import Workbook
>>> from openpyxl.worksheet.datavalidation import DataValidation
>>>
>>> # Create the workbook and worksheet we'll be working with
>>> wb = Workbook()
>>> ws = wb.active
>>>
>>> # Create a data-validation object with list validation
>>> dv = DataValidation(type="list", formula1='"Dog,Cat,Bat"', allow_blank=True)
>>>
>>> # Optionally set a custom error message
>>> dv.error ='Your entry is not in the list'
>>> dv.errorTitle = 'Invalid Entry'
>>>
>>> # Optionally set a custom prompt message
>>> dv.prompt = 'Please select from the list'
>>> dv.promptTitle = 'List Selection'
>>>
>>> # Add the data-validation object to the worksheet
>>> ws.add_data_validation(dv)
```

```
>>> # Create some cells, and add them to the data-validation object
>>> c1 = ws["A1"]
>>> c1.value = "Dog"
>>> dv.add(c1)
>>> c2 = ws["A2"]
```

```
>>> c2.value = "An invalid value"
>>> dv.add(c2)
>>>
>>> # Or, apply the validation to a range of cells
>>> dv.ranges.append('B1:B1048576')
>>>
>>> # Write the sheet out.  If you now open the sheet in Excel, you'll find that
>>> # the cells have data-validation applied.
>>> wb.save("test.xlsx")
```

Other validation examples

Any whole number:

```
dv = DataValidation(type="whole")
```

Any whole number above 100:

```
dv = DataValidation(type="whole",
                    operator="greaterThan",
                    formula1=100)
```

Any decimal number:

```
dv = DataValidation(type="decimal")
```

Any decimal number between 0 and 1:

```
dv = DataValidation(type="decimal",
                    operator="between",
                    formula1=0,
                    formula2=1)
```

Any date:

```
dv = DataValidation(type="date")
```

or time:

```
dv = DataValidation(type="time")
```

Any string at most 15 characters:

```
dv = DataValidation(type="textLength",
                    operator="lessThanOrEqual"),
                    formula1=15)
```

Cell range validation:

```
from openpyxl.utils import quote_sheetname
dv = DataValidation(type="list",
                    formula1="{0}!$B$1:$B$10".format(quote_sheetname(sheetname))
                    )
```

Custom rule:

```
dv = DataValidation(type="custom",
                    formula1"=SOMEFORMULA")
```

Note: See http://www.contextures.com/xlDataVal07.html for custom rules

7.13 Defined Names & Ranges

7.13.1 Defined Names

The specification has the following to say about defined names:

> "Defined names are descriptive text that is used to represents a cell, range of cells, formula, or constant value."

This means they are very loosely defined. They might contain a constant, a formula, a single cell reference, a range of cells or multiple ranges of cells across different worksheets. Or all of the above. They are defined globally for a workbook and accessed from there *defined_names* attribue.

Sample use for ranges

Accessing a range called "my_range":

```
my_range = wb.defined_names['my_range']
# if this contains a range of cells then the destinations attribute is not None
dests = my_range.destinations # returns a generator of (worksheet title, cell range)␣
↪tuples

cells = []
for title, coord in dests:
    ws = wb[title]
    cells.append(ws[coord])
```

7.14 Parsing Formulas

7.14.1 Parsing Formulas

openpyxl supports limited parsing of formulas embedded in cells. The *openpyxl.formula* package contains a *Tokenizer* class to break formulas into their consitutuent tokens. Usage is as follows:

```
>>> from openpyxl.formula import Tokenizer
>>> tok = Tokenizer("""=IF($A$1,"then True",MAX(DEFAULT_VAL,'Sheet 2'!B1))""")
>>> print("\n".join("%12s%11s%9s" % (t.value, t.type, t.subtype) for t in tok.items))
          IF(       FUNC      OPEN
         $A$1    OPERAND     RANGE
           ,         SEP       ARG
 "then True"    OPERAND      TEXT
           ,         SEP       ARG
        MAX(       FUNC      OPEN
 DEFAULT_VAL    OPERAND     RANGE
```

```
           ,        SEP       ARG
'Sheet 2'!B1    OPERAND     RANGE
           )       FUNC     CLOSE
           )       FUNC     CLOSE
```

As shown above, tokens have three attributes of interest:

- `.value`: The substring of the formula that produced this token
- `.type`: The type of token this represents. Can be one of
  - `Token.LITERAL`: If the cell does not contain a formula, its value is represented by a single `LITERAL` token.
  - `Token.OPERAND`: A generic term for any value in the Excel formula. (See `.subtype` below for more details).
  - `Token.FUNC`: Function calls are broken up into tokens for the opener (e.g., `SUM(`), followed by the arguments, followed by the closer (i.e., `)`). The function name and opening parenthesis together form one `FUNC` token, and the matching parenthesis forms another `FUNC` token.
  - `Token.ARRAY`: Array literals (enclosed between curly braces) get two `ARRAY` tokens each, one for the opening `{` and one for the closing `}`.
  - `Token.PAREN`: When used for grouping subexpressions (and not to denote function calls), parentheses are tokenized as `PAREN` tokens (one per character).
  - `Token.SEP`: These tokens are created from either commas (`,`) or semicolons (`;`). Commas create `SEP` tokens when they are used to separate function arguments (e.g., `SUM(a,b)`) or when they are used to separate array elements (e.g., `{a,b}`). (They have another use as an infix operator for joining ranges). Semicolons are always used to separate rows in an array literal, so always create `SEP` tokens.
  - `Token.OP_PRE`: Designates a prefix unary operator. Its value is always `+` or `-`
  - `Token.OP_IN`: Designates an infix binary operator. Possible values are `>=`, `<=`, `<>`, `=`, `>`, `<`, `*`, `/`, `+`, `-`, `^`, or `&`.
  - `Token.OP_POST`: Designates a postfix unary operator. Its value is always `%`.
  - `Token.WSPACE`: Created for any whitespace encountered. Its value is always a single space, regardless of how much whitespace is found.
- `.subtype`: Some of the token types above use the subtype to provide additional information about the token. Possible subtypes are:
  - `Token.TEXT`, `Token.NUMBER`, `Token.LOGICAL`, `Token.ERROR`, `Token.RANGE`: these subtypes describe the various forms of `OPERAND` found in formulae. `LOGICAL` is either `TRUE` or `FALSE`, `RANGE` is either a named range or a direct reference to another range. `TEXT`, `NUMBER`, and `ERROR` all refer to literal values in the formula
  - `Token.OPEN` and `Token.CLOSE`: these two subtypes are used by `PAREN`, `FUNC`, and `ARRAY`, to describe whether the token is opening a new subexpression or closing it.
  - `Token.ARG` and `Token.ROW`: are used by the `SEP` tokens, to distinguish between the comma and semicolon. Commas produce tokens of subtype `ARG` whereas semicolons produce tokens of subtype `ROW`

7.15 Protection

7.15.1 Protection

Warning: Password protecting a workbook or worksheet only provides a quite basic level of security. The data is not encrypted, so can be modified by any number of freely available tools. In fact the specification states: "Worksheet or workbook element protection should not be confused with file security. It is not meant to make your workbook safe from unintentional modification, and cannot protect it from malicious modification."

Openpyxl provides support for protecting a workbook and worksheet from modification. The Open XML "Legacy Password Hash Algorithm" is used to generate hashed password values unless another algorithm is explicitly configured.

Workbook Protection

To prevent other users from viewing hidden worksheets, adding, moving, deleting, or hiding worksheets, and renaming worksheets, you can protect the structure of your workbook with a password. The password can be set using the `openpyxl.workbook.protection.WorkbookProtection.workbookPassword()` property

```
>>> wb.security.workbookPassword = '...'
```

Similarly removing change tracking and change history from a shared workbook can be prevented by setting another password. This password can be set using the `openpyxl.workbook.protection.WorkbookProtection.revisionsPassword()` property

```
>>> wb.security.revisionsPassword = '...'
```

Other properties on the `openpyxl.workbook.protection.WorkbookProtection` object control exactly what restrictions are in place, but these will only be enforced if the appropriate password is set.

Specific setter functions are provided if you need to set the raw password value without using the default hashing algorithm - e.g.

```
hashed_password = ...
wb.security.set_workbook_password(hashed_password, already_hashed=True)
```

Worksheet Protection

Various aspects of a worksheet can also be locked by setting attributes on the `openpyxl.worksheet.protection.SheetProtection` object. Unlike the worksheet protection, sheet protection may be enabled with or without using a password. Sheet protection is enabled using the `openpxyl.worksheet.protection.SheetProtection.sheet` attribute

```
>>> ws = wb.active
>>> wb.protection.sheet = True
```

If no password is specified, users can disable configured sheet protection without specifying a password. Otherwise they must supply a password to change configured protections. The password is set using the `openpxyl.worksheet.protection.SheetProtection.password()` property

```
>>> ws = wb.active
>>> ws.protection.password = '...'
```

CHAPTER 8

Information for Developers

8.1 Development

With the ongoing development of openpyxl, there is occasional information useful to assist developers.

8.1.1 What is suppoprted

The primary aim of openpyxl is to support reading and writing Microsoft Excel 2010 files. Where possible support for files generated by other libraries or programs is available but this is not guaranteed.

8.1.2 Supporting different Python versions

We have a small library of utility functions to support development for Python 2 and 3. This is openpyxl.compat for Python and openpyxl.xml for XML functions.

8.1.3 Coding style

Use PEP-8 except when implementing attributes for roundtripping but always use Python data conventions (boolean, None, etc.) Note exceptions in docstrings.

8.1.4 Getting the source

The source code is hosted on bitbucket.org. You can get it using a Mercurial client and the following URL.

```
$ hg clone https://bitbucket.org/openpyxl/openpyxl
$ hg up 2.4
$ virtualenv openpyxl
$ cd openpyxl
$ source bin/activate
```

```
$ pip install -U -r requirements.txt
$ python setup.py develop
```

8.1.5 Specification

The file specification for OOXML was released jointly as ECMA 376 and ISO 29500.

8.1.6 Testing

Contributions without tests will **not** be accepted.

We use pytest as the test runner with pytest-cov for coverage information and pytest-flakes for static code analysis.

Coverage

The goal is 100 % coverage for unit tests - data types and utility functions. Coverage information can be obtained using

```
py.test --cov openpyxl
```

Organisation

Tests should be preferably at package / module level e.g openpyxl/cell. This makes testing and getting statistics for code under development easier:

```
py.test --cov openpyxl/cell openpyxl/cell
```

Checking XML

Use the `openpyxl.tests.helper.compare_xml` function to compare generated and expected fragments of XML.

Schema validation

When working on code to generate XML it is possible to validate that the generated XML conforms to the published specification. Note, this won't necessarily guarantee that everything is fine but is preferable to reverse engineering!

Microsoft Tools

Along with the SDK, Microsoft also has a "Productivity Tool" for working with Office OpenXML.

This allows you to quickly inspect or compare whole Excel files. Unfortunately, validation errors contain many false positives. The tool also contain links to the specification and implementers' notes.

Please see *Testing on Windows* for additional information on setting up and testing on Windows.

8.1.7 Contributing

Contributions in the form of pull requests are always welcome. Don't forget to add yourself to the list of authors!

8.1.8 Branch naming convention

We use a "major.minor.patch" numbering system, ie. 2.4.9. Development branches are named after "major.minor" releases. In general, API change will only happen major releases but there will be exceptions. Always communicate API changes to the mailing list before making them. If you are changing an API try and an implement a fallback (with deprecation warning) for the old behaviour.

The "default branch" is used for releases and always has changes from a development branch merged in. It should never be the target for a pull request.

8.1.9 Pull Requests

Pull requests should be submitted to the current, unreleased development branch. Eg. if the current release is 2.4.9, pull requests should be made to the 2.4 branch. Exceptions are bug fixes to released versions which should be made to the relevant release branch and merged upstream into development.

Please use tox to test code for different submissions **before** making a pull request. This is especially important for picking up problems across Python versions.

Documentation

Remember to update the documentation when adding or changing features. Check that documentation is syntactically correct.

```
tox -e doc
```

8.1.10 Benchmarking

Benchmarking and profiling are ongoing tasks. Contributions to these are very welcome as we know there is a lot to do.

Memory Use

There is a tox profile for long-running memory benchmarks using the *memory_utils* package.

```
tox -e memory
```

Pympler

As openpyxl does not include any internal memory benchmarking tools, the python *pympler* package was used during the testing of styles to profile the memory usage in `openpyxl.reader.excel.read_style_table()`:

```
# in openpyxl/reader/style.py
from pympler import muppy, summary

def read_style_table(xml_source):
  ...
  if cell_xfs is not None:  # ~ line 47
      initialState = summary.summarize(muppy.get_objects())  # Capture the initial␣
↪state
      for index, cell_xfs_node in enumerate(cell_xfs_nodes):
          ...
          table[index] = new_style
```

```
    finalState = summary.summarize(muppy.get_objects())  # Capture the final state
    diff = summary.get_diff(initialState, finalState)  # Compare
    summary.print_(diff)
```

`pympler.summary.print_()` prints to the console a report of object memory usage, allowing the comparison of different methods and examination of memory usage. A useful future development would be to construct a benchmarking package to measure the performance of different components.

8.2 Testing on Windows

Although openpyxl itself is pure Python and should run on any Python, we do use some libraries that require compiling for tests and documentation. The setup for testing on Windows is somewhat different.

8.2.1 Getting started

Once you have installed the versions of Python (2.6, 2.7, 3.3, 3.4) you should setup a development environment for testing so that you do not adversely affect the system install.

8.2.2 Setting up a development environment

First of all you should checkout a copy of the repository. Atlassian provides a nice GUI client SourceTree that allows you to do this with a single-click from the browser.

By default the repository will be installed under your user folder. eg. c:UsersYOURUSERopenpyxl

Switch to the branch you want to work on by double-clicking it. The default branch should never be used for development work.

Creating a virtual environment

You will need to manually install virtualenv. This is best done by first installing pip. open a command line and download the script "get_pip.py" to your preferred Python folder:

```
bitsadmin /transfer pip http://bootstrap.pypa.io/get-pip.py c:\python27\get-pip.py #␣
↪change the path as necessary
```

Install pip (it needs to be at least pip 6.0):

```
python get_pip.py
```

Now you can install virtualenv:

```
Scripts\pip install virtualenv
Scripts\virtualenv c:\Users\YOURUSER\openpyxl
```

8.2.3 lxml

openpyxl needs *lxml* in order to run the tests. Unfortunately, automatic installation of lxml on Windows is tricky as pip defaults to try and compile it. This can be avoided by using pre-compiled versions of the library.

1. In the command line switch to your repository folder:

```
cd c:\Users\YOURUSER\openpyxl
```

2. Activate the virtualenv:

```
Scripts\activate
```

3. Install a development version of openpyxl:

```
pip install -e .
```

4. Download all the relevant lxml Windows wheels

   Releases for legacy versions of Python:

   - lxml 3.5.0 for Python 2.6
   - lxml 3.5.0 for Python 3.3

5. Move all these files to a folder called "downloads" in your openpyxl checkout
6. Install the project requirements:

```
pip download -r requirements.txt -d downloads
pip install --no-index --find-links downloads -r requirements.txt
```

To run tests for the virtualenv:

```
py.test -xrf openpyxl # the flag will stop testing at the first error
```

8.2.4 tox

We use *tox* to run the tests on different Python versions and configurations. Using it is as simple as:

```
set PIP_FIND_LINKS=downloads
tox openpyxl
```

CHAPTER 9

API Documentation

9.1 openpyxl package

9.1.1 Subpackages

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

CHAPTER 11

Release Notes

11.1 2.4.9 (2017-10-19)

11.1.1 Bugfixes

- #809 Incomplete documentation of *copy_worksheet* method
- #811 Scoped definedNames not removed when worksheet is deleted
- #824 Raise an exception if a chart is used in multiple sheets
- #842 Non-ASCII table column headings cause an exception in Python 2
- #846 Conditional formats not supported in write-only mode
- #849 Conditional formats with no sqref cause an exception
- #859 Headers that start with a number conflict with font size
- #902 TableStyleElements don't always have a condtional format
- #908 Read-only mode sometimes returns too many cells

11.1.2 Pull requests

- #179 Cells kept in a set
- #180 Support for Workbook protection
- #182 Read support for page breaks
- #183 Improve documentation of *copy_worksheet* method
- #198 Fix for #908

11.2 2.4.8 (2017-05-30)

11.2.1 Bugfixes

- AutoFilter.sortState being assignd to the ws.sortState
- #766 Sheetnames with apostrophes need additional escaping
- #729 Cannot open files created by Microsoft Dynamics
- #819 Negative percents not case correctly
- #821 Runtime imports can cause deadlock
- #855 Print area containing only columns leads to corrupt file

11.2.2 Minor changes

- Preserve any table styles

11.3 2.4.7 (2017-04-24)

11.3.1 Bugfixes

- #807 Sample files being included by mistake in sdist

11.4 2.4.6 (2017-04-14)

11.4.1 Bugfixes

- #776 Cannot apply formatting to plot area
- #780 Exception when element attributes are Python keywords
- #781 Exception raised when saving files with styled columns
- #785 Number formats for data labels are incorrect
- #788 Worksheet titles not quoted in defined names
- #800 Font underlines not read correctly

11.5 2.4.5 (2017-03-07)

11.5.1 Bugfixes

- #750 Adding images keeps file handles open
- #772 Exception for column-only ranges
- #773 Cannot copy worksheets with non-ascii titles on Python 2

11.5.2 Pull requests

- 161 Support for non-standard names for Workbook part.
- 162 Documentation correction

11.6 2.4.4 (2017-02-23)

11.6.1 Bugfixes

- #673 Add close method to workbooks
- #762 openpyxl can create files with invalid style indices
- #729 Allow images in write-only mode
- #744 Rounded corners for charts
- #747 Use repr when handling non-convertible objects
- #764 Hashing function is incorrect
- #765 Named styles share underlying array

11.6.2 Minor Changes

- Add roundtrip support for worksheet tables.

11.6.3 Pull requests

- 160 Don't init mimetypes more than once.

11.7 2.4.3 (unreleased)

bad release

11.8 2.4.2 (2017-01-31)

11.8.1 Bug fixes

- #727 DeprecationWarning is incorrect
- #734 Exception raised if userName is missing
- #739 Always provide a date1904 attribute
- #740 Hashes should be stored as Base64
- #743 Print titles broken on sheetnames with spaces
- #748 Workbook breaks when active sheet is removed
- #754 Incorrect descriptor for Filter values

- #756 Potential XXE vulerability
- #758 Cannot create files with page breaks and charts
- #759 Problems with worksheets with commas in their titles

11.8.2 Minor Changes

- Add unicode support for sheet name incrementation.

11.9 2.4.1 (2016-11-23)

11.9.1 Bug fixes

- #643 Make checking for duplicate sheet titles case insensitive
- #647 Trouble handling LibreOffice files with named styles
- #687 Directly assigned new named styles always refer to "Normal"
- #690 Cannot parse print titles with multiple sheet names
- #691 Cannot work with macro files created by LibreOffice
- Prevent duplicate differential styles
- #694 Allow sheet titles longer than 31 characters
- #697 Cannot unset hyperlinks
- #699 Exception raised when format objects use cell references
- #703 Copy height and width when copying comments
- #705 Incorrect content type for VBA macros
- #707 IndexError raised in read-only mode when accessing individual cells
- #711 Files with external links become corrupted
- #715 Cannot read files containing macro sheets
- #717 Details from named styles not preserved when reading files
- #722 Remove broken Print Title and Print Area definitions

11.9.2 Minor changes

- Add support for Python 3.6
- Correct documentation for headers and footers

11.9.3 Deprecations

Worksheet methods *get_named_range()* and *get_sqaured_range()*

11.9.4 Bug fixes

11.10 2.4.0 (2016-09-15)

11.10.1 Bug fixes

- #652 Exception raised when epoch is 1904
- #642 Cannot handle unicode in headers and footers in Python 2
- #646 Cannot handle unicode sheetnames in Python 2
- #658 Chart styles, and axis units should not be 0
- #663 Strings in external workbooks not unicode

11.10.2 Major changes

- Add support for builtin styles and include one for Pandas

11.10.3 Minor changes

- Add a *keep_links* option to *load_workbook*. External links contain cached copies of the external workbooks. If these are big it can be advantageous to be able to disable them.
- Provide an example for using cell ranges in DataValidation.
- PR 138 - add copy support to comments.

11.11 2.4.0-b1 (2016-06-08)

11.11.1 Minor changes

- Add an the alias *hide_drop_down* to DataValidation for *showDropDown* because that is how Excel works.

11.11.2 Bug fixes

- #625 Exception raises when inspecting EmptyCells in read-only mode
- #547 Functions for handling OOXML "escaped" ST_XStrings
- #629 Row Dimensions not supported in write-only mode
- #530 Problems when removing worksheets with charts
- #630 Cannot use SheetProtection in write-only mode

11.11.3 Features

- Add write support for worksheet tables

11.12 2.4.0-a1 (2016-04-11)

11.12.1 Minor changes

- Remove deprecated methods from DataValidation
- Remove deprecated methods from PrintPageSetup
- Convert AutoFilter to Serialisable and extend support for filters
- Add support for SortState
- Removed *use_iterators* keyword when loading workbooks. Use *read_only* instead.
- Removed *optimized_write* keyword for new workbooks. Use *write_only* instead.
- Improve print title support
- Add print area support
- New implementation of defined names
- New implementation of page headers and footers
- Add support for Python's NaN
- Added iter_cols method for worksheets
- ws.rows and ws.columns now always return generators and start at the top of the worksheet
- Add a *values* property for worksheets
- Default column width changed to 8 as per the specification

11.12.2 Deprecations

- Cell anchor method
- Worksheet point_pos method
- Worksheet add_print_title method
- Worksheet HeaderFooter attribute, replaced by individual ones
- Flatten function for cells
- Workbook get_named_range, add_named_range, remove_named_range, get_sheet_names, get_sheet_by_name
- Comment text attribute
- Use of range strings deprecated for ws.iter_rows()
- Use of coordinates deprecated for ws.cell()
- Deprecate .copy() method for StyleProxy objects

11.12.3 Bug fixes

- #152 Hyperlinks lost when reading files
- #171 Add function for copying worksheets
- #386 Cells with inline strings considered empty

- #397 Add support for ranges of rows and columns
- #446 Workbook with definedNames corrupted by openpyxl
- #481 “safe” reserved ranges are not read from workbooks
- #501 Discarding named ranges can lead to corrupt files
- #574 Exception raised when using the class method to parse Relationships
- #579 Crashes when reading defined names with no content
- #597 Cannot read worksheets without coordinates
- #617 Customised named styles not correctly preserved

11.13 2.3.5 (2016-04-11)

11.13.1 Bug fixes

- #618 Comments not written in write-only mode

11.14 2.3.4 (2016-03-16)

11.14.1 Bug fixes

- #594 Content types might be missing when keeping VBA
- #599 Cells with only one cell look empty
- #607 Serialise NaN as ‘’

11.14.2 Minor changes

- Preserve the order of external references because formualae use numerical indices.
- Typo corrected in cell unit tests (PR 118)

11.15 2.3.3 (2016-01-18)

11.15.1 Bug fixes

- #540 Cannot read merged cells in read-only mode
- #565 Empty styled text blocks cannot be parsed
- #569 Issue warning rather than raise Exception raised for unparsable definedNames
- #575 Cannot open workbooks with embdedded OLE files
- #584 Exception when saving borders with attributes

11.15.2 Minor changes

- PR 103 Documentation about chart scaling and axis limits
- Raise an exception when trying to copy cells from other workbooks.

11.16 2.3.2 (2015-12-07)

11.16.1 Bug fixes

- #554 Cannot add comments to a worksheet when preserving VBA
- #561 Exception when reading phonetic text
- #562 DARKBLUE is the same as RED
- #563 Minimum for row and column indexes not enforced

11.16.2 Minor changes

- PR 97 One VML file per worksheet.
- PR 96 Correct descriptor for CharacterProperties.rtl
- #498 Metadata is not essential to use the package.

11.17 2.3.1 (2015-11-20)

11.17.1 Bug fixes

- #534 Exception when using columns property in read-only mode.
- #536 Incorrectly handle comments from Google Docs files.
- #539 Flexible value types for conditional formatting.
- #542 Missing content types for images.
- #543 Make sure images fit containers on all OSes.
- #544 Gracefully handle missing cell styles.
- #546 ExternalLink duplicated when editing a file with macros.
- #548 Exception with non-ASCII worksheet titles
- #551 Combine multiple LineCharts

11.17.2 Minor changes

- PR 88 Fix page margins in parser.

11.18 2.3.0 (2015-10-20)

11.18.1 Major changes

- Support the creation of chartsheets

11.18.2 Bug fixes

- #532 Problems when cells have no style in read-only mode.

11.18.3 Minor changes

- PR 79 Make PlotArea editable in charts
- Use graphicalProperties as the alias for spPr

11.19 2.3.0-b2 (2015-09-04)

11.19.1 Bug fixes

- #488 Support hashValue attribute for sheetProtection
- #493 Warn that unsupported extensions will be dropped
- #494 Cells with exponentials causes a ValueError
- #497 Scatter charts are broken
- #499 Inconsistent conversion of localised datetimes
- #500 Adding images leads to unreadable files
- #509 Improve handling of sheet names
- #515 Non-ascii titles have bad repr
- #516 Ignore unassigned worksheets

11.19.2 Minor changes

- Worksheets are now iterable by row.
- Assign individual cell styles only if they are explicitly set.

11.20 2.3.0-b1 (2015-06-29)

11.20.1 Major changes

- Shift to using (row, column) indexing for cells. Cells will at some point *lose* coordinates.
- New implementation of conditional formatting. Databars now partially preserved.

- et_xmlfile is now a standalone library.
- Complete rewrite of chart package
- Include a tokenizer for fomulae to be able to adjust cell references in them. PR 63

11.20.2 Minor changes

- Read-only and write-only worksheets renamed.
- Write-only workbooks support charts and images.
- PR76 Prevent comment images from conflicting with VBA

11.20.3 Bug fixes

- #81 Support stacked bar charts
- #88 Charts break hyperlinks
- #97 Pie and combination charts
- #99 Quote worksheet names in chart references
- #150 Support additional chart options
- #172 Support surface charts
- #381 Preserve named styles
- #470 Adding more than 10 worksheets with the same name leads to duplicates sheet names and an invalid file

11.21 2.2.6 (unreleased)

11.21.1 Bug fixes

- #502 Unexpected keyword “mergeCell”
- #503 tostring missing in dump_worksheet
- #506 Non-ASCII formulae cannot be parsed
- #508 Cannot save files with coloured tabs
- Regex for ignoring named ranges is wrong (character class instead of prefix)

11.22 2.2.5 (2015-06-29)

11.22.1 Bug fixes

- #463 Unexpected keyword “mergeCell”
- #484 Unusual dimensions breaks read-only mode
- #485 Move return out of loop

11.23 2.2.4 (2015-06-17)

11.23.1 Bug fixes

- #464 Cannot use images when preserving macros
- #465 ws.cell() returns an empty cell on read-only workbooks
- #467 Cannot edit a file with ActiveX components
- #471 Sheet properties elements must be in order
- #475 Do not redefine class __slots__ in subclasses
- #477 Write-only support for SheetProtection
- #478 Write-only support for DataValidation
- Improved regex when checking for datetime formats

11.24 2.2.3 (2015-05-26)

11.24.1 Bug fixes

- #451 fitToPage setting ignored
- #458 Trailing spaces lost when saving files.
- #459 setup.py install fails with Python 3
- #462 Vestigial rId conflicts when adding charts, images or comments
- #455 Enable Zip64 extensions for all versions of Python

11.25 2.2.2 (2015-04-28)

11.25.1 Bug fixes

- #447 Uppercase datetime number formats not recognised.
- #453 Borders broken in shared_styles.

11.26 2.2.1 (2015-03-31)

11.26.1 Minor changes

- PR54 Improved precision on times near midnight.
- PR55 Preserve macro buttons

11.26.2 Bug fixes

- #429 Workbook fails to load because header and footers cannot be parsed.
- #433 File-like object with encoding=None
- #434 SyntaxError when writing page breaks.
- #436 Read-only mode duplicates empty rows.
- #437 Cell.offset raises an exception
- #438 Cells with pivotButton and quotePrefix styles cannot be read
- #440 Error when customised versions of builtin formats
- #442 Exception raised when a fill element contains no children
- #444 Styles cannot be copied

11.27 2.2.0 (2015-03-11)

11.27.1 Bug fixes

- #415 Improved exception when passing in invalid in memory files.

11.28 2.2.0-b1 (2015-02-18)

11.28.1 Major changes

- Cell styles deprecated, use formatting objects (fonts, fills, borders, etc.) directly instead
- Charts will no longer try and calculate axes by default
- Support for template file types - PR21
- Moved ancillary functions and classes into utils package - single place of reference
- PR 34 Fully support page setup
- Removed SAX-based XML Generator. Special thanks to Elias Rabel for implementing xmlfile for xml.etree
- Preserve sheet view definitions in existing files (frozen panes, zoom, etc.)

11.28.2 Bug fixes

- #103 Set the zoom of a sheet
- #199 Hide gridlines
- #215 Preserve sheet view setings
- #262 Set the zoom of a sheet
- #392 Worksheet header not read
- #387 Cannot read files without styles.xml
- #410 Exception when preserving whitespace in strings

- #417 Cannot create print titles
- #420 Rename confusing constants
- #422 Preserve color index in a workbook if it differs from the standard

11.28.3 Minor changes

- Use a 2-way cache for column index lookups
- Clean up tests in cells
- PR 40 Support frozen panes and autofilter in write-only mode
- Use ws.calculate_dimension(force=True) in read-only mode for unsized worksheets

11.29 2.1.5 (2015-02-18)

11.29.1 Bug fixes

- #403 Cannot add comments in write-only mode
- #401 Creating cells in an empty row raises an exception
- #408 from_excel adjustment for Julian dates 1 < x < 60
- #409 refersTo is an optional attribute

11.29.2 Minor changes

- Allow cells to be appended to standard worksheets for code compatibility with write-only mode.

11.30 2.1.4 (2014-12-16)

11.30.1 Bug fixes

- #393 IterableWorksheet skips empty cells in rows
- #394 Date format is applied to all columns (while only first column contains dates)
- #395 temporary files not cleaned properly
- #396 Cannot write "=" in Excel file
- #398 Cannot write empty rows in write-only mode with LXML installed

11.30.2 Minor changes

- Add relation namespace to root element for compatibility with iWork
- Serialize comments relation in LXML-backend

11.31 2.1.3 (2014-12-09)

11.31.1 Minor changes

- PR 31 Correct tutorial
- PR 32 See #380
- PR 37 Bind worksheet to ColumnDimension objects

11.31.2 Bug fixes

- #379 ws.append() doesn't set RowDimension Correctly
- #380 empty cells formatted as datetimes raise exceptions

11.32 2.1.2 (2014-10-23)

11.32.1 Minor changes

- PR 30 Fix regex for positive exponentials
- PR 28 Fix for #328

11.32.2 Bug fixes

- #120, #168 defined names with formulae raise exceptions, #292
- #328 ValueError when reading cells with hyperlinks
- #369 IndexError when reading definedNames
- #372 number_format not consistently applied from styles

11.33 2.1.1 (2014-10-08)

11.33.1 Minor changes

- PR 20 Support different workbook code names
- Allow auto_axis keyword for ScatterCharts

11.33.2 Bug fixes

- #332 Fills lost in ConditionalFormatting
- #360 Support value="none" in attributes
- #363 Support undocumented value for textRotation
- #364 Preserve integers in read-only mode

- #366 Complete read support for DataValidation
- #367 Iterate over unsized worksheets

11.34 2.1.0 (2014-09-21)

11.34.1 Major changes

- "read_only" and "write_only" new flags for workbooks
- Support for reading and writing worksheet protection
- Support for reading hidden rows
- Cells now manage their styles directly
- ColumnDimension and RowDimension object manage their styles directly
- Use xmlfile for writing worksheets if available - around 3 times faster
- Datavalidation now part of the worksheet package

11.34.2 Minor changes

- Number formats are now just strings
- Strings can be used for RGB and aRGB colours for Fonts, Fills and Borders
- Create all style tags in a single pass
- Performance improvement when appending rows
- Cleaner conversion of Python to Excel values
- PR6 reserve formatting for empty rows
- standard worksheets can append from ranges and generators

11.34.3 Bug fixes

- #153 Cannot read visibility of sheets and rows
- #181 No content type for worksheets
- 241 Cannot read sheets with inline strings
- 322 1-indexing for merged cells
- 339 Correctly handle removal of cell protection
- 341 Cells with formulae do not round-trip
- 347 Read DataValidations
- 353 Support Defined Named Ranges to external workbooks

11.35 2.0.5 (2014-08-08)

11.35.1 Bug fixes

- #348 incorrect casting of boolean strings
- #349 roundtripping cells with formulae

11.36 2.0.4 (2014-06-25)

11.36.1 Minor changes

- Add a sample file illustrating colours

11.36.2 Bug fixes

- #331 DARKYELLOW was incorrect
- Correctly handle extend attribute for fonts

11.37 2.0.3 (2014-05-22)

11.37.1 Minor changes

- Updated docs

11.37.2 Bug fixes

- #319 Cannot load Workbooks with vertAlign styling for fonts

11.38 2.0.2 (2014-05-13)

11.39 2.0.1 (2014-05-13) brown bag

11.40 2.0.0 (2014-05-13) brown bag

11.40.1 Major changes

- This is last release that will support Python 3.2
- Cells are referenced with 1-indexing: A1 == cell(row=1, column=1)
- Use jdcal for more efficient and reliable conversion of datetimes
- Significant speed up when reading files
- Merged immutable styles

- Type inference is disabled by default
- RawCell renamed ReadOnlyCell
- ReadOnlyCell.internal_value and ReadOnlyCell.value now behave the same as Cell
- Provide no size information on unsized worksheets
- Lower memory footprint when reading files

11.40.2 Minor changes

- All tests converted to pytest
- Pyflakes used for static code analysis
- Sample code in the documentation is automatically run
- Support GradientFills
- BaseColWidth set

11.40.3 Pull requests

- #70 Add filterColumn, sortCondition support to AutoFilter
- #80 Reorder worksheets parts
- #82 Update API for conditional formatting
- #87 Add support for writing Protection styles, others
- #89 Better handling of content types when preserving macros

11.40.4 Bug fixes

- #46 ColumnDimension style error
- #86 reader.worksheet.fast_parse sets booleans to integers
- #98 Auto sizing column widths does not work
- #137 Workbooks with chartsheets
- #185 Invalid PageMargins
- #230 Using v in cells creates invalid files
- #243 - IndexError when loading workbook
- #263 - Forded conversion of line breaks
- #267 - Raise exceptions when passed invalid types
- #270 - Cannot open files which use non-standard sheet names or reference Ids
- #269 - Handling unsized worksheets in IterableWorksheet
- #270 - Handling Workbooks with non-standard references
- #275 - Handling auto filters where there are only custom filters
- #277 - Harmonise chart and cell coordinates

- #280- Explicit exception raising for invalid characters
- #286 - Optimized writer can not handle a datetime.time value
- #296 - Cell coordinates not consistent with documentation
- #300 - Missing column width causes load_workbook() exception
- #304 - Handling Workbooks with absolute paths for worksheets (from Sharepoint)

11.41 1.8.6 (2014-05-05)

11.41.1 Minor changes

Fixed typo for import Elementtree

11.41.2 Bugfixes

- #279 Incorrect path for comments files on Windows

11.42 1.8.5 (2014-03-25)

11.42.1 Minor changes

- The '=' string is no longer interpreted as a formula
- When a client writes empty xml tags for cells (e.g. <c r='A1'></c>), reader will not crash

11.43 1.8.4 (2014-02-25)

11.43.1 Bugfixes

- #260 better handling of undimensioned worksheets
- #268 non-ascii in formualae
- #282 correct implementation of register_namepsace for Python 2.6

11.44 1.8.3 (2014-02-09)

11.44.1 Major changes

Always parse using cElementTree

11.44.2 Minor changes

Slight improvements in memory use when parsing

- #256 - error when trying to read comments with optimised reader
- #260 - unsized worksheets
- #264 - only numeric cells can be dates

11.45 1.8.2 (2014-01-17)

- #247 - iterable worksheets open too many files
- #252 - improved handling of lxml
- #253 - better handling of unique sheetnames

11.46 1.8.1 (2014-01-14)

- #246

11.47 1.8.0 (2014-01-08)

11.47.1 Compatibility

Support for Python 2.5 dropped.

11.47.2 Major changes

- Support conditional formatting
- Support lxml as backend
- Support reading and writing comments
- pytest as testrunner now required
- Improvements in charts: new types, more reliable

11.47.3 Minor changes

- load_workbook now accepts data_only to allow extracting values only from formulae. Default is false.
- Images can now be anchored to cells
- Docs updated
- Provisional benchmarking
- Added convenience methods for accessing worksheets and cells by key

11.48 1.7.0 (2013-10-31)

11.48.1 Major changes

Drops support for Python < 2.5 and last version to support Python 2.5

11.48.2 Compatibility

Tests run on Python 2.5, 2.6, 2.7, 3.2, 3.3

11.48.3 Merged pull requests

- 27 Include more metadata
- 41 Able to read files with chart sheets
- 45 Configurable Worksheet classes
- 3 Correct serialisation of Decimal
- 36 Preserve VBA macros when reading files
- 44 Handle empty oddheader and oddFooter tags
- 43 Fixed issue that the reader never set the active sheet
- 33 Reader set value and type explicitly and TYPE_ERROR checking
- 22 added page breaks, fixed formula serialization
- 39 Fix Python 2.6 compatibility
- 47 Improvements in styling

11.48.4 Known bugfixes

- #109
- #165
- #179
- #209
- #112
- #166
- #109
- #223
- #124
- #157

11.48.5 Miscellaneous

Performance improvements in optimised writer

Docs updated

Index

O