xlwings - Make Excel Fly!

Release 0.7.1

Zoomer Analytics LLC

Oct 06, 2018

Contents

CHAPTER 1

Installation

The easiest way to install xlwings is via pip:

```
pip install xlwings
```

or conda:

```
conda install xlwings
```

Alternatively, it can be installed from source. From within the `xlwings` directory, execute:

```
python setup.py install
```

Note: When you are using Mac Excel 2016 and are installing xlwings with `conda` (or use the version that comes with Anaconda), you'll need to run `$ xlwings runpython install` once to enable the `RunPython` calls from VBA. Alternatively, you can simply install xlwings with `pip`.

1.1 Dependencies

- **Windows**: `pywin32`, `comtypes`

  On Windows, it is recommended to use one of the scientific Python distributions like Anaconda, WinPython or Canopy as they already include pywin32. Otherwise it needs to be installed from here.

- **Mac**: `psutil`, `appscript`

  On Mac, the dependencies are automatically being handled if xlwings is installed with `pip`. However, the Xcode command line tools need to be available. Mac OS X 10.4 (*Tiger*) or later is required. The

recommended Python distribution for Mac is Anaconda.

1.2 Optional Dependencies

- NumPy
- Pandas
- Matplotlib
- Pillow/PIL

These packages are not required but highly recommended as they play very nicely with xlwings.

1.3 Python version support

xlwings is tested on Python 2.6-2.7 and 3.3+

CHAPTER 2

Quickstart

This guide assumes you have xlwings already installed. If that's not the case, head over to *Installation*.

2.1 Interact with Excel from Python

Writing/reading values to/from Excel and adding a chart is as easy as:

```
>>> import xlwings as xw
>>> wb = xw.Workbook()  # Creates a connection with a new workbook
>>> xw.Range('A1').value = 'Foo 1'
>>> xw.Range('A1').value
'Foo 1'
>>> xw.Range('A1').value = [['Foo 1', 'Foo 2', 'Foo 3'], [10.0, 20.0, 30.0]]
>>> xw.Range('A1').table.value  # or: Range('A1:C2').value
[['Foo 1', 'Foo 2', 'Foo 3'], [10.0, 20.0, 30.0]]
>>> xw.Sheet(1).name
'Sheet1'
>>> chart = xw.Chart.add(source_data=xw.Range('A1').table)
```

The Range and Chart objects as used above will refer to the active sheet of the current Workbook `wb`. Include the Sheet name like this:

```
xw.Range('Sheet1', 'A1:C3').value
xw.Range(1, (1,1), (3,3)).value  # index notation
xw.Chart.add('Sheet1', source_data=xw.Range('Sheet1', 'A1').table)
```

Qualify the Workbook additionally like this:

```
xw.Range('Sheet1', 'A1', wkb=wb).value
xw.Chart.add('Sheet1', wkb=wb, source_data=xw.Range('Sheet1', 'A1', wkb=wb).
↪table)
xw.Sheet(1, wkb=wb).name
```

or simply set the current workbook first:

```
wb.set_current()
xw.Range('Sheet1', 'A1').value
xw.Chart.add('Sheet1', source_data=xw.Range('Sheet1', 'A1').table)
xw.Sheet(1).name
```

These commands also work seamlessly with **NumPy arrays** and **Pandas DataFrames**, see *Data Structures Tutorial* for details.

Matplotlib figures can be shown as pictures in Excel:

```
import matplotlib.pyplot as plt
fig = plt.figure()
plt.plot([1, 2, 3, 4, 5])

plot = xw.Plot(fig)
plot.show('Plot1')
```

2.2 Call Python from Excel

If, for example, you want to fill your spreadsheet with standard normally distributed random numbers, your VBA code is just one line:

```
Sub RandomNumbers()
    RunPython ("import mymodule; mymodule.rand_numbers()")
End Sub
```

This essentially hands over control to `mymodule.py`:

```
import numpy as np
from xlwings import Workbook, Range

def rand_numbers():
    """ produces standard normally distributed random numbers with shape (n,n)
↪"""
    wb = Workbook.caller()  # Creates a reference to the calling Excel file
    n = int(Range('Sheet1', 'B1').value)  # Write desired dimensions into␣
↪Cell B1
    rand_num = np.random.randn(n, n)
    Range('Sheet1', 'C3').value = rand_num
```

To make this run, just import the VBA module `xlwings.bas` in the VBA editor (Open the VBA editor with `Alt-F11`, then go to `File > Import File...` and import the `xlwings.bas` file.). It can be found in the directory of your `xlwings` installation.

Note: Always instantiate the `Workbook` within the function that is called from Excel and not outside as global variable.

For further details, see *VBA: Calling Python from Excel*.

2.3 User Defined Functions (UDFs) - Currently Windows only

Writing a UDF in Python is as easy as:

```
import xlwings as xw

@xw.func
def double_sum(x, y):
    """Returns twice the sum of the two arguments"""
    return 2 * (x + y)
```

This then needs to be imported into Excel: For further details, see *UDF Tutorial*.

2.4 Easy deployment

Deployment is really the part where xlwings shines:

- Just zip-up your Spreadsheet with your Python code and send it around. The receiver only needs to have an installation of Python with xlwings (and obviously all the other packages you're using).
- There is no need to install any Excel add-in.

CHAPTER 3

Connect to Workbooks

First things first: to be able to talk to Excel from Python or call `RunPython` in VBA, you need to establish a connection with an Excel Workbook.

3.1 Python to Excel

There are various ways to connect to an Excel workbook from Python:

- `wb = Workbook()` connects to a a new workbook
- `wb = Workbook.active()` connects to the active workbook (supports multiple Excel instances)
- `wb = Workbook('Book1')` connects to an unsaved workbook
- `wb = Workbook('MyWorkbook.xlsx')` connects to a saved (open) workbook by name (incl. xlsx etc.)
- `wb = Workbook(r'C:\path\to\file.xlsx')` connects to a saved (open or closed) workbook by path

Note: When specifying file paths on Windows, you should either use raw strings by putting an `r` in front of the string or use double back-slashes like so: `C:\\path\\to\\file.xlsx`.

3.2 Excel to Python (RunPython)

To make a connection from Excel, i.e. when calling a Python script with `RunPython`, use `Workbook.caller()`, see *Call Python with "RunPython"*. Check out the section about *Debugging* to see how you can

call a script from both sides, Python and Excel, without the need to constantly change between `Workbook.caller()` and one of the methods explained above.

3.3 User Defined Functions (UDFs)

UDFs work differently and don't need the explicit instantiation of a `Workbook`, see *UDF Tutorial*. However, `xw.Workbook.caller()` can be used in UDFs although just read-only.

CHAPTER 4

Data Structures Tutorial

This page gives you a quick introduction to the most common use cases and default behaviour of xlwings when reading and writing values. For an in-depth documentation of how to control things using the `options` method, have a look at *Converters and Options*.

4.1 Single Cells

Single cells are by default returned either as `float`, `unicode`, `None` or `datetime` objects, depending on whether the cell contains a number, a string, is empty or represents a date:

```
>>> from xlwings import Workbook, Range
>>> from datetime import datetime
>>> wb = Workbook()
>>> Range('A1').value = 1
>>> Range('A1').value
1.0
>>> Range('A2').value = 'Hello'
>>> Range('A2').value
'Hello'
>>> Range('A3').value is None
True
>>> Range('A4').value = datetime(2000, 1, 1)
>>> Range('A4').value
datetime.datetime(2000, 1, 1, 0, 0)
```

4.2 Lists

- 1d lists: Ranges that represent rows or columns in Excel are returned as simple lists, which means that once they are in Python, you've lost the information about the orientation. If that is an issue, read under the next point how this information can be preserved:

```
>>> wb = Workbook()
>>> Range('A1').value = [[1],[2],[3],[4],[5]]  # Column orientation␣
↪(nested list)
>>> Range('A1:A5').value
[1.0, 2.0, 3.0, 4.0, 5.0]
>>> Range('A1').value = [1, 2, 3, 4, 5]
>>> Range('A1:E1').value
[1.0, 2.0, 3.0, 4.0, 5.0]
```

To force a single cell to arrive as list, use:

```
>>> Range('A1').options(ndim=1).value
[1.0]
```

Note: To write a list in column orientation to Excel, use `transpose`: `Range('A1').options(transpose=True).value = [1,2,3,4]`

- 2d lists: If the row or column orientation has to be preserved, set `ndim` in the Range options. This will return the Ranges as nested lists ("2d lists"):

```
>>> Range('A1:A5').options(ndim=2).value
[[1.0], [2.0], [3.0], [4.0], [5.0]]
>>> Range('A1:E1').options(ndim=2).value
[[1.0, 2.0, 3.0, 4.0, 5.0]]
```

- 2 dimensional Ranges are automatically returned as nested lists. When assigning (nested) lists to a Range in Excel, it's enough to just specify the top left cell as target address. This sample also makes use of index notation to read the values back into Python:

```
>>> Range('A10').value = [['Foo 1', 'Foo 2', 'Foo 3'], [10, 20, 30]]
>>> Range((10,1),(11,3)).value
[['Foo 1', 'Foo 2', 'Foo 3'], [10.0, 20.0, 30.0]]
```

Note: Try to minimize the number of interactions with Excel. It is always more efficient to do `Range('A1').value = [[1,2],[3,4]]` than `Range('A1').value = [1, 2]` and `Range('A2').value = [3, 4]`.

4.3 Range expanding: "table", "vertical" and "horizontal"

You can get the dimensions of Excel Ranges dynamically through either the Range properties `table`, `vertical` and `horizontal` or through `options(expand='table')` (same for `'vertical'` and `'horizontal'`). While properties give back a changed Range object, options are only evaluated when accessing the values of a Range. The difference is best explained with an example:

```
>>> wb = Workbook()
>>> Range('A1').value = [[1,2], [3,4]]
>>> rng1 = Range('A1').table
>>> rng2 = Range('A1').options(expand='table')
>>> rng1.value
[[1.0, 2.0], [3.0, 4.0]]
>>> rng2.value
[[1.0, 2.0], [3.0, 4.0]]
>>> Range('A3').value = [5, 6]
>>> rng1.value
[[1.0, 2.0], [3.0, 4.0]]
>>> rng2.value
[[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]]
```

Note: Using `table` together with a named Range as top left cell gives you a flexible setup in Excel: You can move around the table and change it's size without having to adjust your code, e.g. by using something like `Range('NamedRange').table.value`.

4.4 NumPy arrays

NumPy arrays work similar to nested lists. However, empty cells are represented by `nan` instead of `None`. If you want to read in a Range as array, set `convert=np.array`:

```
>>> import numpy as np
>>> wb = Workbook()
>>> Range('A1').value = np.eye(3)
>>> Range('A1').options(np.array, expand='table').value
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

4.5 Pandas DataFrames

```
>>> wb = Workbook()
>>> df = pd.DataFrame([[1.1, 2.2], [3.3, None]], columns=['one', 'two'])
>>> df
   one  two
```

(continues on next page)

(continued from previous page)

```
0  1.1  2.2
1  3.3  NaN
>>> Range('A1').value = df
>>> Range('A1:C3').options(pd.DataFrame).value
   one  two
0  1.1  2.2
1  3.3  NaN
# options: work for reading and writing
>>> Range('A5').options(index=False).value = df
>>> Range('A9').options(index=False, header=False).value = df
```

4.6 Pandas Series

```
>>> import pandas as pd
>>> import numpy as np
>>> wb = Workbook()
>>> s = pd.Series([1.1, 3.3, 5., np.nan, 6., 8.], name='myseries')
>>> s
0    1.1
1    3.3
2    5.0
3    NaN
4    6.0
5    8.0
Name: myseries, dtype: float64
>>> Range('A1').value = s
>>> Range('A1:B7').options(pd.Series).value
0    1.1
1    3.3
2    5.0
3    NaN
4    6.0
5    8.0
Name: myseries, dtype: float64
```

Note: You only need to specify the top left cell when writing a list, an NumPy array or a Pandas DataFrame to Excel, e.g.: `Range('A1').value = np.eye(10)`

CHAPTER 5

VBA: Calling Python from Excel

5.1 xlwings VBA module

To get access to the `RunPython` function and/or to be able to run User Defined Functions (UDFs), you need to have the xlwings VBA module available in your Excel workbook.

For new projects, by far the easiest way to get started is by using the command line client with the quickstart option, see *Command Line Client* for details:

```
$ xlwings quickstart myproject
```

This will create a new folder in your current directory with a fully prepared Excel file and an empty Python file.

Alternatively, you can also open a new spreadsheet from a template (`$ xlwings template open`) or manually insert the module in an existing workbook like so:

- Open the VBA editor with `Alt-F11`
- Then go to `File > Import File...` and import the `xlwings.bas` file. It can be found in the directory of your `xlwings` installation.

If you don't know the location of your xlwings installation, you can find it as follows:

```
$ python
>>> import xlwings
>>> xlwings.__path__
```

5.2 Settings

While the defaults will often work out-of-the box, you can change the settings at the top of the xlwings VBA module under `Function Settings`:

```
PYTHON_WIN = ""
PYTHON_MAC = ""
PYTHON_FROZEN = ThisWorkbook.Path & "\build\exe.win32-2.7"
PYTHONPATH = ThisWorkbook.Path
UDF_MODULES = ""
UDF_DEBUG_SERVER = False
LOG_FILE = ThisWorkbook.Path & "\xlwings_log.txt"
SHOW_LOG = True
OPTIMIZED_CONNECTION = False
```

- `PYTHON_WIN`: This is the full path of the Python interpreter on Windows, , e.g. `"C:\Python35\pythonw.exe"`. `""` resolves to your default Python installation on the PATH, i.e. the one you can start by just typing `python` at a command prompt.
- `PYTHON_MAC`: This is the full path of the Python interpreter on Mac OSX, e.g. `"/usr/local/bin/python3.5"`. `""` resolves to your default installation as per PATH on .bash_profile. To get special folders on Mac, type `GetMacDir("Name")` where `Name` is one of the following: `Home`, `Desktop`, `Applications`, `Documents`.
- `PYTHON_FROZEN` [Optional]: Currently only on Windows, indicates the directory of the exe file that has been frozen by either using `cx_Freeze` or `py2exe`. Can be set to `""` if unused.
- `PYTHONPATH` [Optional]: If the source file of your code is not found, add the path here. Otherwise set it to `""`.
- `UDF_MODULES` [Optional, Windows only]: Names of Python modules (without .py extension) from which the UDFs are being imported. Separate multiple modules by ";". Example: `UDF_PATH = "common_udfs;myproject"` Default: `UDF_PATH = ""` defaults to a file in the same directory of the Excel spreadsheet with the same name but ending in `.py`.
- `UDF_DEBUG_SERVER`: Set this to True if you want to run the xlwings COM server manually for debugging, see *Debugging*.
- `LOG_FILE` [Optional]: Leave empty for default location (see below) or provide directory including file name.
- `SHOW_LOG`: If False, no pop-up with the Log messages (usually errors) will be shown. Use with care.
- `OPTIMIZED_CONNECTION`: Currently only on Windows, use a COM Server for an efficient connection (experimental!)

5.3 LOG_FILE default locations

- Windows: `%APPDATA%\xlwings_log.txt`
- Mac with Excel 2011: `/tmp/xlwings_log.txt`

- Mac with Excel 2016: `~/Library/Containers/com.microsoft.Excel/Data/xlwings_log.txt`

Note: If the settings (especially `PYTHONPATH` and `LOG_FILE`) need to work on Windows on Mac, use backslashes in relative file path, i.e. `ThisWorkbook.Path & "\mydirectory"`.

5.4 Call Python with "RunPython"

After your workbook contains the xlwings VBA module with potentially adjusted Settings, go to `Insert > Module` (still in the VBA-Editor). This will create a new Excel module where you can write your Python call as follows (note that the `quickstart` or `template` commands already add an empty Module1, so you don't need to insert a new module manually):

```
Sub MyMacro()
    RunPython ("import mymodule; mymodule.rand_numbers()")
End Sub
```

This essentially hands over control to `mymodule.py`:

```
import numpy as np
from xlwings import Workbook, Range

def rand_numbers():
    """ produces std. normally distributed random numbers with shape (n,n)"""
    wb = Workbook.caller()  # Creates a reference to the calling Excel file
    n = int(Range('Sheet1', 'B1').value)  # Write desired dimensions into␣
↪Cell B1
    rand_num = np.random.randn(n, n)
    Range('Sheet1', 'C3').value = rand_num
```

You can then attach `MyMacro` to a button or run it directly in the VBA Editor by hitting `F5`.

Note: Always place `Workbook.caller()` within the function that is being called from Excel and not outside as global variable. Otherwise it prevents Excel from shutting down properly upon exiting and leaves you with a zombie process when you use `OPTIMIZED_CONNECTION = True`.

5.5 Function Arguments and Return Values

While it's technically possible to include arguments in the function call within `RunPython`, it's not very convenient. To do that easily and to also be able to return values from Python, use UDFs, see *UDF Tutorial* - however, this is currently limited to Windows only.

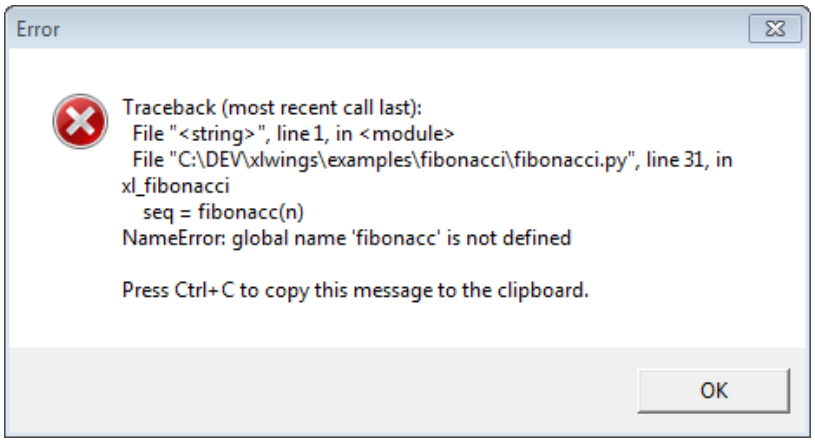
CHAPTER 6

Debugging

Since xlwings runs in every Python environment, you can use your preferred way of debugging.

- **RunPython**: When calling Python through `RunPython`, you can set a `mock_caller` to make it easy to switch back and forth between calling things from Excel and Python.
- **UDFs**: For debugging User Defined Functions, xlwings offers a convenient debugging server

To begin with, Excel will show Python errors in a Message Box:



Note: On Mac, if the `import` of a module/package fails before xlwings is imported, the popup will not be shown and the StatusBar will not be reset. However, the error will still be logged in the log file. For the location of the logfile, see *LOG_FILE default locations*.

6.1 RunPython

Consider the following code structure of your Python source code:

```
import os
from xlwings import Workbook, Range

def my_macro():
    wb = Workbook.caller()
    Range('A1').value = 1

if __name__ == '__main__':
    # Expects the Excel file next to this source file, adjust accordingly.
    path = os.path.abspath(os.path.join(os.path.dirname(__file__), 'myfile.
↪xlsm'))
    Workbook.set_mock_caller(path)
    my_macro()
```

`my_macro()` can now easily be run from Python for debugging and from Excel via `RunPython` without having to change the source code:

```
Sub my_macro()
    RunPython ("import my_module; my_module.my_macro()")
End Sub
```

6.2 UDF debug server

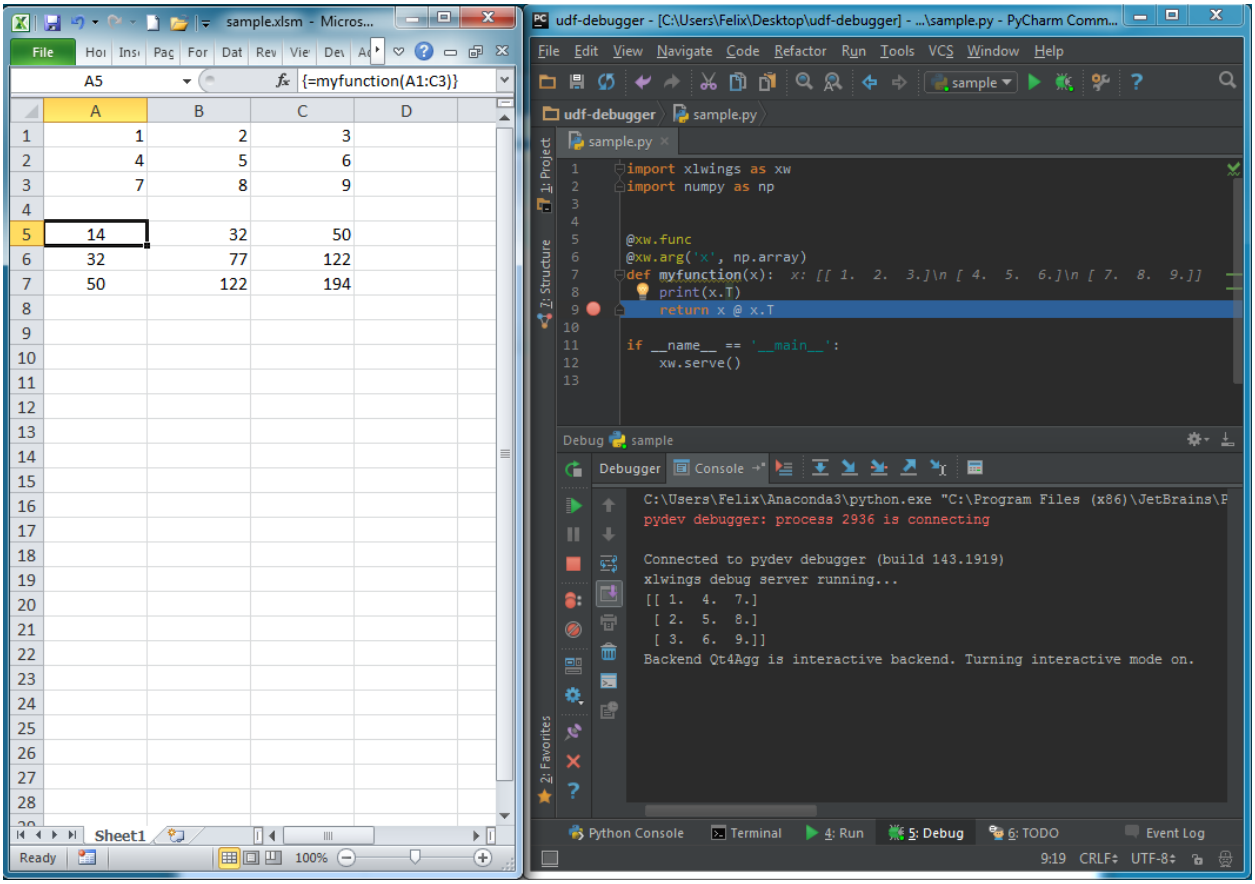
Windows only: To debug UDFs, just set `UDF_DEBUG_SERVER = True` in the VBA Settings, at the top of the xlwings VBA module. Then add the following lines at the end of your Python source file and run it. Depending on which IDE you use, you might want to run things in "debug" mode (e.g. in case your using PyCharm or PyDev):

```
if __name__ == '__main__':
    xw.serve()
```

When you recalculate the Sheet (`Ctrl-Alt-F9`), the code will stop at breakpoints or output any print calls that you may have.

The following screenshot shows the code stopped at a breakpoint in the community version of PyCharm:

Note: When running the debug server from a command prompt, there is currently no gracious way to terminate it, but closing the command prompt will kill it.


sample.xlsm - Micros...
File
A5
{=myfunction(A1:C3)}
Sheet1
Ready
100%
udf-debugger - [C:\Users\Felix\Desktop\udf-debugger] - ...\sample.py - PyCharm Comm...
File Edit View Navigate Code Refactor Run Tools VCS Window Help
udf-debugger
sample.py
import xlwings as xw
import numpy as np
@xw.func
@xw.arg('x', np.array)
def myfunction(x):
print(x.T)
return x @ x.T
if __name__ == '__main__':
xw.serve()
Debug sample
Debugger
Console
C:\Users\Felix\Anaconda3\python.exe "C:\Program Files (x86)\JetBrains\P
pydev debugger: process 2936 is connecting
Connected to pydev debugger (build 143.1919)
xlwings debug server running...
[[1. 4. 7.]
[2. 5. 8.]
[3. 6. 9.]]
Backend Qt4Agg is interactive backend. Turning interactive mode on.
Python Console
Terminal
4: Run
5: Debug
6: TODO
Event Log
9:19
CRLF
UTF-8

CHAPTER 7

Matplotlib

xlwings.Plot() allows for an easy integration of Matplotlib with Excel. The plot is pasted into Excel as picture.

7.1 Getting started

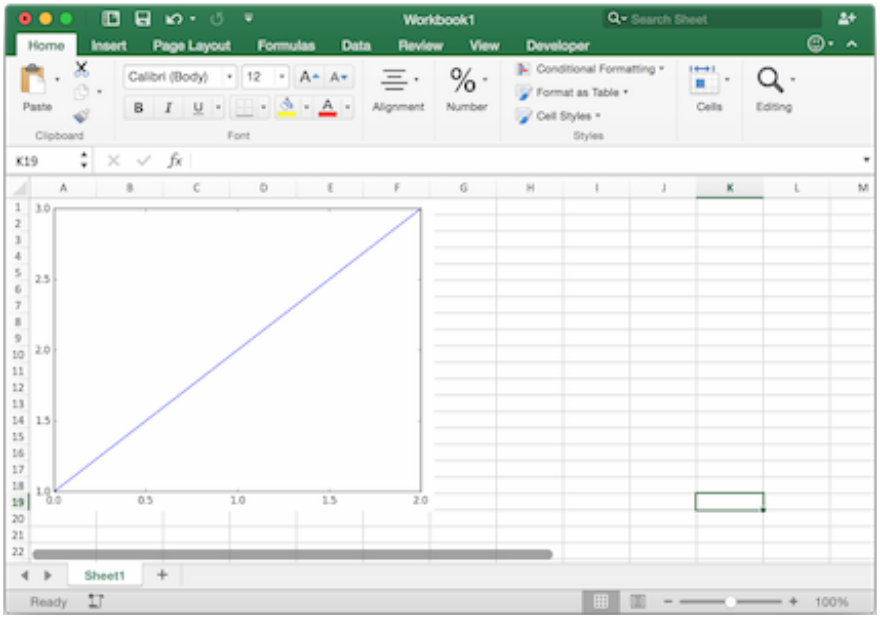
The easiest sample boils down to:

```
>>> import matplotlib.pyplot as plt
>>> import xlwings as xw

>>> fig = plt.figure()
>>> plt.plot([1, 2, 3])

>>> wb = xw.Workbook()
>>> xw.Plot(fig).show('MyPlot')
```

Note: You can now resize and position the plot on Excel: subsequent calls to `show` with the same name (`'MyPlot'`) will update the picture without changing its position or size.

7.2 Full integration with Excel

Calling the above code with *RunPython* and binding it e.g. to a button is straightforward and works cross-platform.



However, on Windows you can make things feel even more integrated by setting up a *UDF* along the following lines:

```
@xw.func
def myplot(n):
    wb = xw.Workbook.caller()
    fig = plt.figure()
    plt.plot(range(int(n)))
    xw.Plot(fig).show('MyPlot')
    return 'Plotted with n={}'.format(n)
```

If you import this function and call it from cell B2, then the plot gets automatically updated when cell B1 changes:

7.3 Properties

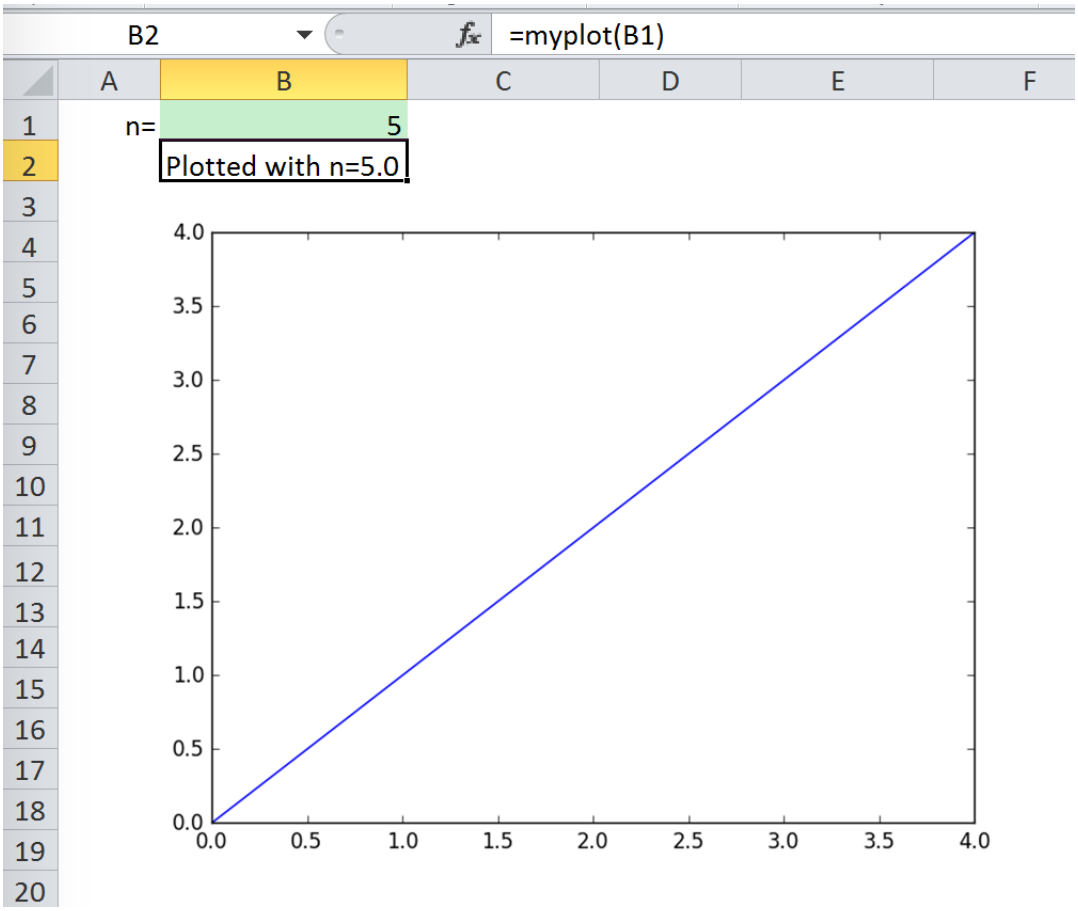
Size, position and other properties can either be set as arguments within `show`, see *`xlwings.Plot.show()`*, or by manipulating the picture object as returned by `show`, see *`xlwings.Picture()`*.

For example:

```
>>> xw.Plot(fig).show('MyPlot', left=xw.Range('B5').left, top=xw.Range('B5').
↪top)
```

or:

```
>>> plot = xw.Plot(fig).show('MyPlot')
>>> plot.height /= 2
>>> plot.width /= 2
```


B2
=myplot(B1)
A
B
C
D
E
F
n=
5
Plotted with n=5.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

Note: Once the picture is shown in Excel, you can only change it's properties via the picture object and not within the `show` method.

7.4 Getting a matplotlib figure

Here are a few examples of how you get a matplotlib `figure` object:

- via PyPlot interface:

```
import matplotlib.pyplot as plt
fig = plt.figure()
plt.plot([1, 2, 3, 4, 5])
```

  or:

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4, 5])
fig = plt.gcf()
```

- via object oriented interface:

```
from matplotlib.figure import Figure
fig = Figure(figsize=(8, 6))
ax = fig.add_subplot(111)
ax.plot([1, 2, 3, 4, 5])
```

- via Pandas:

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])
ax = df.plot(kind='bar')
fig = ax.get_figure()
```

Then show it in Excel as picture as seen above:

```
plot = Plot(fig)
plot.show('Plot1')
```

CHAPTER 8

UDF Tutorial

Note: UDFs are currently only available on Windows.

This tutorial gets you quickly started on how to write User Defined Functions. For details of how to control the behaviour of the arguments and return values, have a look at *Converters and Options*.

8.1 One-time Excel preparations

Required: Enable `Trust access to the VBA project object model` under `File > Options > Trust Center > Trust Center Settings > Macro Settings`

Recommended: Install the add-in via command prompt: `xlwings addin install` (see *Command Line Client*) to be able to easily import the functions.

8.2 Workbook preparation

The easiest way to start a new project is to run `xlwings quickstart myproject` on a command prompt (see *Command Line Client*). Alternative ways of getting the xlwings VBA module into your workbook are described under *VBA: Calling Python from Excel*

8.3 A simple UDF

The default settings (see *VBA settings*) expect a Python source file in the way it is created by `quickstart`:

- in the same directory as the Excel file

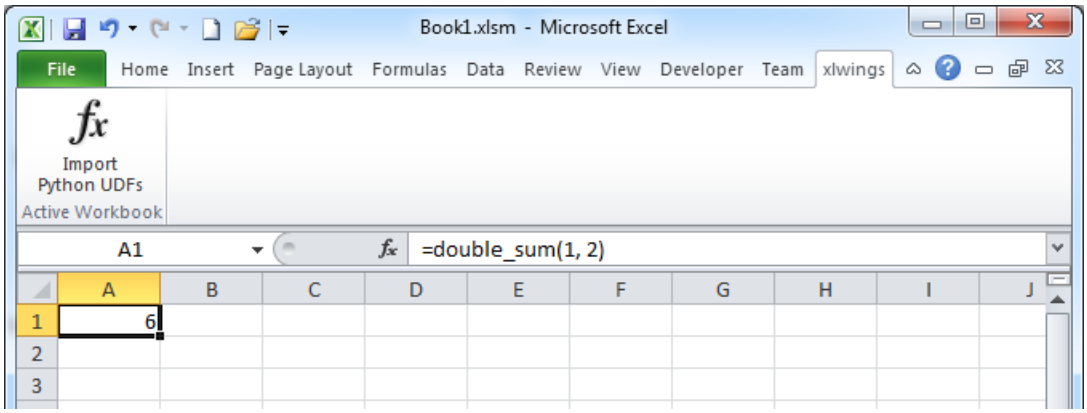
- with the same name as the Excel file, but with a `.py` ending instead of `.xlsm`.

Alternatively, you can point to a specific source file by setting the `UDF_PATH` in the VBA settings.

Let's assume you have a Workbook `myproject.xlsm`, then you would write the following code in `myproject.py`:

```
import xlwings as xw

@xw.func
def double_sum(x, y):
    """Returns twice the sum of the two arguments"""
    return 2 * (x + y)
```

- Now click on `Import Python UDFs` in the xlwings tab to pick up the changes made to `myproject.py`. If you don't want to install/use the add-in, you could also run the `ImportPythonUDFs` macro directly (one possibility to do that is to hit `Alt + F8` and select the macro from the pop-up menu).
- Enter the formula `=double_sum(1, 2)` into a cell and you will see the correct result:



- This formula can be used in VBA, too.
- The docstring (in triple-quotes) will be shown as function description in Excel.

Note:

- You only need to re-import your functions if you change the function arguments or the function name.
- Code changes in the actual functions are picked up automatically (i.e. at the next calculation of the formula, e.g. triggered by `Ctrl-Alt-F9`), but changes in imported modules are not. This is the very behaviour of how Python imports work. The easiest way to come around this is by working with the debug server that can easily be restarted, see: *Debugging*. If you aren't working with the debug server, the `pythonw.exe` process currently has to be killed via Windows Task Manager.
- The `@xw.func` decorator is only used by xlwings when the function is being imported into Excel. It tells xlwings for which functions it should create a VBA wrapper function, otherwise it has no effect on how the functions behave in Python.

8.4 Array formulas: Get efficient

Calling one big array formula in Excel is much more efficient than calling many single-cell formulas, so it's generally a good idea to use them, especially if you hit performance problems.

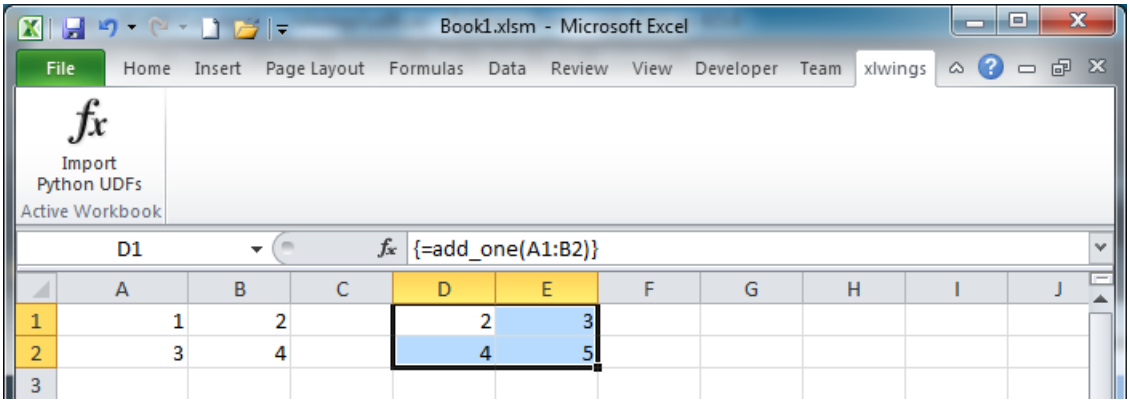
You can pass an Excel Range as a function argument, as opposed to a single cell and it will show up in Python as list of lists.

For example, you can write the following function to add 1 to every cell in a Range:

```
@xw.func
def add_one(data):
    return [[cell + 1 for cell in row] for row in data]
```

To use this formula in Excel,

- Click on `Import Python UDFs` again
- Fill in the values in the range `A1:B2`
- Select the range `D1:E2`
- Type in the formula `=add_one(A1:B2)`
- Press `Ctrl+Shift+Enter` to create an array formula. If you did everything correctly, you'll see the formula surrounded by curly braces as in this screenshot:



8.4.1 Number of array dimensions: ndim

The above formula has the issue that it expects a "two dimensional" input, e.g. a nested list of the form `[[1, 2], [3, 4]]`. Therefore, if you would apply the formula to a single cell, you would get the following error: `TypeError: 'float' object is not iterable`.

To force Excel to always give you a two-dimensional array, no matter whether the argument is a single cell, a column/row or a two-dimensional Range, you can extend the above formula like this:

```
@xw.func
@xw.arg('data', ndim=2)
```

(continues on next page)

(continued from previous page)

```
def add_one(data):
    return [[cell + 1 for cell in row] for row in data]
```

8.5 Array formulas with NumPy and Pandas

Often, you'll want to use NumPy arrays or Pandas DataFrames in your UDF, as this unlocks the full power of Python's ecosystem for scientific computing.

To define a formula for matrix multiplication using numpy arrays, you would define the following function:

```
import xlwings as xw
import numpy as np

@xw.func
@xw.arg('x', np.array, ndim=2)
@xw.arg('y', np.array, ndim=2)
def matrix_mult(x, y):
    return x @ y
```

Note: If you are not on Python >= 3.5 with NumPy >= 1.10, use `x.dot(y)` instead of `x @ y`.

A great example of how you can put Pandas at work is the creation of an array-based `CORREL` formula. Excel's version of `CORREL` only works on 2 datasets and is cumbersome to use if you want to quickly get the correlation matrix of a few time-series, for example. Pandas makes the creation of an array-based `CORREL2` formula basically a one-liner:

```
import xlwings as xw
import pandas as pd

@xw.func
@xw.arg('x', pd.DataFrame, index=False, header=False)
@xw.ret(index=False, header=False)
def CORREL2(x):
    """Like CORREL, but as array formula for more than 2 data sets"""
    return x.corr()
```

8.6 @xw.arg and @xw.ret decorators

These decorators are to UDFs what the `options` method is to `Range` objects: they allow you to apply converters and their options to function arguments (`@xw.arg`) and to the return value (`@xw.ret`). For example, to convert the argument `x` into a pandas DataFrame and suppress the index when returning it, you would do the following:

```
@xw.func
@xw.arg('x', pd.DataFrame)
@xw.ret(index=False)
def myfunction(x):
    # x is a DataFrame, do something with it
    return x
```

For further details see the *Converters and Options* documentation.

8.7 The "vba" keyword

It's often helpful to get the address of the calling cell. Right now, one of the easiest ways to accomplish this is to use the `vba` keyword. `vba`, in fact, allows you to access any available VBA expression e.g. `Application`. Note, however, that currently you're acting directly on the pywin32 COM object:

```
@xw.func
@xw.arg('xl_app', vba='Application')
def get_caller_address(xl_app)
    return xl_app.Caller.Address
```

8.8 Macros

On Windows, as alternative to calling macros via *RunPython*, you can also use the `@xw.sub` decorator:

```
import xlwings as xw

@xw.sub
def my_macro():
    """Writes the name of the Workbook into Range("A1") of Sheet 1"""
    wb = xw.Workbook.caller()
    xw.Range(1, 'A1').value = wb.name
```

After clicking on `Import Python UDFs`, you can then use this macro by executing it via `Alt + F8` or by binding it e.g. to a button. To to the latter, make sure you have the `Developer` tab selected under `File > Options > Customize Ribbon`. Then, under the `Developer` tab, you can insert a button via `Insert > Form Controls`. After drawing the button, you will be prompted to assign a macro to it and you can select `my_macro`.

CHAPTER 9

Converters and Options

Introduced with v0.7.0, converters define how Excel ranges and their values are converted both during **reading** and **writing** operations. They also provide a consistent experience across **xlwings.Range** objects and **User Defined Functions** (UDFs).

Converters are explicitely set in the `options` method when manipulating `xlwings.Range` objects or in the `@xw.arg` and `@xw.ret` decorators when using UDFs. If no converter is specified, the default converter is applied when reading. When writing, xlwings will automatically apply the correct converter (if available) according to the object's type that is being written to Excel. If no converter is found for that type, it falls back to the default converter.

Syntax:

	Range	UDF
reading	`Range.options(convert=None, **kwargs).value`	`@arg('x', convert=None, **kwargs)`
writing	`Range.options(convert=None, **kwargs).value = myvalue`	`@ret(convert=None, **kwargs)`

Note: Keyword arguments (`kwargs`) may refer to the specific converter or the default converter. For example, to set the `numbers` option in the default converter and the `index` option in the DataFrame converter, you would write:

```
Range('A1:C3').options(pd.DataFrame, index=False, numbers=int).value
```

9.1 Default Converter

If no options are set, the following conversions are performed:

- single cells are read in as `floats` in case the Excel cell holds a number, as `unicode` in case it holds text, as `datetime` if it contains a date and as `None` in case it is empty.
- columns/rows are read in as lists, e.g. `[None, 1.0, 'a string']`
- 2d cell ranges are read in as list of lists, e.g. `[[None, 1.0, 'a string'], [None, 2.0, 'another string']]`

The following options can be set:

- **ndim**

  Force the value to have either 1 or 2 dimensions regardless of the shape of the range:

```
>>> import xlwings as xw
>>> wb = Workbook()
>>> xw.Range('A1').value = [[1, 2], [3, 4]]
>>> xw.Range('A1').value
1.0
>>> xw.Range('A1').options(ndim=1).value
[1.0]
>>> xw.Range('A1').options(ndim=2).value
[[1.0]]
>>> xw.Range('A1:A2').value
[1.0 3.0]
>>> xw.Range('A1:A2').options(ndim=2).value
[[1.0], [3.0]]
```

- **numbers**

  By default cells with numbers are read as `float`, but you can change it to `int`:

```
>>> xw.Range('A1').value = 1
>>> xw.Range('A1').value
1.0
>>> xw.Range('A1').options(numbers=int).value
1
```

  Alternatively, you can specify any other function or type which takes a single float argument.

  Using this on UDFs looks like this:

```
@xw.func
@xw.arg('x', numbers=int)
def myfunction(x):
    # all numbers in x arrive as int
    return x
```

  Note: Excel always stores numbers internally as floats.

- **dates**

By default cells with dates are read as `datetime.datetime`, but you can change it to `datetime.date`:

- Range:

```
>>> import datetime as dt
>>> xw.Range('A1').options(dates=dt.date).value
```

- UDFs: `@xw.arg('x', dates=dt.date)`

Alternatively, you can specify any other function or type which takes the same keyword arguments as `datetime.datetime`, for example:

```
>>> my_date_handler = lambda year, month, day, **kwargs: "%04i-%02i-%02i
↪" % (year, month, day)
>>> xw.Range('A1').options(dates=my_date_handler).value
'2017-02-20'
```

- **empty**

  Empty cells are converted per default into `None`, you can change this as follows:

  - Range: `>>> xw.Range('A1').options(empty='NA').value`
  - UDFs: `@xw.arg('x', empty='NA')`

- **transpose**

  This works for reading and writing and allows us to e.g. write a list in column orientation to Excel:

  - Range: `Range('A1').options(transpose=True).value = [1, 2, 3]`
  - UDFs:

```
@xw.arg('x', transpose=True)
@xw.ret(transpose=True)
def myfunction(x):
    # x will be returned unchanged as transposed both when reading␣
↪and writing
    return x
```

- **expand**

  This works the same as the Range properties `table`, `vertical` and `horizontal` but is only evaluated when getting the values of a Range:

```
>>> import xlwings as xw
>>> wb = xw.Workbook()
>>> xw.Range('A1').value = [[1,2], [3,4]]
>>> rng1 = xw.Range('A1').table
>>> rng2 = xw.Range('A1').options(expand='table')
>>> rng1.value
[[1.0, 2.0], [3.0, 4.0]]
>>> rng2.value
[[1.0, 2.0], [3.0, 4.0]]
```

(continues on next page)

(continued from previous page)

```
>>> xw.Range('A3').value = [5, 6]
>>> rng1.value
[[1.0, 2.0], [3.0, 4.0]]
>>> rng2.value
[[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]]
```

Note: The `expand` option is only available on `Range` objects as UDFs only allow to manipulate the calling cells.

9.2 Built-in Converters

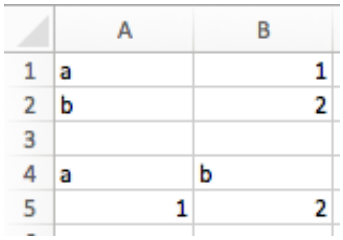
xlwings offers several built-in converters that perform type conversion to **dictionaries**, **NumPy arrays**, **Pandas Series** and **DataFrames**. These build on top of the default converter, so in most cases the options described above can be used in this context, too (unless they are meaningless, for example the `ndim` in the case of a dictionary).

It is also possible to write and register custom converter for additional types, see below.

The samples below can be used with both `xlwings.Range` objects and UDFs even though only one version may be shown.

9.2.1 Dictionary converter

The dictionary converter turns two Excel columns into a dictionary. If the data is in row orientation, use `transpose`:



	A	B
1	a	1
2	b	2
3		
4	a	b
5	1	2

```
>>> Range('A1:B2').options(dict).value
{'a': 1.0, 'b': 2.0}
>>> Range('A4:B5').options(dict, transpose=True).value
{'a': 1.0, 'b': 2.0}
```

9.2.2 Numpy array converter

options: `dtype=None, copy=True, order=None, ndim=None`

The first 3 options behave the same as when using `np.array()` directly. Also, `ndim` works the same as shown above for lists (under default converter) and hence returns either numpy scalars, 1d arrays or 2d arrays.

Example:

```
>>> import numpy as np
>>> Range('A1').options(transpose=True).value = np.array([1, 2, 3])
>>> xw.Range('A1:A3').options(np.array, ndim=2).value
array([[ 1.],
       [ 2.],
       [ 3.]])
```

9.2.3 Pandas Series converter

options: `dtype=None, copy=False, index=1, header=True`

The first 2 options behave the same as when using `pd.Series()` directly. `ndim` doesn't have an effect on Pandas series as they are always expected and returned in column orientation.

`index`: int or Boolean

When reading, it expects the number of index columns shown in Excel.
When writing, include or exclude the index by setting it to `True` or `False`.

`header`: Boolean

When reading, set it to `False` if Excel doesn't show either index or series names.
When writing, include or exclude the index and series names by setting it to `True` or `False`.

For `index` and `header`, `1` and `True` may be used interchangeably.

Example:

	A	B	C	D	E
1	date	series name		01/01/01	1
2	01/01/01	1		02/01/01	2
3	02/01/01	2		03/01/01	3
4	03/01/01	3		04/01/01	4
5	04/01/01	4		05/01/01	5
6	05/01/01	5		06/01/01	6
7	06/01/01	6			

```
>>> s = xw.Range('A1').options(pd.Series, expand='table').value
>>> s
date
2001-01-01    1
2001-01-02    2
2001-01-03    3
2001-01-04    4
2001-01-05    5
```

(continues on next page)

(continued from previous page)

```
2001-01-06    6
Name: series name, dtype: float64
>>> xw.Range('D1', header=False).value = s
```

9.2.4 Pandas DataFrame converter

options: `dtype=None, copy=False, index=1, header=1`

The first 2 options behave the same as when using `pd.DataFrame()` directly. `ndim` doesn't have an effect on Pandas DataFrames as they are automatically read in with `ndim=2`.

`index`: int or Boolean

When reading, it expects the number of index columns shown in Excel.

When writing, include or exclude the index by setting it to `True` or `False`.

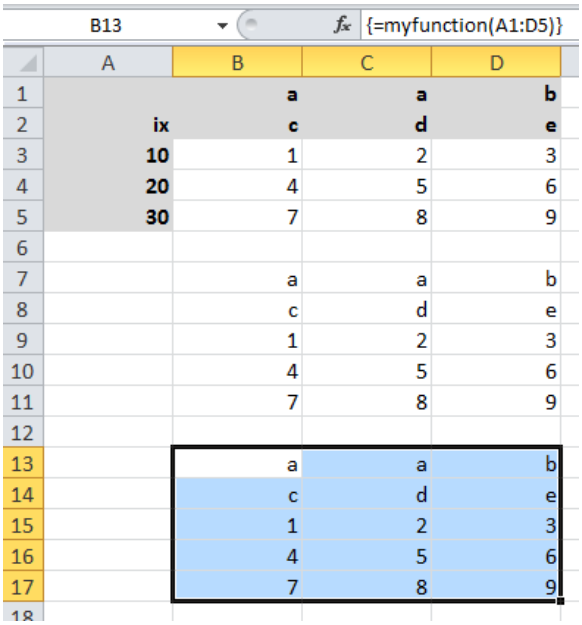
`header`: int or Boolean

When reading, it expects the number of column headers shown in Excel.

When writing, include or exclude the index and series names by setting it to `True` or `False`.

For `index` and `header`, `1` and `True` may be used interchangeably.

Example:




```
>>> df = xw.Range('A1:D5').options(pd.DataFrame, header=2).value
>>> df
    a     b
    c  d  e
ix
10  1  2  3
```

(continues on next page)

(continued from previous page)

```
20   4   5   6
30   7   8   9

# Writing back using the defaults:
>>> Range('A1').value = df

# Writing back and changing some of the options, e.g. getting rid of the␣
↪index:
>>> Range('B7').options(index=False).value = df
```

The same sample for **UDF** (starting in `Range('A13')` on screenshot) looks like this:

```
@xw.func
@xw.arg('x', pd.DataFrame, header=2)
@xw.ret(index=False)
def myfunction(x):
    # x is a DataFrame, do something with it
    return x
```

9.2.5 xw.Range and 'raw' converters

Technically speaking, these are "no-converters".

- If you need access to the `xlwings.Range` object directly, you can do:

  ```
  @xw.func
  @xw.arg('x', xw.Range)
  def myfunction(x):
      return x.formula
  ```

  This returns x as `xlwings.Range` object, i.e. without applying any converters or options.

- The `raw` converter delivers the values unchanged from the underlying libraries (`pywin32` on Windows and `appscript` on Mac), i.e. no sanitizing/cross-platform harmonizing of values are being made. This might be useful in a few cases for efficiency reasons. E.g:

  ```
  >>> Range('A1:B2').value
  [[1.0, 'text'], [datetime.datetime(2016, 2, 1, 0, 0), None]]

  >>> Range('A1:B2').options('raw').value
  ((1.0, 'text'), (pywintypes.datetime(2016, 2, 1, 0, 0,␣
  ↪tzinfo=TimeZoneInfo('GMT Standard Time', True)), None))
  ```

9.3 Custom Converter

Here are the steps to implement your own converter:

- Inherit from `xlwings.conversion.Converter`

- Implement both a `read_value` and `write_value` method as static- or classmethod:
  - In `read_value`, `value` is what the base converter returns: hence, if no `base` has been specified it arrives in the format of the default converter.
  - In `write_value`, `value` is the original object being written to Excel. It must be returned in the format that the base converter expects. Again, if no `base` has been specified, this is the default converter.

  The `options` dictionary will contain all keyword arguments specified in the `Range.options` method, e.g. when calling `Range('A1').options(myoption='some value')` or as specified in the `@arg` and `@ret` decorator when using UDFs. Here is the basic structure:

```
from xlwings.conversion import Converter

class MyConverter(Converter):

    @staticmethod
    def read_value(value, options):
        myoption = options.get('myoption', default_value)
        return_value = value  # Implement your conversion here
        return return_value

    @staticmethod
    def write_value(value, options):
        myoption = options.get('myoption', default_value)
        return_value = value  # Implement your conversion here
        return return_value
```

- Optional: set a `base` converter (`base` expects a class name) to build on top of an existing converter, e.g. for the built-in ones: `DictCoverter`, `NumpyArrayConverter`, `PandasDataFrameConverter`, `PandasSeriesConverter`
- Optional: register the converter: you can **(a)** register a type so that your converter becomes the default for this type during write operations and/or **(b)** you can register an alias that will allow you to explicitly call your converter by name instead of just by class name

The following examples should make it much easier to follow - it defines a DataFrame converter that extends the built-in DataFrame converter to add support for dropping nan's:

```
from xlwings.conversion import Converter, PandasDataFrameConverter

class DataFrameDropna(Converter):

    base = PandasDataFrameConverter

    @staticmethod
    def read_value(builtin_df, options):
        dropna = options.get('dropna', False)  # set default to False
        if dropna:
            converted_df = builtin_df.dropna()
        else:
            converted_df = builtin_df
```

(continues on next page)

(continued from previous page)

```
        # This will arrive in Python when using the DataFrameDropna converter␣
↪for reading
        return converted_df

    @staticmethod
    def write_value(df, options):
        dropna = options.get('dropna', False)
        if dropna:
            converted_df = df.dropna()
        else:
            converted_df = df
        # This will be passed to the built-in PandasDataFrameConverter when␣
↪writing
        return converted_df
```

Now let's see how the different converters can be applied:

```
# Fire up a Workbook and create a sample DataFrame
wb = Workbook()
df = pd.DataFrame([[1.,10.],[2.,np.nan], [3., 30.]])
```

- Default converter for DataFrames:

```
# Write
Range('A1').value = df

# Read
Range('A1:C4').options(pd.DataFrame).value
```

- DataFrameDropna converter:

```
# Write
Range('A7').options(DataFrameDropna, dropna=True).value = df

# Read
Range('A1:C4').options(DataFrameDropna, dropna=True).value
```

- Register an alias (optional):

```
DataFrameDropna.register('df_dropna')

# Write
Range('A12').options('df_dropna', dropna=True).value = df

# Read
Range('A1:C4').options('df_dropna', dropna=True).value
```

- Register DataFrameDropna as default converter for DataFrames (optional):

```
DataFrameDropna.register(pd.DataFrame)
```

(continues on next page)

(continued from previous page)

```
# Write
Range('A13').options(dropna=True).value = df

# Read
Range('A1:C4').options(pd.DataFrame, dropna=True).value
```

These samples all work the same with UDFs, e.g.:

```
@xw.func
@arg('x', DataFrameDropna, dropna=True)
@ret(DataFrameDropna, dropna=True)
def myfunction(x):
    # ...
    return x
```

Note: Python objects run through multiple stages of a transformation pipeline when they are being written to Excel. The same holds true in the other direction, when Excel/COM objects are being read into Python.

Pipelines are internally defined by `Accessor` classes. A Converter is just a special Accessor which converts to/from a particular type by adding an extra stage to the pipeline of the default Accessor. For example, the `PandasDataFrameConverter` defines how a list of list (as delivered by the default Accessor) should be turned into a Pandas DataFrame.

The `Converter` class provides basic scaffolding to make the task of writing a new Converter easier. If you need more control you can subclass `Accessor` directly, but this part requires more work and is currently undocumented.

CHAPTER 10

Command Line Client

xlwings comes with a command line client that makes it easy to set up workbooks and install the developer add-in. On Windows, type the commands into a `Command Prompt`, on Mac, type them into a `Terminal`.

10.1 Quickstart

- `xlwings quickstart myproject`

This command is by far the fastest way to get off the ground: It creates a new folder `myproject` with the necessary Excel workbook (including the xlwings VBA module) and a Python file, ready to be used right away:

```
myproject
  |--myproject.xlsm
  |--myproject.py
```

New in version 0.6.4.

10.2 Add-in (Currently Windows-only)

The add-in is currently in an early stage and only provides one button to import User Defined Functions (UDFs). As such, it is only a developer add-in and not necessary to run Workbooks with xlwings.

Note: Excel needs to be closed before installing/updating the add-in. If you're still getting an error, start the Task Manager and make sure there are no `EXCEL.EXE` processes left.

- `xlwings addin install`: Copies the xlwings add-in to the XLSTART folder
- `xlwings addin update`: Replaces the current add-in with the latest one
- `xlwings addin remove`: Removes the add-in from the XLSTART folder
- `xlwings addin status`: Shows if the add-in is installed together with the installation path

After installing the add-in, it will be available as xlwings tab on the Excel Ribbon.

New in version 0.6.0.

10.3 Template

- `xlwings template open`: Opens a new Workbook with the xlwings VBA module
- `xlwings template install`: Copies the xlwings template file to the correct Excel folder, see below
- `xlwings template update`: Replaces the current xlwings template with the latest one
- `xlwings template remove`: Removes the template from Excel's template folder
- `xlwings template status`: Shows if the template is installed together with the installation path

After installing, the templates are accessible via Excel's Menu:

- Win (Excel 2007, 2010): `File > New > My templates`
- Win (Excel 2013, 2016): There's an additional step needed as explained here
- Mac (Excel 2011, 2016): `File > New from template`

New in version 0.6.0.

10.4 RunPython

Only required if you are on Mac, are using Excel 2016 and have xlwings installed via conda or as part of Anaconda. To enable the `RunPython` calls in VBA, run this one time:

```
xlwings runpython install
```

Alternatively, install xlwings with `pip`.

New in version 0.7.0.

CHAPTER 11

Missing Features

If you're missing a feature in xlwings, do the following:

1. Most importantly, open an issue on GitHub. If it's something bigger or if you want support from other users, consider opening a feature request. Adding functionality should be user driven, so only if you tell us about what you're missing, it's eventually going to find its way into the library. By the way, we also appreciate pull requests!

2. Workaround: in essence, xlwings is just a smart wrapper around pywin32 on Windows and appscript on Mac. You can access the underlying objects by doing:

```
>>> wb = Workbook('Book1')
>>> wb.xl_workbook
<COMObject <unknown>>  # Windows/pywin32
app('/Applications/Microsoft Excel.app').workbooks['Book1']  # Mac/
↪appscript
```

   This works accordingly for `Range.xl_range`, `Sheet.xl_sheet`, `Chart.xl_chart` etc.

   The underlying objects will offer you pretty much everything you can do with VBA, using the syntax of pywin32 (which pretty much feels like VBA) and appscript (which doesn't feel like VBA). But apart from looking ugly, keep in mind that **it makes your code platform specific (!)**, i.e. even if you go for option 2), you should still follow option 1) and open an issue so the feature finds it's way into the library (cross-platform and with a Pythonic syntax).

11.1 Example: Workaround to use VBA's `Range.WrapText`

```
# Windows
Range('A1').xl_range.WrapText = True
```

(continues on next page)

(continued from previous page)

```
# Mac
Range('A1').xl_range.wrap_text.set(True)
```

CHAPTER 12

xlwings with R and Julia

While xlwings is a pure Python package, there are cross-language packages that allow for a relative straightforward use from/with other languages. This means, however, that you'll always need to have Python with xlwings installed in addition to R or Julia. We recommend the Anaconda distribution, see also *Installation*.

12.1 R

The R instructions are for Windows, but things work accordingly on Mac except that calling the R functions as User Defined Functions is not supported at the moment (but `RunPython` works, see *Call Python with "RunPython"*).

Setup:

- Install R and Python
- Add `R_HOME` environment variable to base directory of installation, .e.g `C:\Program Files\R\R-x.x.x`
- Add `R_USER` environment variable to user folder, e.g. `C:\Users\<user>`
- Add `C:\Program Files\R\R-x.x.x\bin` to `PATH`
- Restart Windows because of the environment variables (!)

12.1.1 Simple functions with R

Original R function that we want to access from Excel (saved in `r_file.R`):

```
myfunction <- function(x, y){
    return(x * y)
}
```

Python wrapper code:

```
import xlwings as xw
import rpy2.robjects as robjects
# you might want to use some relative path or place the file in R's current␣
↪working dir
robjects.r.source(r"C:\path\to\r_file.R")

@xw.func
def myfunction(x, y):
    myfunc = robjects.r['myfunction']
    return tuple(myfunc(x, y))
```

After importing this function (see: *UDF Tutorial*), it will be available as UDF from Excel.

12.1.2 Array functions with R

Original R function that we want to access from Excel (saved in `r_file.R`):

```
array_function <- function(m1, m2){
  # Matrix multiplication
  return(m1 %*% m2)
}
```

Python wrapper code:

```
import xlwings as xw
import numpy as np
import rpy2.robjects as robjects
from rpy2.robjects import numpy2ri

robjects.r.source(r"C:\path\to\r_file.R")
numpy2ri.activate()

@xw.func
@xw.arg("x", np.array, ndim=2)
@xw.arg("y", np.array, ndim=2)
def array_function(x, y):
    array_func = robjects.r['array_function']
    return np.array(array_func(x, y))
```

After importing this function (see: *UDF Tutorial*), it will be available as UDF from Excel.

12.2 Julia

Setup:

- Install Julia and Python
- Run `Pkg.add("PyCall")` from an interactive Julia interpreter

xlwings can then be called from Julia with the following syntax (the colons take care of automatic type conversion):

```
julia> using PyCall
julia> @pyimport xlwings as xw

julia> xw.Workbook()
PyObject <Workbook 'Workbook1'>

julia> xw.Range("A1")[:value] = "Hello World"
julia> xw.Range("A1")[:value]
"Hello World"


julia> xw.Range("A1")[:value] = [1 2; 3 4]
julia> xw.Range("A1")[:table][:value]
2x2 Array{Int64,2}:
1  2
3  4
```

CHAPTER 13

API Documentation

The xlwings object model is very similar to the one used by Excel VBA but the hierarchy is flattened. An example:

VBA:

```
Workbooks("Book1").Sheets("Sheet1").Range("A1").Value = "Some Text"
```

xlwings:

```
wb = Workbook("Book1")
Range("Sheet1", "A1").value = "Some Text"
```

13.1 Top-level functions

xlwings.**view**(*obj*)

Opens a new workbook and displays an object on its first sheet.

Parameters **obj** (*any type with built-in converter*) – the object to display

```
>>> import xlwings as xw
>>> import pandas as pd
>>> import numpy as np
>>> df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b
↪', 'c', 'd'])
>>> xw.view(df)
```

New in version 0.7.1.

13.2 Application

class xlwings.**Application**(*wkb*)

Application is dependent on the Workbook since there might be different application instances on Windows.

version

Returns Excel's version string.

New in version 0.5.0.

quit()

Quits the application without saving any workbooks.

New in version 0.3.3.

screen_updating

True if screen updating is turned on. Read/write Boolean.

New in version 0.3.3.

visible

Gets or sets the visibility of Excel to `True` or `False`. This property can also be conveniently set during instantiation of a new Workbook: `Workbook(app_visible=False)`

New in version 0.3.3.

calculation

Returns or sets a Calculation value that represents the calculation mode.

Example

```
>>> from xlwings import Workbook, Application
>>> from xlwings.constants import Calculation
>>> wb = Workbook()
>>> Application(wkb=wb).calculation = Calculation.xlCalculationManual
```

New in version 0.3.3.

calculate()

Calculates all open Workbooks

New in version 0.3.6.

13.3 Workbook

In order to use xlwings, instantiating a workbook object is always the first thing to do:

class xlwings.**Workbook**(*fullname=None*, *xl_workbook=None*, *app_visible=True*, *app_target=None*)

`Workbook` connects an Excel Workbook with Python. You can create a new connection from Python with

- a new workbook: `wb = Workbook()`
- the active workbook: `wb = Workbook.active()`
- an unsaved workbook: `wb = Workbook('Book1')`
- a saved (open) workbook by name (incl. xlsx etc): `wb = Workbook('MyWorkbook.xlsx')`
- a saved (open or closed) workbook by path: `wb = Workbook(r'C:\path\to\file.xlsx')`

Keyword Arguments

- **fullname** (*str, default None*) – Full path or name (incl. xlsx, xlsm etc.) of existing workbook or name of an unsaved workbook.
- **xl_workbook** (*pywin32 or appscript Workbook object, default None*) – This enables to turn existing Workbook objects of the underlying libraries into xlwings objects
- **app_visible** (*boolean, default True*) – The resulting Workbook will be visible by default. To open it without showing a window, set `app_visible=False`. Or, to not alter the visibility (e.g., if Excel is already running), set `app_visible=None`. Note that this property acts on the whole Excel instance, not just the specific Workbook.
- **app_target** (*str, default None*) – Mac-only, use the full path to the Excel application, e.g. `/Applications/Microsoft Office 2011/Microsoft Excel` or `/Applications/Microsoft Excel`

  On Windows, if you want to change the version of Excel that xlwings talks to, go to `Control Panel > Programs and Features` and `Repair` the Office version that you want as default.

To create a connection when the Python function is called from Excel, use:

```
wb = Workbook.caller()
```

classmethod active(*app_target=None*)

Returns the Workbook that is currently active or has been active last. On Windows, this works across all instances.

New in version 0.4.1.

classmethod caller()

Creates a connection when the Python function is called from Excel:

```
wb = Workbook.caller()
```

Always pack the `Workbook` call into the function being called from Excel, e.g.:

```
def my_macro():
    wb = Workbook.caller()
    Range('A1').value = 1
```

To be able to easily invoke such code from Python for debugging, use `Workbook.set_mock_caller()`.

New in version 0.3.0.

static set_mock_caller (*fullpath*)

Sets the Excel file which is used to mock `Workbook.caller()` when the code is called from within Python.

Examples

```
# This code runs unchanged from Excel and Python directly
import os
from xlwings import Workbook, Range

def my_macro():
    wb = Workbook.caller()
    Range('A1').value = 'Hello xlwings!'

if __name__ == '__main__':
    # Mock the calling Excel file
    Workbook.set_mock_caller(r'C:\path\to\file.xlsx')
    my_macro()
```

New in version 0.3.1.

classmethod current()

Returns the current Workbook object, i.e. the default Workbook used by `Sheet`, `Range` and `Chart` if not specified otherwise. On Windows, in case there are various instances of Excel running, opening an existing or creating a new Workbook through `Workbook()` is acting on the same instance of Excel as this Workbook. Use like this: `Workbook.current()`.

New in version 0.2.2.

set_current()

This makes the Workbook the default that `Sheet`, `Range` and `Chart` use if not specified otherwise. On Windows, in case there are various instances of Excel running, opening an existing or creating a new Workbook through `Workbook()` is acting on the same instance of Excel as this Workbook.

New in version 0.2.2.

get_selection()

Returns the currently selected cells from Excel as `Range` object.

Example

```
>>> import xlwings as xw
>>> wb = xw.Workbook.active()
>>> wb.get_selection()
<Range on Sheet 'Sheet1' of Workbook 'Workbook1'>
>>> wb.get_selection.value
[[1.0, 2.0], [3.0, 4.0]]
>>> wb.get_selection().options(transpose=True).value
[[1.0, 3.0], [2.0, 4.0]]
```

Returns

Return type Range object

close()
Closes the Workbook without saving it.

New in version 0.1.1.

save(*path=None*)
Saves the Workbook. If a path is being provided, this works like SaveAs() in Excel. If no path is specified and if the file hasn't been saved previously, it's being saved in the current working directory with the current filename. Existing files are overwritten without prompting.

Parameters **path** (*str, default None*) – Full path to the workbook

Example

```
>>> from xlwings import Workbook
>>> wb = Workbook()
>>> wb.save()
>>> wb.save(r'C:\path\to\new_file_name.xlsx')
```

New in version 0.3.1.

static get_xl_workbook(*wkb*)
Returns the `xl_workbook_current` if `wkb` is `None`, otherwise the `xl_workbook` of `wkb`. On Windows, `xl_workbook` is a pywin32 COM object, on Mac it's an appscript object.

Parameters **wkb** (*Workbook or None*) – Workbook object

static open_template()
Creates a new Excel file with the xlwings VBA module already included. This method must be called from an interactive Python shell:

```
>>> Workbook.open_template()
```

New in version 0.3.3.

names
A collection of all the (platform-specific) name objects in the application or workbook. Each name object represents a defined name for a range of cells (built-in or custom ones).

New in version 0.4.0.

macro(*name*)
Runs a Sub or Function in Excel VBA.

name : Name of Sub or Function with or without module name, e.g. `'Module1.MyMacro'` or `'MyMacro'`

This VBA function:

```
Function MySum(x, y)
    MySum = x + y
End Function
```

can be accessed like this:

```
>>> wb = xw.Workbook.active()
>>> my_sum = wb.macro('MySum')
>>> my_sum(1, 2)
3
```

New in version 0.7.1.

13.4 Sheet

Sheet objects allow you to interact with anything directly related to a Sheet.

class xlwings.**Sheet**(*sheet*, *wkb=None*)
Represents a Sheet of the current Workbook. Either call it with the Sheet name or index:

```
Sheet('Sheet1')
Sheet(1)
```

Parameters **sheet** (*str or int*) – Sheet name or index

Keyword Arguments **wkb** (*Workbook object, default Workbook.current()*) – Defaults to the Workbook that was instantiated last or set via `Workbook.set_current()`.

New in version 0.2.3.

activate()
Activates the sheet.

autofit(*axis=None*)
Autofits the width of either columns, rows or both on a whole Sheet.

Parameters **axis** (*string, default None*) –

- To autofit rows, use one of the following: `rows` or `r`
- To autofit columns, use one of the following: `columns` or `c`
- To autofit rows and columns, provide no arguments

Examples

```
# Autofit columns
Sheet('Sheet1').autofit('c')
# Autofit rows
Sheet('Sheet1').autofit('r')
# Autofit columns and rows
Range('Sheet1').autofit()
```

New in version 0.2.3.

clear_contents()
Clears the content of the whole sheet but leaves the formatting.

clear()
Clears the content and formatting of the whole sheet.

name
Get or set the name of the Sheet.

index
Returns the index of the Sheet.

classmethod active(*wkb=None*)
Returns the active Sheet. Use like so: `Sheet.active()`

classmethod add(*name=None*, *before=None*, *after=None*, *wkb=None*)
Creates a new worksheet: the new worksheet becomes the active sheet. If neither `before` nor `after` is specified, the new Sheet will be placed at the end.

Parameters

- **name** (*str, default None*) – Sheet name, defaults to Excel standard name
- **before** (*str or int, default None*) – Sheet name or index
- **after** (*str or int, default None*) – Sheet name or index

Returns

Return type Sheet object

Examples

```
>>> Sheet.add()  # Place at end with default name
>>> Sheet.add('NewSheet', before='Sheet1')  # Include name and␣
↪position
>>> new_sheet = Sheet.add(after=3)
>>> new_sheet.index
4
```

New in version 0.2.3.

static count(*wkb=None*)

Counts the number of Sheets.

Keyword Arguments **wkb** (*Workbook object, default Workbook.current()*) – Defaults to the Workbook that was instantiated last or set via Workbook.set_current().

Examples

```
>>> Sheet.count()
3
```

New in version 0.2.3.

static all(*wkb=None*)

Returns a list with all Sheet objects.

Keyword Arguments **wkb** (*Workbook object, default Workbook.current()*) – Defaults to the Workbook that was instantiated last or set via Workbook.set_current().

Examples

```
>>> Sheet.all()
[<Sheet 'Sheet1' of Workbook 'Book1'>, <Sheet 'Sheet2' of Workbook
↪'Book1'>]
>>> [i.name.lower() for i in Sheet.all()]
['sheet1', 'sheet2']
>>> [i.autofit() for i in Sheet.all()]
```

New in version 0.2.3.

delete()

Deletes the Sheet.

13.5 Range

The xlwings Range object represents a block of contiguous cells in Excel.

class xlwings.**Range**(**args*, *wkb=None*)

A Range object can be instantiated with the following arguments:

```
Range('A1')           Range('Sheet1', 'A1')           Range(1, 'A1')
Range('A1:C3')        Range('Sheet1', 'A1:C3')        Range(1, 'A1:C3')
Range((1,2))          Range('Sheet1, (1,2))           Range(1, (1,2))
Range((1,1), (3,3))   Range('Sheet1', (1,1), (3,3))   Range(1, (1,1), (3,
↪3))
Range('NamedRange')   Range('Sheet1', 'NamedRange')   Range(1, 'NamedRange
↪')
```

The Sheet can also be provided as Sheet object:

```
sh = Sheet(1)
Range(sh, 'A1')
```

If no worksheet name is provided as first argument, it will take the Range from the active sheet.

You usually want to go for `Range(...).value` to get the values (as list of lists). You can influence the reading/writing behavior by making use of *Converters and Options* and their options: `Range(...).options(...).value`

Parameters ***args** – Definition of sheet (optional) and Range in the above described combinations.

Keyword Arguments **wkb** (*Workbook object, default Workbook.current()*) – Defaults to the Workbook that was instantiated last or set via *Workbook.set_current()*‘.

options(*convert=None*, ***options*)

Allows you to set a converter and their options. Converters define how Excel Ranges and their values are being converted both during reading and writing operations. If no explicit converter is specified, the base converter is being applied, see *Converters and Options*.

Parameters **convert** (*object, default None*) – A converter, e.g. `dict`, `np.array`, `pd.DataFrame`, `pd.Series`, defaults to default converter

Keyword Arguments

- **ndim** (*int, default None*) – number of dimensions
- **numbers** (*type, default None*) – type of numbers, e.g. `int`
- **dates** (*type, default None*) – e.g. `datetime.date` defaults to `datetime.datetime`
- **empty** (*object, default None*) – transformation of empty cells
- **transpose** (*Boolean, default False*) – transpose values
- **expand** (*str, default None*) – One of `'table'`, `'vertical'`, `'horizontal'`, see also `Range.table` etc

  => For converter-specific options, see *Converters and Options*.

Returns

Return type Range object

New in version 0.7.0.

is_cell()
Returns `True` if the Range consists of a single Cell otherwise `False`.

New in version 0.1.1.

is_row()
Returns `True` if the Range consists of a single Row otherwise `False`.

New in version 0.1.1.

is_column()
Returns `True` if the Range consists of a single Column otherwise `False`.

New in version 0.1.1.

is_table()
Returns `True` if the Range consists of a 2d array otherwise `False`.

New in version 0.1.1.

shape
Tuple of Range dimensions.

New in version 0.3.0.

size
Number of elements in the Range.

New in version 0.3.0.

value
Gets and sets the values for the given Range.

Returns Empty cells are set to `None`.

Return type object

formula
Gets or sets the formula for the given Range.

formula_array
Gets or sets an array formula for the given Range.

New in version 0.7.1.

table
Returns a contiguous Range starting with the indicated cell as top-left corner and going down and right as long as no empty cell is hit.

Keyword Arguments **strict** (*boolean, default False*) – `True` stops the table at empty cells even if they contain a formula. Less efficient than if set to `False`.

Returns

Return type Range object

Examples

To get the values of a contiguous range or clear its contents use:

```
Range('A1').table.value
Range('A1').table.clear_contents()
```

vertical

Returns a contiguous Range starting with the indicated cell and going down as long as no empty cell is hit. This corresponds to `Ctrl-Shift-DownArrow` in Excel.

Parameters **strict** (*bool, default False*) – `True` stops the table at empty cells even if they contain a formula. Less efficient than if set to `False`.

Returns

Return type Range object

Examples

To get the values of a contiguous range or clear its contents use:

```
Range('A1').vertical.value
Range('A1').vertical.clear_contents()
```

horizontal

Returns a contiguous Range starting with the indicated cell and going right as long as no empty cell is hit.

Keyword Arguments **strict** (*bool, default False*) – `True` stops the table at empty cells even if they contain a formula. Less efficient than if set to `False`.

Returns

Return type Range object

Examples

To get the values of a contiguous Range or clear its contents use:

```
Range('A1').horizontal.value
Range('A1').horizontal.clear_contents()
```

current_region

This property returns a Range object representing a range bounded by (but not including) any combination of blank rows and blank columns or the edges of the worksheet. It corresponds to `Ctrl-*` on Windows and `Shift-Ctrl-Space` on Mac.

Returns

Return type Range object

number_format
Gets and sets the number_format of a Range.

Examples

```
>>> Range('A1').number_format
'General'
>>> Range('A1:C3').number_format = '0.00%'
>>> Range('A1:C3').number_format
'0.00%'
```

New in version 0.2.3.

clear()
Clears the content and the formatting of a Range.

clear_contents()
Clears the content of a Range but leaves the formatting.

column_width
Gets or sets the width, in characters, of a Range. One unit of column width is equal to the width of one character in the Normal style. For proportional fonts, the width of the character 0 (zero) is used.

If all columns in the Range have the same width, returns the width. If columns in the Range have different widths, returns None.

column_width must be in the range: 0 <= column_width <= 255

Note: If the Range is outside the used range of the Worksheet, and columns in the Range have different widths, returns the width of the first column.

Returns

Return type float

New in version 0.4.0.

row_height
Gets or sets the height, in points, of a Range. If all rows in the Range have the same height, returns the height. If rows in the Range have different heights, returns None.

row_height must be in the range: 0 <= row_height <= 409.5

Note: If the Range is outside the used range of the Worksheet, and rows in the Range have different heights, returns the height of the first row.

Returns

Return type float

New in version 0.4.0.

width
Returns the width, in points, of a Range. Read-only.

Returns

Return type float

New in version 0.4.0.

height
Returns the height, in points, of a Range. Read-only.

Returns

Return type float

New in version 0.4.0.

left
Returns the distance, in points, from the left edge of column A to the left edge of the range. Read-only.

Returns

Return type float

New in version 0.6.0.

top
Returns the distance, in points, from the top edge of row 1 to the top edge of the range. Read-only.

Returns

Return type float

New in version 0.6.0.

autofit(*axis=None*)
Autofits the width of either columns, rows or both.

Parameters **axis** (*string or integer, default None*) –

- To autofit rows, use one of the following: `rows` or `r`
- To autofit columns, use one of the following: `columns` or `c`
- To autofit rows and columns, provide no arguments

Examples

```
# Autofit column A
Range('A:A').autofit('c')
# Autofit row 1
```

(continues on next page)

(continued from previous page)

```
Range('1:1').autofit('r')
# Autofit columns and rows, taking into account Range('A1:E4')
Range('A1:E4').autofit()
# AutoFit rows, taking into account Range('A1:E4')
Range('A1:E4').autofit('rows')
```

New in version 0.2.2.

get_address(*row_absolute=True*, *column_absolute=True*, *include_sheetname=False*, *external=False*)

Returns the address of the range in the specified format.

Parameters

- **row_absolute** (*bool, default True*) – Set to True to return the row part of the reference as an absolute reference.
- **column_absolute** (*bool, default True*) – Set to True to return the column part of the reference as an absolute reference.
- **include_sheetname** (*bool, default False*) – Set to True to include the Sheet name in the address. Ignored if external=True.
- **external** (*bool, default False*) – Set to True to return an external reference with workbook and worksheet name.

Returns

Return type str

Examples

```
>>> Range((1,1)).get_address()
'$A$1'
>>> Range((1,1)).get_address(False, False)
'A1'
>>> Range('Sheet1', (1,1), (3,3)).get_address(True, False, True)
'Sheet1!A$1:C$3'
>>> Range('Sheet1', (1,1), (3,3)).get_address(True, False,␣
↪external=True)
'[Workbook1]Sheet1!A$1:C$3'
```

New in version 0.2.3.

hyperlink

Returns the hyperlink address of the specified Range (single Cell only)

Examples

```
>>> Range('A1').value
'www.xlwings.org'
>>> Range('A1').hyperlink
'http://www.xlwings.org'
```

New in version 0.3.0.

add_hyperlink(*address*, *text_to_display=None*, *screen_tip=None*)
Adds a hyperlink to the specified Range (single Cell)

Parameters

- **address** (*str*) – The address of the hyperlink.
- **text_to_display** (*str, default None*) – The text to be displayed for the hyperlink. Defaults to the hyperlink address.
- **screen_tip** (*str, default None*) – The screen tip to be displayed when the mouse pointer is paused over the hyperlink. Default is set to '<address> - Click once to follow. Click and hold to select this cell.'

New in version 0.3.0.

color
Gets and sets the background color of the specified Range.

To set the color, either use an RGB tuple `(0, 0, 0)` or a color constant. To remove the background, set the color to `None`, see Examples.

Returns RGB

Return type tuple

Examples

```
>>> Range('A1').color = (255,255,255)
>>> from xlwings import RgbColor
>>> Range('A2').color = RgbColor.rgbAqua
>>> Range('A2').color
(0, 255, 255)
>>> Range('A2').color = None
>>> Range('A2').color is None
True
```

New in version 0.3.0.

resize(*row_size=None*, *column_size=None*)
Resizes the specified Range

Parameters

- **row_size** (*int > 0*) – The number of rows in the new range (if None, the number of rows in the range is unchanged).
- **column_size** (*int > 0*) – The number of columns in the new range (if None, the number of columns in the range is unchanged).

Returns Range

Return type Range object

New in version 0.3.0.

offset(*row_offset=None*, *column_offset=None*)

Returns a Range object that represents a Range that's offset from the specified range.

Returns Range

Return type Range object

New in version 0.3.0.

column

Returns the number of the first column in the in the specified range. Read-only.

Returns

Return type Integer

New in version 0.3.5.

row

Returns the number of the first row in the in the specified range. Read-only.

Returns

Return type Integer

New in version 0.3.5.

last_cell

Returns the bottom right cell of the specified range. Read-only.

Returns

Return type Range object

Example

```
>>> rng = Range('A1').table
>>> rng.last_cell.row, rng.last_cell.column
(4, 5)
```

New in version 0.3.5.

name

Sets or gets the name of a Range.

To delete a named Range, use `del wb.names['NamedRange']` if `wb` is your Workbook object.

New in version 0.4.0.

13.6 Shape

class xlwings.**Shape**(**args*, ***kwargs*)

A Shape object represents an existing Excel shape and can be instantiated with the following arguments:

```
Shape(1)              Shape('Sheet1', 1)              Shape(1, 1)
Shape('Shape 1')      Shape('Sheet1', 'Shape 1')      Shape(1, 'Shape 1')
```

The Sheet can also be provided as Sheet object:

```
sh = Sheet(1)
Shape(sh, 'Shape 1')
```

If no Worksheet is provided as first argument, it will take the Shape from the active Sheet.

Parameters ***args** – Definition of Sheet (optional) and shape in the above described combinations.

Keyword Arguments **wkb** (*Workbook object, default Workbook.current()*) – Defaults to the Workbook that was instantiated last or set via `Workbook.set_current()`.

New in version 0.5.0.

name

Returns or sets a String value representing the name of the object.

New in version 0.5.0.

left

Returns or sets a value that represents the distance, in points, from the left edge of the object to the left edge of column A.

New in version 0.5.0.

top

Returns or sets a value that represents the distance, in points, from the top edge of the topmost shape in the shape range to the top edge of the worksheet.

New in version 0.5.0.

width

Returns or sets a value that represents the width, in points, of the object.

New in version 0.5.0.

height

Returns or sets a value that represents the height, in points, of the object.

New in version 0.5.0.

delete()
Deletes the object.

New in version 0.5.0.

activate()
Activates the object.

New in version 0.5.0.

13.7 Chart

Note: The chart object is currently still lacking a lot of important methods/attributes.

class xlwings.**Chart**(**args*, ***kwargs*)
Bases: xlwings.main.Shape

A Chart object represents an existing Excel chart and can be instantiated with the following arguments:

```
Chart(1)              Chart('Sheet1', 1)              Chart(1, 1)
Chart('Chart 1')      Chart('Sheet1', 'Chart 1')      Chart(1, 'Chart 1')
```

The Sheet can also be provided as Sheet object:

```
sh = Sheet(1)
Chart(sh, 'Chart 1')
```

If no Worksheet is provided as first argument, it will take the Chart from the active Sheet.

To insert a new Chart into Excel, create it as follows:

```
Chart.add()
```

Parameters ***args** – Definition of Sheet (optional) and chart in the above described combinations.

Keyword Arguments **wkb** (*Workbook object, default Workbook.current()*) – Defaults to the Workbook that was instantiated last or set via Workbook.set_current().

Example

```
>>> from xlwings import Workbook, Range, Chart, ChartType
>>> wb = Workbook()
>>> Range('A1').value = [['Foo1', 'Foo2'], [1, 2]]
```

(continues on next page)

(continued from previous page)

```
>>> chart = Chart.add(source_data=Range('A1').table, chart_
↪type=ChartType.xlLine)
>>> chart.name
'Chart1'
>>> chart.chart_type = ChartType.xl3DArea
```

activate()
Activates the object.

New in version 0.5.0.

delete()
Deletes the object.

New in version 0.5.0.

height
Returns or sets a value that represents the height, in points, of the object.

New in version 0.5.0.

left
Returns or sets a value that represents the distance, in points, from the left edge of the object to the left edge of column A.

New in version 0.5.0.

name
Returns or sets a String value representing the name of the object.

New in version 0.5.0.

top
Returns or sets a value that represents the distance, in points, from the top edge of the topmost shape in the shape range to the top edge of the worksheet.

New in version 0.5.0.

width
Returns or sets a value that represents the width, in points, of the object.

New in version 0.5.0.

classmethod add(*sheet=None*, *left=0*, *top=0*, *width=355*, *height=211*, ***kwargs*)
Inserts a new Chart into Excel.

Parameters

- **sheet** (*str or int or xlwings.Sheet, default None*) – Name or index of the Sheet or Sheet object, defaults to the active Sheet
- **left** (*float, default 0*) – left position in points
- **top** (*float, default 0*) – top position in points
- **width** (*float, default 375*) – width in points

- **height** (*float, default 225*) – height in points

Keyword Arguments

- **chart_type** (*xlwings.ChartType member, default xlColumnClustered*) – Excel chart type. E.g. xlwings.ChartType.xlLine
- **name** (*str, default None*) – Excel chart name. Defaults to Excel standard name if not provided, e.g. 'Chart 1'
- **source_data** (Range) – e.g. Range('A1').table
- **wkb** (*Workbook object, default Workbook.current()*) – Defaults to the Workbook that was instantiated last or set via `Workbook.set_current()`.

Returns

Return type xlwings Chart object

chart_type
Gets and sets the chart type of a chart.

New in version 0.1.1.

set_source_data(*source*)
Sets the source for the chart.

Parameters **source** (Range) – Range object, e.g. `Range('A1')`

13.8 Picture

class xlwings.**Picture**(**args*, ***kwargs*)
Bases: xlwings.main.Shape

A Picture object represents an existing Excel Picture and can be instantiated with the following arguments:

```
Picture(1)              Picture('Sheet1', 1)              Picture(1, 1)
Picture('Picture 1')    Picture('Sheet1', 'Picture 1')    Picture(1,
↪'Picture 1')
```

The Sheet can also be provided as Sheet object:

```
sh = Sheet(1)
Shape(sh, 'Picture 1')
```

If no Worksheet is provided as first argument, it will take the Picture from the active Sheet.

Parameters ***args** – Definition of Sheet (optional) and picture in the above described combinations.

Keyword Arguments **wkb** (*Workbook object, default Workbook.current()*) – Defaults to the Workbook that was instantiated last or set via `Workbook.set_current()`.

New in version 0.5.0.

activate()
Activates the object.

New in version 0.5.0.

delete()
Deletes the object.

New in version 0.5.0.

height
Returns or sets a value that represents the height, in points, of the object.

New in version 0.5.0.

left
Returns or sets a value that represents the distance, in points, from the left edge of the object to the left edge of column A.

New in version 0.5.0.

name
Returns or sets a String value representing the name of the object.

New in version 0.5.0.

top
Returns or sets a value that represents the distance, in points, from the top edge of the topmost shape in the shape range to the top edge of the worksheet.

New in version 0.5.0.

width
Returns or sets a value that represents the width, in points, of the object.

New in version 0.5.0.

classmethod add(*filename*, *sheet=None*, *name=None*, *link_to_file=False*, *save_with_document=True*, *left=0*, *top=0*, *width=None*, *height=None*, *wkb=None*)
Inserts a picture into Excel.

Parameters **filename** (*str*) – The full path to the file.

Keyword Arguments

- **sheet** (*str or int or xlwings.Sheet, default None*) – Name or index of the Sheet or `xlwings.Sheet` object, defaults to the active Sheet
- **name** (*str, default None*) – Excel picture name. Defaults to Excel standard name if not provided, e.g. 'Picture 1'

- **left** (*float, default 0*) – Left position in points.
- **top** (*float, default 0*) – Top position in points.
- **width** (*float, default None*) – Width in points. If PIL/Pillow is installed, it defaults to the width of the picture. Otherwise it defaults to 100 points.
- **height** (*float, default None*) – Height in points. If PIL/Pillow is installed, it defaults to the height of the picture. Otherwise it defaults to 100 points.
- **wkb** (*Workbook object, default Workbook.current()*) – Defaults to the Workbook that was instantiated last or set via `Workbook.set_current()`.

Returns

Return type xlwings Picture object

New in version 0.5.0.

update(*filename*)

Replaces an existing picture with a new one, taking over the attributes of the existing picture.

Parameters **filename** (*str*) – Path to the picture.

New in version 0.5.0.

13.9 Plot

class xlwings.**Plot**(*figure*)

Plot allows to easily display Matplotlib figures as pictures in Excel.

Parameters **figure** (*matplotlib.figure.Figure*) – Matplotlib figure

Example

Get a matplotlib `figure` object:

- via PyPlot interface:

```
import matplotlib.pyplot as plt
fig = plt.figure()
plt.plot([1, 2, 3, 4, 5])
```

- via object oriented interface:

```
from matplotlib.figure import Figure
fig = Figure(figsize=(8, 6))
ax = fig.add_subplot(111)
ax.plot([1, 2, 3, 4, 5])
```

- via Pandas:

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd
↪'])
ax = df.plot(kind='bar')
fig = ax.get_figure()
```

Then show it in Excel as picture:

```
plot = Plot(fig)
plot.show('Plot1')
```

New in version 0.5.0.

show(*name*, *sheet=None*, *left=0*, *top=0*, *width=None*, *height=None*, *wkb=None*)

Inserts the matplotlib figure as picture into Excel if a picture with that name doesn't exist yet. Otherwise it replaces the picture, taking over its position and size.

Parameters **name** (*str*) – Name of the picture in Excel

Keyword Arguments

- **sheet** (*str or int or xlwings.Sheet, default None*) – Name or index of the Sheet or `xlwings.Sheet` object, defaults to the active Sheet
- **left** (*float, default 0*) – Left position in points. Only has an effect if the picture doesn't exist yet in Excel.
- **top** (*float, default 0*) – Top position in points. Only has an effect if the picture doesn't exist yet in Excel.
- **width** (*float, default None*) – Width in points, defaults to the width of the matplotlib figure. Only has an effect if the picture doesn't exist yet in Excel.
- **height** (*float, default None*) – Height in points, defaults to the height of the matplotlib figure. Only has an effect if the picture doesn't exist yet in Excel.
- **wkb** (*Workbook object, default Workbook.current()*) – Defaults to the Workbook that was instantiated last or set via `Workbook.set_current()`.

Returns

- *xlwings Picture object*
- *.. versionadded:: 0.5.0*

Index