

---

# **xlwings - Make Excel Fly!**

*Release 0.10.4*

**Zoomer Analytics LLC**

**Jun 23, 2017**



<b>1</b>	<b>Migrate to v0.9</b>	<b>1</b>
1.1	Full qualification: Using collections . . . . .	1
1.2	Connecting to Books . . . . .	1
1.3	Active Objects . . . . .	2
1.4	Round vs. Square Brackets . . . . .	2
1.5	Access the underlying Library/Engine . . . . .	2
1.6	Cheat sheet . . . . .	2
<b>2</b>	<b>Installation</b>	<b>5</b>
2.1	Dependencies . . . . .	5
2.2	Optional Dependencies . . . . .	6
2.3	Python version support . . . . .	6
<b>3</b>	<b>Quickstart</b>	<b>7</b>
3.1	1. Scripting: Automate/interact with Excel from Python . . . . .	7
3.2	2. Macros: Call Python from Excel . . . . .	8
3.3	3. UDFs: User Defined Functions (Windows only) . . . . .	9
<b>4</b>	<b>Connect to a Book</b>	<b>11</b>
4.1	Python to Excel . . . . .	11
4.2	Excel to Python (RunPython) . . . . .	12
4.3	User Defined Functions (UDFs) . . . . .	12
<b>5</b>	<b>Syntax Overview</b>	<b>13</b>
5.1	Active Objects . . . . .	13
5.2	Full qualification . . . . .	14
5.3	Range indexing/slicing . . . . .	14
5.4	Range Shortcuts . . . . .	14
5.5	Object Hierarchy . . . . .	15
<b>6</b>	<b>Data Structures Tutorial</b>	<b>17</b>
6.1	Single Cells . . . . .	17
6.2	Lists . . . . .	18

6.3	Range expanding . . . . .	19
6.4	NumPy arrays . . . . .	19
6.5	Pandas DataFrames . . . . .	19
6.6	Pandas Series . . . . .	20
<b>7</b>	<b>VBA: Calling Python from Excel</b>	<b>21</b>
7.1	xlwings VBA module . . . . .	21
7.2	Settings . . . . .	22
7.3	LOG_FILE default locations . . . . .	22
7.4	Call Python with “RunPython” . . . . .	23
7.5	Function Arguments and Return Values . . . . .	23
<b>8</b>	<b>Debugging</b>	<b>25</b>
8.1	RunPython . . . . .	26
8.2	UDF debug server . . . . .	26
<b>9</b>	<b>Matplotlib</b>	<b>29</b>
9.1	Getting started . . . . .	29
9.2	Full integration with Excel . . . . .	29
9.3	Properties . . . . .	30
9.4	Getting a Matplotlib figure . . . . .	32
<b>10</b>	<b>UDF Tutorial</b>	<b>33</b>
10.1	One-time Excel preparations . . . . .	33
10.2	Workbook preparation . . . . .	33
10.3	A simple UDF . . . . .	34
10.4	Array formulas: Get efficient . . . . .	35
10.5	Array formulas with NumPy and Pandas . . . . .	36
10.6	@xw.arg and @xw.ret decorators . . . . .	37
10.7	Dynamic Array Formulas . . . . .	37
10.8	Docstrings . . . . .	38
10.9	The “vba” keyword . . . . .	38
10.10	Macros . . . . .	38
<b>11</b>	<b>Converters and Options</b>	<b>41</b>
11.1	Default Converter . . . . .	42
11.2	Built-in Converters . . . . .	44
11.3	Custom Converter . . . . .	48
<b>12</b>	<b>Command Line Client</b>	<b>51</b>
12.1	Quickstart . . . . .	51
12.2	Add-in (Currently Windows-only) . . . . .	51
12.3	Template . . . . .	52
12.4	RunPython . . . . .	52
<b>13</b>	<b>Missing Features</b>	<b>53</b>
13.1	Example: Workaround to use VBA’s Range.WrapText . . . . .	53
<b>14</b>	<b>xlwings with R and Julia</b>	<b>55</b>

14.1	R	55
14.2	Julia	56
<b>15</b>	<b>API Documentation</b>	<b>59</b>
15.1	Top-level functions	59
15.2	Object model	60
15.3	UDF decorators	85



The purpose of this document is to enable you a smooth experience when upgrading to xlwings v0.9.0 and above by laying out the concept and syntax changes in detail. If you want to get an overview of the new features and bug fixes, have a look at the release notes. Note that the syntax for User Defined Functions (UDFs) didn't change.

### Full qualification: Using collections

The new object model allows to specify the Excel application instance if needed:

- **old:** `xw.Range('Sheet1', 'A1', wkb=xw.Workbook('Book1'))`
- **new:** `xw.apps[0].books['Book1'].sheets['Sheet1'].range('A1')`

See *Syntax Overview* for the details of the new object model.

### Connecting to Books

- **old:** `xw.Workbook()`
- **new:** `xw.Book()` or via `xw.books` if you need to control the app instance.

See *Connect to a Book* for the details.

## Active Objects

```
# Active app (i.e. Excel instance)
>>> app = xw.apps.active

# Active book
>>> wb = xw.books.active # in active app
>>> wb = app.books.active # in specific app

# Active sheet
>>> sht = xw.sheets.active # in active book
>>> sht = wb.sheets.active # in specific book

# Range on active sheet
>>> xw.Range('A1') # on active sheet of active book of active app
```

## Round vs. Square Brackets

Round brackets follow Excel's behavior (i.e. 1-based indexing), while square brackets use Python's 0-based indexing/slicing.

As an example, the following all reference the same range:

```
xw.apps[0].books[0].sheets[0].range('A1')
xw.apps(1).books(1).sheets(1).range('A1')
xw.apps[0].books['Book1'].sheets['Sheet1'].range('A1')
xw.apps(1).books('Book1').sheets('Sheet1').range('A1')
```

## Access the underlying Library/Engine

- **old:** `xw.Range('A1').xl_range` and `xl_sheet` etc.
- **new:** `xw.Range('A1').api`, same for all other objects

This returns a `pywin32` COM object on Windows and an `appscript` object on Mac.

## Cheat sheet

Note that `sht` stands for a sheet object, like e.g. (in 0.9.0 syntax): `sht = xw.books['Book1'].sheets[0]`

	v0.9.0	v0.7.2
Active Excel instance	<code>xw.apps.active</code>	unsupported
New Excel instance	<code>app = xw.App()</code>	unsupported
Get app from book	<code>app = wb.app</code>	<code>app = xw.Applicat</code>



Table 1.1 – continued from previous page

	v0.9.0	v0.7.2
Target installation (Mac)	<code>app = xw.App(spec=...)</code>	<code>wb = xw.Workbook(...)</code>
Hide Excel Instance	<code>app = xw.App(visible=False)</code>	<code>wb = xw.Workbook(...)</code>
Selected Range	<code>app.selection</code>	<code>wb.get_selection(...)</code>
Calculation mode	<code>app.calculation = 'manual'</code>	<code>app.calculation = ...</code>
All books in app	<code>app.books</code>	<b>unsupported</b>
Fully qualified book	<code>app.books['Book1']</code>	<b>unsupported</b>
Active book in active app	<code>xw.books.active</code>	<code>xw.Workbook.active</code>
New book in active app	<code>wb = xw.Book(...)</code>	<code>wb = xw.Workbook(...)</code>
New book in specific app	<code>wb = app.books.add(...)</code>	<b>unsupported</b>
All sheets in book	<code>wb.sheets</code>	<code>xw.Sheet.all(wb)</code>
Call a macro in an addin	<code>app.macro('MacroName')</code>	<b>unsupported</b>
First sheet of book wb	<code>wb.sheets[0]</code>	<code>xw.Sheet(1, wkb=w)</code>
Active sheet	<code>wb.sheets.active</code>	<code>xw.Sheet.active(w)</code>
Add sheet	<code>wb.sheets.add(...)</code>	<code>xw.Sheet.add(wkb=...)</code>
Sheet count	<code>wb.sheets.count</code> or <code>len(wb.sheets)</code>	<code>xw.Sheet.count(wb)</code>
Add chart to sheet	<code>chart = wb.sheets[0].charts.add(...)</code>	<code>chart = xw.Chart(...)</code>
Existing chart	<code>wb.sheets['Sheet 1'].charts[0]</code>	<code>xw.Chart('Sheet 1')</code>
Chart Type	<code>chart.chart_type = '3d_area'</code>	<code>chart.chart_type = ...</code>
Add picture to sheet	<code>wb.sheets[0].pictures.add('path/to/pic')</code>	<code>xw.Picture.add('p')</code>
Existing picture	<code>wb.sheets['Sheet 1'].pictures[0]</code>	<code>xw.Picture('Sheet 1')</code>
Matplotlib	<code>sht.pictures.add(fig, name='x', update=True)</code>	<code>xw.Plot(fig).show</code>
Table expansion	<code>sht.range('A1').expand('table')</code>	<code>xw.Range(sht, 'A1')</code>
Vertical expansion	<code>sht.range('A1').expand('down')</code>	<code>xw.Range(sht, 'A1')</code>
Horizontal expansion	<code>sht.range('A1').expand('right')</code>	<code>xw.Range(sht, 'A1')</code>
Set name of range	<code>sht.range('A1').name = 'name'</code>	<code>xw.Range(sht, 'A1')</code>
Get name of range	<code>sht.range('A1').name.name</code>	<code>xw.Range(sht, 'A1')</code>
mock caller	<code>xw.Book('file.xlsm').set_mock_caller(...)</code>	<code>xw.Workbook.set_m...</code>



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`.

---

## 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](#) which can be a hassle.

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

On Mac, the dependencies are automatically being handled if xlwings is installed with `conda` or `pip`. However, with `pip`, 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](#).

### Optional Dependencies

- NumPy
- Pandas
- Matplotlib
- Pillow/PIL

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

### Python version support

xlwings is tested on Python 2.7 and 3.3+

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

## 1. Scripting: Automate/interact with Excel from Python

Establish a connection to a workbook:

```
>>> import xlwings as xw
>>> wb = xw.Book() # this will create a new workbook
>>> wb = xw.Book('FileName.xlsx') # connect to an existing file in the
↳current working directory
>>> wb = xw.Book(r'C:\path\to\file.xlsx') # on Windows: use raw strings to
↳escape backslashes
```

If you have the same file open in two instances of Excel, you need to fully qualify it and include the app instance:

```
>>> xw.apps[0].books['FileName.xlsx']
```

Instantiate a sheet object:

```
>>> sht = wb.sheets['Sheet1']
```

Reading/writing values to/from ranges is as easy as:

```
>>> sht.range('A1').value = 'Foo 1'
>>> sht.range('A1').value
'Foo 1'
```

There are many **convenience features** available, e.g. Range expanding:

```
>>> sht.range('A1').value = [['Foo 1', 'Foo 2', 'Foo 3'], [10.0, 20.0, 30.0]]
>>> sht.range('A1').expand().value
[['Foo 1', 'Foo 2', 'Foo 3'], [10.0, 20.0, 30.0]]
```

**Powerful converters** handle most data types of interest, including Numpy arrays and Pandas DataFrames in both directions:

```
>>> import pandas as pd
>>> df = pd.DataFrame([[1,2], [3,4]], columns=['a', 'b'])
>>> sht.range('A1').value = df
>>> sht.range('A1').options(pd.DataFrame, expand='table').value
      a  b
0.0  1.0  2.0
1.0  3.0  4.0
```

**Matplotlib** figures can be shown as pictures in Excel:

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.plot([1, 2, 3, 4, 5])
[<matplotlib.lines.Line2D at 0x1071706a0>]
>>> sht.pictures.add(fig, name='MyPlot', update=True)
<Picture 'MyPlot' in <Sheet [Workbook4]Sheet1>>
```

**Shortcut** for the active sheet: `xw.Range`

If you want to quickly talk to the active sheet in the active workbook, you don't need instantiate a workbook and sheet object, but can simply do:

```
>>> import xlwings as xw
>>> xw.Range('A1').value = 'Foo'
>>> xw.Range('A1').value
'Foo'
```

**Note:** You should only use `xw.Range` when interacting with Excel. In scripts, you should always go via book and sheet objects as shown above.

## 2. Macros: Call Python from Excel

You can call Python functions from VBA using the `RunPython` function:

```
Sub HelloWorld()
    RunPython ("import hello; hello.world()")
End Sub
```

Per default, `RunPython` expects `hello.py` in the same directory as the Excel file. Refer to the calling Excel book by using `xw.Book.caller`:

```
# hello.py
import numpy as np
```

```
import xlwings as xw

def world():
    wb = xw.Book.caller()
    wb.sheets[0].range('A1').value = 'Hello World!'
```

To make this run, you'll need to have the xlwings VBA module in your Excel book. The easiest way to get everything set up is to use the xlwings command line client from either a command prompt on Windows or a terminal on Mac: `xlwings quickstart myproject`.

To import the xlwings VBA module differently, and for more details, see *VBA: Calling Python from Excel*.

### 3. UDFs: User Defined Functions (Windows only)

Writing a UDF in Python is as easy as:

```
import xlwings as xw

@xw.func
def hello(name):
    return 'Hello {}'.format(name)
```

Converters can be used with UDFs, too. Again a Pandas DataFrame example:

```
import xlwings as xw
import pandas as pd

@xw.func
@xw.arg('x', pd.DataFrame)
def correl2(x):
    # x arrives as DataFrame
    return x.corr()
```

Import this function into Excel by clicking the import button of the xlwings add-in: For further details, see *UDF Tutorial*.





---

## Connect to a Book

---

When reading/writing data to the active sheet, you don't need a book object:

```
>>> import xlwings as xw
>>> xw.Range('A1').value = 'something'
```

## Python to Excel

The easiest way to connect to a book is offered by `xw.Book`: it looks for the book in all app instances and returns an error, should the same book be open in multiple instances. To connect to a book in the active app instance, use `xw.books` and to refer to a specific app, use:

```
>>> app = xw.App() # or something like xw.apps[0] for existing apps
>>> app.books['Book1']
```

	<code>xw.Book</code>	<code>xw.books</code>
New book	<code>xw.Book()</code>	<code>xw.books.add()</code>
Unsaved book	<code>xw.Book('Book1')</code>	<code>xw.books['Book1']</code>
Book by (full)name	<code>xw.Book(r'C:/path/to/file.xlsx')</code>	<code>xw.books.open(r'C:/path/to/file.xlsx')</code>

**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`.

---

## Excel to Python (RunPython)

To reference the calling book when using `RunPython` in VBA, use `xw.Book.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 `xw.Book.caller()` and one of the methods explained above.

## User Defined Functions (UDFs)

Unlike `RunPython`, UDFs don't need a call to `xw.Book.caller()`, see *UDF Tutorial*. However, it's available (restricted to read-only though), which sometimes proves to be useful.

The xlwings object model is very similar to the one used by VBA.

All code samples below depend on the following import:

```
>>> import xlwings as xw
```

### Active Objects

```
# Active app (i.e. Excel instance)
>>> app = xw.apps.active

# Active book
>>> wb = xw.books.active # in active app
>>> wb = app.books.active # in specific app

# Active sheet
>>> sht = xw.sheets.active # in active book
>>> sht = wb.sheets.active # in specific book

# Range on active sheet
>>> xw.Range('A1') # on active sheet of active book of active app
```

A Range can be instantiated with A1 notation, a tuple of Excel's 1-based indices, a named range or two Range objects:

```
xw.Range('A1')
xw.Range('A1:C3')
xw.Range((1,1))
xw.Range((1,1), (3,3))
```

```
xw.Range('NamedRange')
xw.Range(xw.Range('A1'), xw.Range('B2'))
```

### Full qualification

Round brackets follow Excel's behavior (i.e. 1-based indexing), while square brackets use Python's 0-based indexing/slicing. As an example, the following expressions all reference the same range:

```
xw.apps[0].books[0].sheets[0].range('A1')
xw.apps(1).books(1).sheets(1).range('A1')
xw.apps[0].books['Book1'].sheets['Sheet1'].range('A1')
xw.apps(1).books('Book1').sheets('Sheet1').range('A1')
```

### Range indexing/slicing

Range objects support indexing and slicing, a few examples:

```
>>> rng = xw.Book().sheets[0].range('A1:D5')
>>> rng[0, 0]
<Range [Workbook1] Sheet1!$A$1>
>>> rng[1]
<Range [Workbook1] Sheet1!$B$1>
>>> rng[:, 3:]
<Range [Workbook1] Sheet1!$D$1:$D$5>
>>> rng[1:3, 1:3]
<Range [Workbook1] Sheet1!$B$2:$C$3>
```

### Range Shortcuts

Sheet objects offer a shortcut for range objects by using index/slice notation on the sheet object. This evaluates to either `sheet.range` or `sheet.cells` depending on whether you pass a string or indices/slices:

```
>>> sht = xw.Book().sheets['Sheet1']
>>> sht['A1']
<Range [Book1] Sheet1!$A$1>
>>> sht['A1:B5']
<Range [Book1] Sheet1!$A$1:$B$5>
>>> sht[0, 1]
<Range [Book1] Sheet1!$B$1>
>>> sht[:10, :10]
<Range [Book1] Sheet1!$A$1:$J$10>
```

## Object Hierarchy

The following shows an example of the object hierarchy, i.e. how to get from an app to a range object and all the way back:

```
>>> rng = xw.apps[0].books[0].sheets[0].range('A1')
>>> rng.sheet.book.app
<Excel App 1644>
```



---

## Data Structures Tutorial

---

This tutorial 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 the behavior using the `options` method, have a look at *Converters and Options*.

All code samples below depend on the following import:

```
>>> import xlwings as xw
```

### 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:

```
>>> import datetime as dt
>>> sht = xw.Book().sheets[0]
>>> sht.range('A1').value = 1
>>> sht.range('A1').value
1.0
>>> sht.range('A2').value = 'Hello'
>>> sht.range('A2').value
'Hello'
>>> sht.range('A3').value is None
True
>>> sht.range('A4').value = dt.datetime(2000, 1, 1)
>>> sht.range('A4').value
datetime.datetime(2000, 1, 1, 0, 0)
```

## 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, the next point shows you how to preserve this info:

```
>>> sht = xw.Book().sheets[0]
>>> sht.range('A1').value = [[1],[2],[3],[4],[5]] # Column orientation
↳ (nested list)
>>> sht.range('A1:A5').value
[1.0, 2.0, 3.0, 4.0, 5.0]
>>> sht.range('A1').value = [1, 2, 3, 4, 5]
>>> sht.range('A1:E1').value
[1.0, 2.0, 3.0, 4.0, 5.0]
```

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

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

---

**Note:** To write a list in column orientation to Excel, use transpose: `sht.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”):

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

```
>>> sht.range('A10').value = [['Foo 1', 'Foo 2', 'Foo 3'], [10, 20, 30]]
>>> sht.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 `sht.range('A1').value = [[1,2],[3,4]]` than `sht.range('A1').value = [1, 2]` and `sht.range('A2').value = [3, 4]`.

---



## Range expanding

You can get the dimensions of Excel Ranges dynamically through either the method `expand` or through the `expand` keyword in the `options` method. While `expand` gives back an expanded Range object, `options` are only evaluated when accessing the values of a Range. The difference is best explained with an example:

```
>>> sht = xw.Book().sheets[0]
>>> sht.range('A1').value = [[1,2], [3,4]]
>>> rng1 = sht.range('A1').expand('table') # or just .expand()
>>> rng2 = sht.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]]
>>> sht.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]]
```

'table' expands to 'down' and 'right', the other available options which can be used for column or row only expansion, respectively.

**Note:** Using `expand()` 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 `sht.range('NamedRange').expand().value`.

## 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` in the `options` method:

```
>>> import numpy as np
>>> sht = xw.Book().sheets[0]
>>> sht.range('A1').value = np.eye(3)
>>> sht.range('A1').options(np.array, expand='table').value
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

## Pandas DataFrames

```
>>> sht = xw.Book().sheets[0]
>>> df = pd.DataFrame([[1.1, 2.2], [3.3, None]], columns=['one', 'two'])
>>> df
```

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

## Pandas Series

```
>>> import pandas as pd
>>> import numpy as np
>>> sht = xw.Book().sheets[0]
>>> 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
>>> sht.range('A1').value = s
>>> sht.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, a NumPy array or a Pandas DataFrame to Excel, e.g.: `sht.range('A1').value = np.eye(10)`

---

---

## VBA: Calling Python from Excel

---

### 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 book.

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__
```

### 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 = ""
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: uses a COM Server. This will be faster, as the interpreter doesn't shut down after each call

### 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"`.

---

## Call Python with “RunPython”

After adding the xlwings VBA module to your Excel file, 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 HelloWorld()  
    RunPython ("import hello; hello.world()")  
End Sub
```

This calls the following code in `hello.py`:

```
# hello.py  
import numpy as np  
import xlwings as xw  
  
def world():  
    wb = xw.Book.caller()  
    wb.sheets[0].range('A1').value = 'Hello World!'
```

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

---

**Note:** Place `xw.Book.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`.

---

## Function Arguments and Return Values

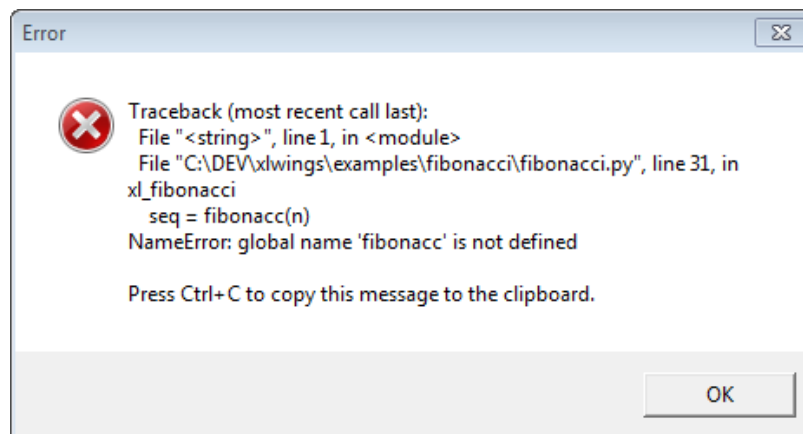
While it's technically possible to include arguments in the function call within `RunPython`, it's not very convenient. To do that easily and to also be able to return values from Python, use UDFs, see [UDF Tutorial](#) - however, this is currently limited to Windows only.



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 the function 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*.

## RunPython

Consider the following sample code of your Python source code `my_module.py`:

```
# my_module.py
import os
import xlwings as xw

def my_macro():
    wb = xw.Book.caller()
    wb.sheets[0].range('A1').value = 1

if __name__ == '__main__':
    # Expects the Excel file next to this source file, adjust accordingly.
    xw.Book('myfile.xlsx').set_mock_caller()
    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
```

## 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 need to run the code in “debug” mode (e.g. in case you’re 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.

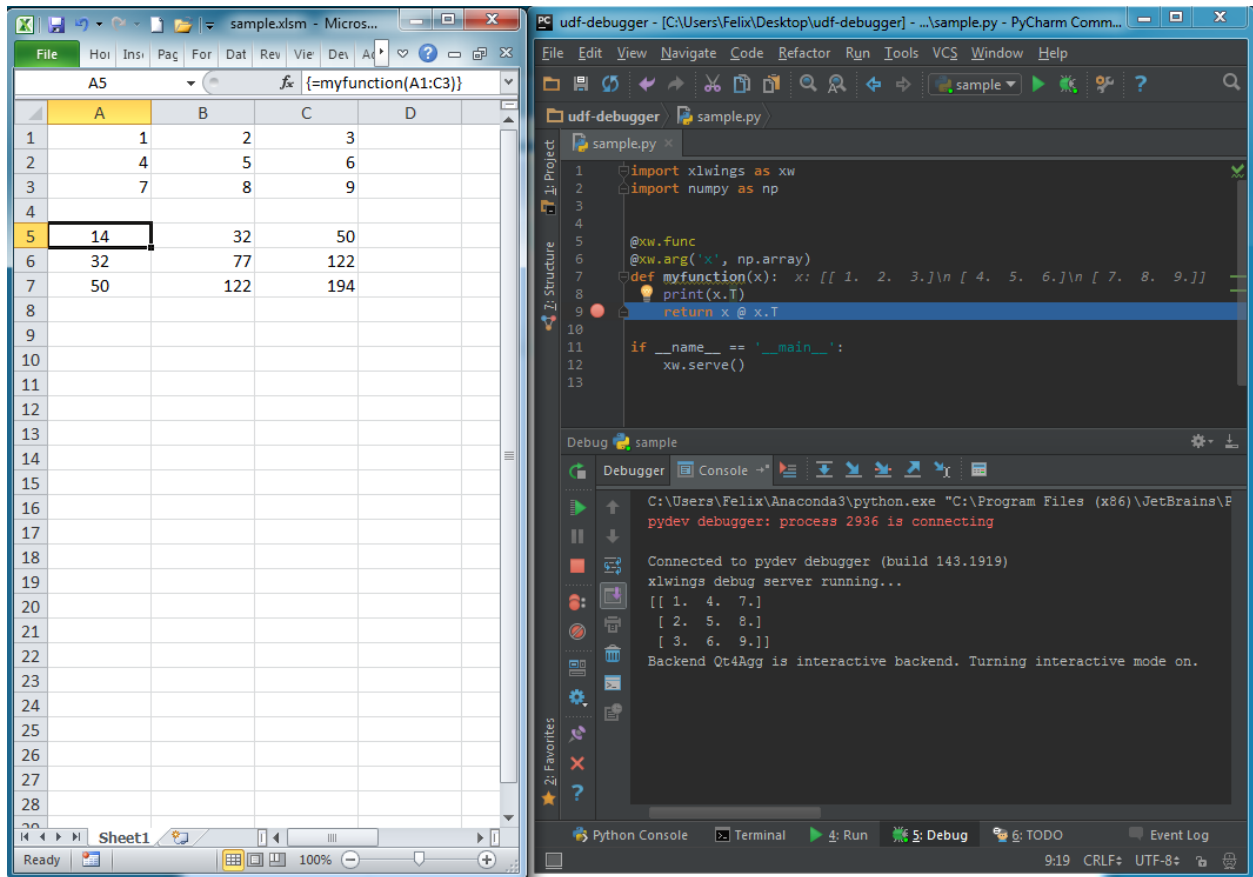
The following screenshot shows the code stopped at a breakpoint in the community version of PyCharm:

---

**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.

---







Using `pictures.add()`, it is easy to paste a Matplotlib plot as picture in Excel.

## 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])

>>> sht = xw.Book().sheets[0]
>>> sht.pictures.add(fig, name='MyPlot', update=True)
```

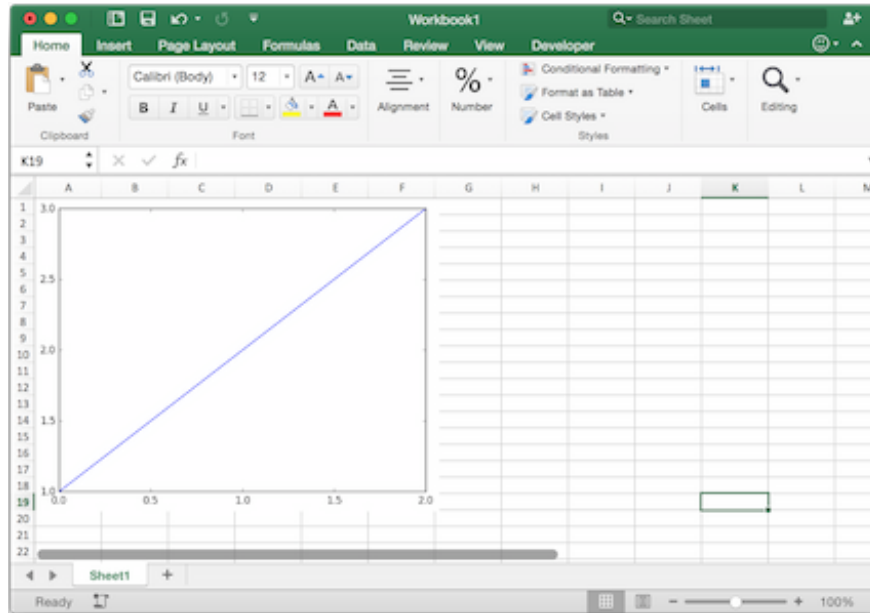
---

**Note:** If you set `update=True`, you can resize and position the plot on Excel: subsequent calls to `pictures.add()` with the same name ('MyPlot') will update the picture without changing its position or size.

---

## 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):
    sht = xw.Book.caller().sheets.active
    fig = plt.figure()
    plt.plot(range(int(n)))
    sht.pictures.add(fig, name='MyPlot', update=True)
    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:

## Properties

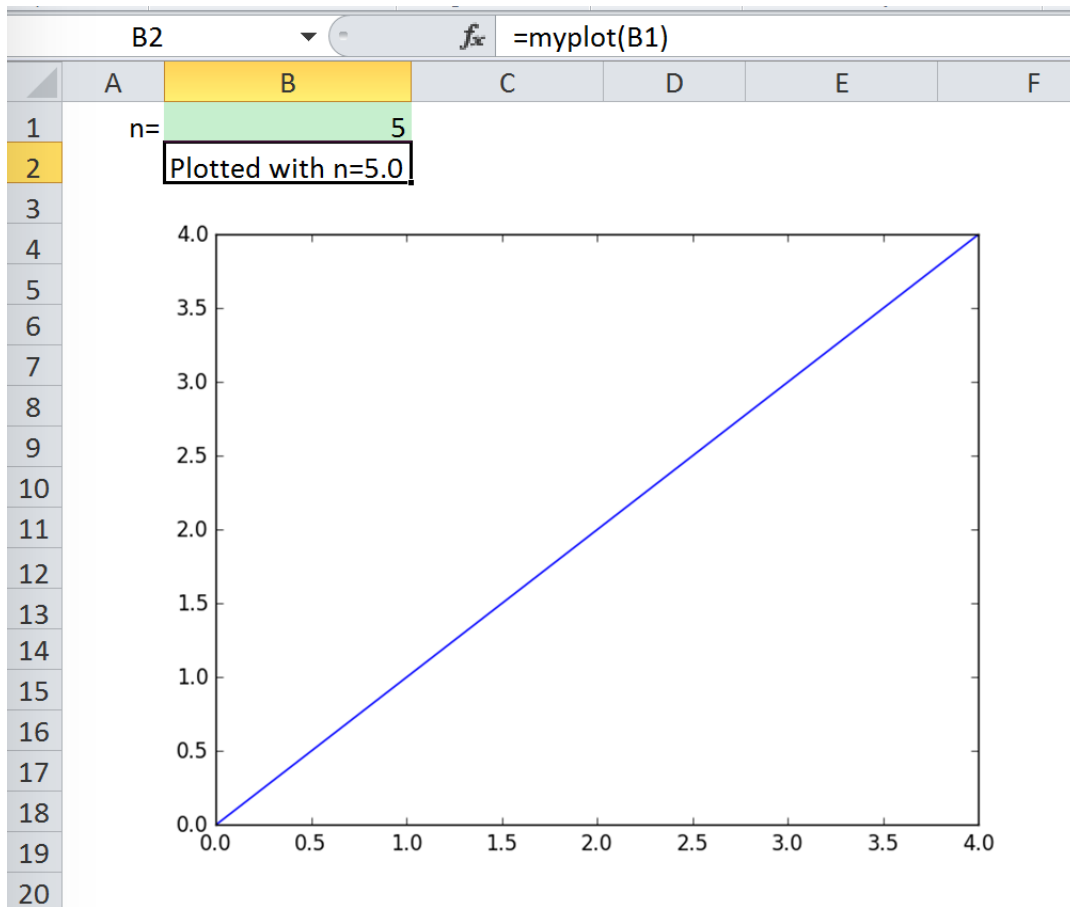
Size, position and other properties can either be set as arguments within `pictures.add()`, or by manipulating the picture object that is returned, see `xlwings.Picture()`.

For example:

```
>>> sht = xw.Book().sheets[0]
>>> sht.pictures.add(fig, name='MyPlot', update=True,
                    left=sht.range('B5').left, top=sht.range('B5').top)
```

or:

```
>>> plot = sht.pictures.add(fig, name='MyPlot', update=True)
>>> plot.height /= 2
>>> plot.width /= 2
```



## 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()
```

This tutorial gets you quickly started on how to write User Defined Functions.

---

**Note:**

- UDFs are currently only available on Windows.
  - For details of how to control the behaviour of the arguments and return values, have a look at *Converters and Options*.
  - For a comprehensive overview of the available decorators and their options, check out the corresponding API docs: *UDF decorators*.
- 

## 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.

## 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*

### 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`.

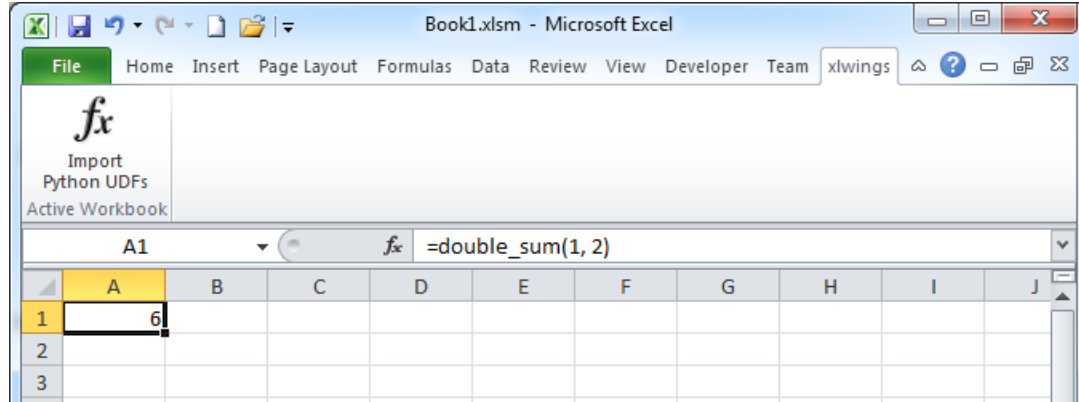
Alternatively, you can point to a specific source file by setting the `UDF_PATH` in the VBA settings.

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.

## 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.

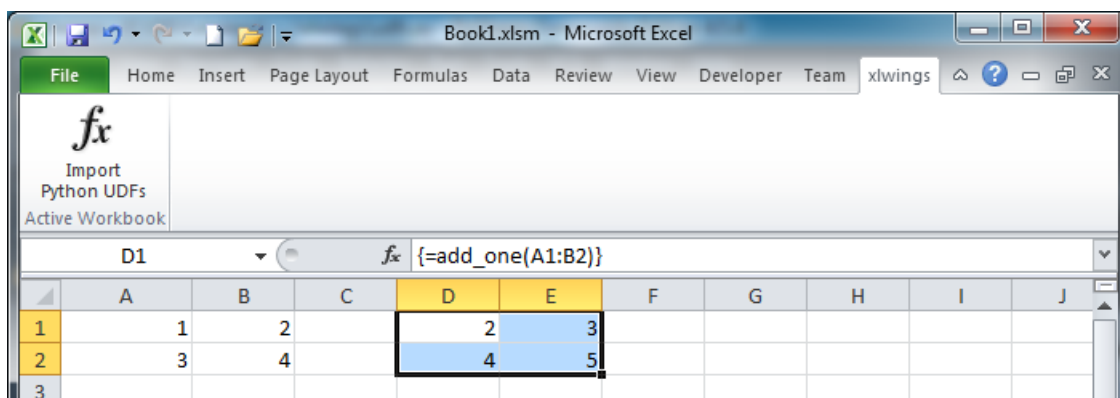
You can pass an Excel Range as a function argument, as opposed to a single cell and it will show up in Python as list of lists.

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:



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

### 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()
```

## @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.

## Dynamic Array Formulas

As seen above, to use Excel's array formulas, you need to specify their dimensions up front by selecting the result array first, then entering the formula and finally hitting `Ctrl-Shift-Enter`. While this makes sense from a data integrity point of view, in practice, it often turns out to be a cumbersome limitation, especially when working with dynamic arrays such as time series data. Since v0.10, xlwings offers dynamic UDF expansion:

This is a simple example that demonstrates the syntax and effect of UDF expansion:

```
import numpy as np

@xw.func
@xw.ret(expand='table')
def dynamic_array(r, c):
    return np.random.randn(int(r), int(c))
```

	A	B	C	D	E
1		rows:	columns:		
2		5	2		
3					
4		2.01156647	-0.0985618		
5		-0.2152179	-0.7541961		
6		0.37168657	-0.1978662		
7		-1.0643897	1.37592295		
8		0.5272535	-0.0508628		
9					

### Note:

- Expanding array formulas will overwrite cells without prompting and leave an empty border around them, i.e. they will clear the row to the bottom and the column to the right of the array.

	A	B	C	D	E	F
1		rows:	columns:			
2		2	5			
3						
4		-0.6788379	-1.0009999	-0.6342434	-0.9362773	1.02582914
5		-2.1803953	0.18511092	0.3121721	0.20600051	0.3799863
6						

- The way that dynamic array formulas are currently implemented doesn't allow them to have volatile functions as arguments, e.g. you cannot use functions like =TODAY () as arguments.

## Docstrings

The following sample shows how to include docstrings both for the function and for the arguments x and y that then show up in the function wizard in Excel:

```
import xlwings as xw

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

## 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
```

## 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.Book.caller()  
wb.sheets[0].range('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 do 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.



---

## 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 explicitly set in the `options` method when manipulating `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.

All code samples below depend on the following import:

```
>>> import xlwings as xw
```

### Syntax:

	<b>xw.Range</b>	<b>UDFs</b>
<b>read- ing</b>	<code>xw.Range.options(convert=None, **kwargs).value</code>	<code>@arg('x', convert=None, **kwargs)</code>
<b>writ- ing</b>	<code>xw.Range.options(convert=None, **kwargs).value = myvalue</code>	<code>@ret(convert=None, **kwargs)</code>

**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:

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

## 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
>>> sht = xw.Book().sheets[0]
>>> sht.range('A1').value = [[1, 2], [3, 4]]
>>> sht.range('A1').value
1.0
>>> sht.range('A1').options(ndim=1).value
[1.0]
>>> sht.range('A1').options(ndim=2).value
[[1.0]]
>>> sht.range('A1:A2').value
[1.0 3.0]
>>> sht.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`:

```
>>> sht.range('A1').value = 1
>>> sht.range('A1').value
1.0
>>> sht.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, which is the reason why the `int` converter rounds numbers first before turning them into integers. Otherwise it could happen that e.g. 5 might be



returned as 4 in case it is represented as a floating point number that is slightly smaller than 5. Should you require Python's original *int* in your converter, use *raw int* instead.

- **dates**

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

- Range:

```
>>> import datetime as dt
>>> sht.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)
>>> sht.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: `>>> sht.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: `sht.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
>>> sht = xw.Book().sheets[0]
>>> sht.range('A1').value = [[1,2], [3,4]]
>>> rng1 = sht.range('A1').expand()
>>> rng2 = sht.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]]
>>> sht.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` method is only available on `Range` objects as UDFs only allow to manipulate the calling cells.

---

## 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.

### 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
6		2

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

## 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
>>> sht = xw.Book().sheets[0]
>>> sht.range('A1').options(transpose=True).value = np.array([1, 2, 3])
>>> sht.range('A1:A3').options(np.array, ndim=2).value
array([[ 1.],
       [ 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			

```
>>> sht = xw.Book().sheets[0]
>>> s = sht.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
2001-01-06    6
Name: series name, dtype: float64
>>> sht.range('D1', header=False).value = s
```

## 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:

	A	B	C	D
1		a	a	b
2	ix	c	d	e
3	10	1	2	3
4	20	4	5	6
5	30	7	8	9
6				
7		a	a	b
8		c	d	e
9		1	2	3
10		4	5	6
11		7	8	9
12				
13		a	a	b
14		c	d	e
15		1	2	3
16		4	5	6
17		7	8	9
18				

```

>>> sht = xw.Book().sheets[0]
>>> df = sht.range('A1:D5').options(pd.DataFrame, header=2).value
>>> df
   a    b
   c  d  e
ix
10  1  2  3
20  4  5  6
30  7  8  9

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

# Writing back and changing some of the options, e.g. getting rid of the
↳index:
>>> sht.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

```

## 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:

```

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

>>> sht.range('A1:B2').options('raw').value # or sht.range('A1:B2').raw_
↳value
((1.0, 'text'), (pywintypes.datetime(2016, 2, 1, 0, 0,
↳tzinfo=TimeZoneInfo('GMT Standard Time', True)), None))

```

## 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 `xw.Range.options` method, e.g. when calling `xw.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: `DictConverter`, `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
    # 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
sht = xw.Book().sheets[0]
df = pd.DataFrame([[1., 10.], [2., np.nan], [3., 30.]])

```

- Default converter for DataFrames:

```

# Write
sht.range('A1').value = df

# Read
sht.range('A1:C4').options(pd.DataFrame).value

```

- DataFrameDropna converter:

```

# Write
sht.range('A7').options(DataFrameDropna, dropna=True).value = df

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

```

- Register an alias (optional):

```

DataFrameDropna.register('df_dropna')

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

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

```

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

```
DataFrameDropna.register(pd.DataFrame)

# Write
sht.range('A13').options(dropna=True).value = df

# Read
sht.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.

---



---

## 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`.

### 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.

### 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.

## 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.

## 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.

---

## 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 calling the `api` property:

```
>>> sht = xw.Book().sheets[0]
>>> sht.api
<COMObject <unknown>> # Windows/pywin32
app(pid=2319).workbooks['Workbook1'].worksheets[1] # Mac/appscript
```

This works accordingly for the other objects like `sht.range('A1').api` 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 its way into the library (cross-platform and with a Pythonic syntax).

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

```
# Windows
sht.range('A1').api.WrapText = True
```

```
# Mac  
sht.range('A1').api.wrap_text.set(True)
```

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](#).

### 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\`
- Add `C:\Program Files\R\R-x.x.x\bin` to `PATH`
- Restart Windows because of the environment variables (!)

### 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.

## 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.

## 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.Book()
PyObject <Book [Workbook1]>

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





## Top-level functions

`xlwings.view(obj, sheet=None)`

Opens a new workbook and displays an object on its first sheet by default. If you provide a sheet object, it will clear the sheet before displaying the object on the existing sheet.

### Parameters

- **obj** (*any type with built-in converter*) – the object to display, e.g. numbers, strings, lists, numpy arrays, pandas dataframes
- **sheet** (*Sheet, default None*) – Sheet object. If none provided, the first sheet of a new workbook is used.

### Examples

```
>>> 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.

## Object model

### Apps

**class** `xlwings.main.Apps` (*impl*)

A collection of all *app* objects:

```
>>> import xlwings as xw
>>> xw.apps
Apps ([<Excel App 1668>, <Excel App 1644>])
```

**active**

Returns the active app.

New in version 0.9.0.

**add()**

Creates a new App. The new App becomes the active one. Returns an App object.

**count**

Returns the number of apps.

New in version 0.9.0.

### App

**class** `xlwings.App` (*visible=None, spec=None, add\_book=True, impl=None*)

An app corresponds to an Excel instance. New Excel instances can be fired up like so:

```
>>> import xlwings as xw
>>> app1 = xw.App()
>>> app2 = xw.App()
```

An app object is a member of the *apps* collection:

```
>>> xw.apps
Apps ([<Excel App 1668>, <Excel App 1644>])
>>> xw.apps[0]
<Excel App 1668>
>>> xw.apps.active
<Excel App 1668>
```

#### Parameters

- **visible** (*bool, default None*) – Returns or sets a boolean value that determines whether the app is visible. The default leaves the state unchanged or sets `visible=True` if the object doesn't exist yet.
- **spec** (*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.

---

**Note:** On Mac, while xlwings allows you to run multiple instances of Excel, it's a feature that is not officially supported by Excel for Mac: Unlike on Windows, Excel will not ask you to open a read-only version of a file if it is already open in another instance. This means that you need to watch out yourself so that the same file is not being overwritten from different instances.

---

**activate** (*steal\_focus=False*)

Activates the Excel app.

**Parameters** **steal\_focus** (*bool, default False*) – If True, make front-most application and hand over focus from Python to Excel.

New in version 0.9.0.

**api**

Returns the native object (`pywin32` or `appscript` obj) of the engine being used.

New in version 0.9.0.

**books**

A collection of all Book objects that are currently open.

New in version 0.9.0.

**calculate** ()

Calculates all open books.

New in version 0.3.6.

**calculation**

Returns or sets a calculation value that represents the calculation mode. Modes: 'manual', 'automatic', 'semiautomatic'

## Examples

```
>>> import xlwings as xw
>>> wb = xw.Book()
>>> wb.app.calculation = 'manual'
```

Changed in version 0.9.0.

**display\_alerts**

The default value is True. Set this property to False to suppress prompts and alert messages while code is running; when a message requires a response, Excel chooses the default response.

New in version 0.9.0.

### **hwnd**

Returns the Window handle (Windows-only).

New in version 0.9.0.

### **kill ()**

Forces the Excel app to quit by killing its process.

New in version 0.9.0.

### **macro (name)**

Runs a Sub or Function in Excel VBA that are not part of a specific workbook but e.g. are part of an add-in.

**Parameters name** (Name of Sub or Function with or without module name, e.g. 'Module1.MyMacro' or 'MyMacro') –

## Examples

This VBA function:

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

can be accessed like this:

```
>>> import xlwings as xw
>>> app = xw.App()
>>> my_sum = app.macro('MySum')
>>> my_sum(1, 2)
3
```

See also: *Book.macro()*

New in version 0.9.0.

### **pid**

Returns the PID of the app.

New in version 0.9.0.

### **quit ()**

Quits the application without saving any workbooks.

New in version 0.3.3.

### **range (cell1, cell2=None)**

Range object from the active sheet of the active book, see *Range()*.

New in version 0.9.0.

### **screen\_updating**

Turn screen updating off to speed up your script. You won't be able to see what the script is

doing, but it will run faster. Remember to set the `screen_updating` property back to `True` when your script ends.

New in version 0.3.3.

### **selection**

Returns the selected cells as `Range`.

New in version 0.9.0.

### **version**

Returns the Excel version number object.

## **Examples**

```
>>> import xlwings as xw
>>> xw.App().version
VersionNumber('15.24')
>>> xw.apps[0].version.major
15
```

Changed in version 0.9.0.

### **visible**

Gets or sets the visibility of Excel to `True` or `False`.

New in version 0.3.3.

## **Books**

**class** `xlwings.main.Books` (*impl*)

A collection of all *book* objects:

```
>>> import xlwings as xw
>>> xw.books # active app
Books([<Book [Book1]>, <Book [Book2]>])
>>> xw.apps[0].books # specific app
Books([<Book [Book1]>, <Book [Book2]>])
```

New in version 0.9.0.

### **active**

Returns the active `Book`.

### **add()**

Creates a new `Book`. The new `Book` becomes the active `Book`. Returns a `Book` object.

### **open** (*fullname*)

Opens a `Book` if it is not open yet and returns it. If it is already open, it doesn't raise an exception but simply returns the `Book` object.

**Parameters fullname** (*str*) – filename or fully qualified filename, e.g. `r'C:\path\to\file.xlsx'` or `'file.xlsxm'`. Without a full path, it looks for the file in the current working directory.

**Returns Book**

**Return type** Book that has been opened.

## Book

**class xlwings.Book** (*fullname=None, impl=None*)

A book object is a member of the `books` collection:

```
>>> import xlwings as xw
>>> xw.books[0]
<Book [Book1]>
```

The easiest way to connect to a book is offered by `xw.Book`: it looks for the book in all app instances and returns an error, should the same book be open in multiple instances. To connect to a book in the active app instance, use `xw.books` and to refer to a specific app, use:

```
>>> app = xw.App() # or something like xw.apps[0] for existing apps
>>> app.books['Book1']
```

	<code>xw.Book</code>	<code>xw.books</code>
New book	<code>xw.Book()</code>	<code>xw.books.add()</code>
Unsaved book	<code>xw.Book('Book1')</code>	<code>xw.books['Book1']</code>
Book by (fullname)	<code>xw.Book(r'C:/path/to/file.xlsx')</code>	<code>xw.books.open(r'C:/path/to/file.xlsx')</code>

**Parameters fullname** (*str, default None*) – Full path or name (incl. `xlsx`, `xlsm` etc.) of existing workbook or name of an unsaved workbook. Without a full path, it looks for the file in the current working directory.

**activate** (*steal\_focus=False*)

Activates the book.

**Parameters steal\_focus** (*bool, default False*) – If True, make front-most window and hand over focus from Python to Excel.

**api**

Returns the native object (`pywin32` or `appscript` obj) of the engine being used.

New in version 0.9.0.

**app**

Returns an app object that represents the creator of the book.

New in version 0.9.0.

**classmethod caller ()**

References the calling book when the Python function is called from Excel via `RunPython`. Pack it into the function being called from Excel, e.g.:

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

New in version 0.3.0.

**close ()**

Closes the book without saving it.

New in version 0.1.1.

**fullname**

Returns the name of the object, including its path on disk, as a string. Read-only String.

**macro (name)**

Runs a Sub or Function in Excel VBA.

**Parameters name** (Name of Sub or Function with or without module name, e.g. 'Module1.MyMacro' or 'MyMacro') –

**Examples**

This VBA function:

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

can be accessed like this:

```
>>> import xlwings as xw
>>> wb = xw.books.active
>>> my_sum = wb.macro('MySum')
>>> my_sum(1, 2)
3
```

See also: [App.macro\(\)](#)

New in version 0.7.1.

**name**

Returns the name of the book as str.

**names**

Returns a names collection that represents all the names in the specified book (including all sheet-specific names).

Changed in version 0.9.0.

### **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:

```
>>> xw.Book.open_template ()
```

See also: *Command Line Client*

New in version 0.3.3.

### **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**

```
>>> import xlwings as xw
>>> wb = xw.Book ()
>>> wb.save ()
>>> wb.save (r'C:\path\to\new_file_name.xlsx')
```

New in version 0.3.1.

### **selection**

Returns the selected cells as Range.

New in version 0.9.0.

### **set\_mock\_caller ()**

Sets the Excel file which is used to mock `xw.Book.caller ()` when the code is called from Python and not from Excel via RunPython.

### **Examples**

```
# This code runs unchanged from Excel via RunPython and from Python_
↳ directly
import os
import xlwings as xw

def my_macro ():
    sht = xw.Book.caller ().sheets [0]
    sht.range ('A1').value = 'Hello xlwings!'

if __name__ == '__main__':
    xw.Book ('file.xlsm').set_mock_caller ()
    my_macro ()
```

New in version 0.3.1.



**sheets**

Returns a sheets collection that represents all the sheets in the book.

New in version 0.9.0.

**Sheets**

**class** xlwings.main.**Sheets** (*impl*)

A collection of all *sheet* objects:

```
>>> import xlwings as xw
>>> xw.sheets # active book
Sheets ([<Sheet [Book1]Sheet1>, <Sheet [Book1]Sheet2>])
>>> xw.apps[0].books['Book1'].sheets # specific book
Sheets ([<Sheet [Book1]Sheet1>, <Sheet [Book1]Sheet2>])
```

New in version 0.9.0.

**active**

Returns the active Sheet.

**add** (*name=None, before=None, after=None*)

Creates a new Sheet and makes it the active sheet.

**Parameters**

- **name** (*str, default None*) – Name of the new sheet. If None, will default to Excel's default name.
- **before** (*Sheet, default None*) – An object that specifies the sheet before which the new sheet is added.
- **after** (*Sheet, default None*) – An object that specifies the sheet after which the new sheet is added.

**Sheet**

**class** xlwings.**Sheet** (*sheet=None, impl=None*)

A sheet object is a member of the *sheets* collection:

```
>>> import xlwings as xw
>>> wb = xw.Book()
>>> wb.sheets[0]
<Sheet [Book1]Sheet1>
>>> wb.sheets['Sheet1']
<Sheet [Book1]Sheet1>
>>> wb.sheets.add()
<Sheet [Book1]Sheet2>
```

Changed in version 0.9.0.

### **activate ()**

Activates the Sheet and returns it.

### **api**

Returns the native object (pywin32 or appscript obj) of the engine being used.

New in version 0.9.0.

### **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

```
>>> import xlwings as xw
>>> wb = xw.Book ()
>>> wb.sheets['Sheet1'].autofit ('c')
>>> wb.sheets['Sheet1'].autofit ('r')
>>> wb.sheets['Sheet1'].autofit ()
```

New in version 0.2.3.

### **book**

Returns the Book of the specified Sheet. Read-only.

### **cells**

Returns a Range object that represents all the cells on the Sheet (not just the cells that are currently in use).

New in version 0.9.0.

### **charts**

See *Charts*

New in version 0.9.0.

### **clear ()**

Clears the content and formatting of the whole sheet.

### **clear\_contents ()**

Clears the content of the whole sheet but leaves the formatting.

### **delete ()**

Deletes the Sheet.

### **index**

Returns the index of the Sheet (1-based as in Excel).

**name**

Gets or sets the name of the Sheet.

**names**

Returns a names collection that represents all the sheet-specific names (names defined with the “SheetName!” prefix).

New in version 0.9.0.

**pictures**

See *Pictures*

New in version 0.9.0.

**range** (*cell1*, *cell2=None*)

Returns a Range object from the active sheet of the active book, see *Range()*.

New in version 0.9.0.

**select** ()

Selects the Sheet. Select only works on the active book.

New in version 0.9.0.

**shapes**

See *Shapes*

New in version 0.9.0.

## Range

**class** `xlwings.Range` (*cell1=None*, *cell2=None*, *\*\*options*)

Returns a Range object that represents a cell or a range of cells.

### Parameters

- **cell1** (*str or tuple or Range*) – Name of the range in the upper-left corner in A1 notation or as index-tuple or as name or as `xw.Range` object. It can also specify a range using the range operator (a colon), .e.g. ‘A1:B2’
- **cell2** (*str or tuple or Range, default None*) – Name of the range in the lower-right corner in A1 notation or as index-tuple or as name or as `xw.Range` object.

### Examples

Active Sheet:

```
import xlwings as xw
xw.Range('A1')
xw.Range('A1:C3')
xw.Range((1,1))
xw.Range((1,1), (3,3))
```

```
xw.Range('NamedRange')
xw.Range(xw.Range('A1'), xw.Range('B2'))
```

Specific Sheet:

```
xw.books['MyBook.xlsx'].sheets[0].range('A1')
```

**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.

### address

Returns a string value that represents the range reference. Use `get_address()` to be able to provide parameters.

New in version 0.9.0.

### api

Returns the native object (pywin32 or appscript obj) of the engine being used.

New in version 0.9.0.

### autofit()

Autofits the width and height of all cells in the range.

- To autofit only the width of the columns use `xw.Range('A1:B2').columns.autofit()`
- To autofit only the height of the rows use `xw.Range('A1:B2').rows.autofit()`

Changed in version 0.9.0.

### clear()

Clears the content and the formatting of a Range.

### clear\_contents()

Clears the content of a Range but leaves the formatting.

### 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**

```

>>> import xlwings as xw
>>> wb = xw.Book()
>>> xw.Range('A1').color = (255,255,255)
>>> xw.Range('A2').color
(255, 255, 255)
>>> xw.Range('A2').color = None
>>> xw.Range('A2').color is None
True

```

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.

**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:  $0 \leq \text{column\_width} \leq 255$

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.

**columns**

Returns a *RangeColumns* object that represents the columns in the specified range.

New in version 0.9.0.

**count**

Returns the number of cells.

**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

**end** (*direction*)

Returns a Range object that represents the cell at the end of the region that contains the source range. Equivalent to pressing Ctrl+Up, Ctrl+down, Ctrl+left, or Ctrl+right.

**Parameters** *direction* (One of 'up', 'down', 'right', 'left')–

### Examples

```
>>> import xlwings as xw
>>> wb = xw.Book()
>>> xw.Range('A1:B2').value = 1
>>> xw.Range('A1').end('down')
<Range [Book1]Sheet1!$A$2>
>>> xw.Range('B2').end('right')
<Range [Book1]Sheet1!$B$2>
```

New in version 0.9.0.

**expand** (*mode='table'*)

Expands the range according to the mode provided. Ignores empty top-left cells (unlike Range.end()).

**Parameters** *mode* (*str*, default 'table') – One of 'table' (=down and right), 'down', 'right'.

### Returns

**Return type** Range

### Examples

```
>>> import xlwings as xw
>>> wb = xw.Book()
>>> xw.Range('A1').value = [[None, 1], [2, 3]]
>>> xw.Range('A1').expand().address
$A$1:$B$2
>>> xw.Range('A1').expand('right').address
$A$1:$B$1
```

New in version 0.9.0.

**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.

**get\_address** (*row\_absolute=True, column\_absolute=True, include\_sheetname=False, external=False*)

Returns the address of the range in the specified format. `address` can be used instead if none of the defaults need to be changed.

#### 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

```
>>> import xlwings as xw
>>> wb = xw.Book()
>>> xw.Range((1,1)).get_address()
'$A$1'
>>> xw.Range((1,1)).get_address(False, False)
'A1'
>>> xw.Range((1,1), (3,3)).get_address(True, False, True)
'Sheet1!A$1:C$3'
>>> xw.Range((1,1), (3,3)).get_address(True, False, external=True)
'[Book1]Sheet1!A$1:C$3'
```

New in version 0.2.3.

#### height

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

#### Returns

**Return type** float

New in version 0.4.0.

#### hyperlink

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

## Examples

```
>>> import xlwings as xw
>>> wb = xw.Book()
>>> xw.Range('A1').value
'www.xlwings.org'
>>> xw.Range('A1').hyperlink
'http://www.xlwings.org'
```

New in version 0.3.0.

### **last\_cell**

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

#### **Returns**

**Return type** *Range*

## Example

```
>>> import xlwings as xw
>>> wb = xw.Book()
>>> rng = xw.Range('A1:E4')
>>> rng.last_cell.row, rng.last_cell.column
(4, 5)
```

New in version 0.3.5.

### **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.

### **name**

Sets or gets the name of a Range.

New in version 0.4.0.

### **number\_format**

Gets and sets the number\_format of a Range.

## Examples

```
>>> import xlwings as xw
>>> wb = xw.Book()
>>> xw.Range('A1').number_format
```



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

New in version 0.2.3.

**offset** (*row\_offset=0, column\_offset=0*)

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

**Returns Range object**

**Return type** *Range*

New in version 0.3.0.

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

**Returns**

**Return type** Range object

New in version 0.7.0.

**raw\_value**

Gets and sets the values directly as delivered from/accepted by the engine that is being used (pywin32 or appscript) without going through any of xlwings' data cleaning/converting. This can be helpful if speed is an issue but naturally will be engine specific, i.e. might remove the cross-platform compatibility.

**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 object**

**Return type** *Range*

New in version 0.3.0.

**row**

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

**Returns**

**Return type** Integer

New in version 0.3.5.

**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:  $0 \leq \text{row\_height} \leq 409.5$

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.

**rows**

Returns a *RangeRows* object that represents the rows in the specified range.

New in version 0.9.0.

**select ()**

Selects the range. Select only works on the active book.

New in version 0.9.0.

**shape**

Tuple of Range dimensions.

New in version 0.3.0.

**sheet**

Returns the Sheet object to which the Range belongs.

New in version 0.9.0.

**size**

Number of elements in the Range.

New in version 0.3.0.

**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.

**value**

Gets and sets the values for the given Range.

**Returns object**

**Return type** returned object depends on the converter being used, see *xlwings.Range.options()*

**width**

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

**Returns**

**Return type** float

New in version 0.4.0.

## RangeRows

**class** `xlwings.RangeRows` (*rng*)

Represents the rows of a range. Do not construct this class directly, use *Range.rows* instead.

### Example

```
import xlwings as xw

rng = xw.Range('A1:C4')

assert len(rng.rows) == 4 # or rng.rows.count

rng.rows[0].value = 'a'

assert rng.rows[2] == xw.Range('A3:C3')
assert rng.rows(2) == xw.Range('A2:C2')

for r in rng.rows:
    print(r.address)
```

**autofit()**

Autofits the height of the rows.

**count**

Returns the number of rows.

New in version 0.9.0.

### RangeColumns

**class** `xlwings.RangeColumns` (*rng*)

Represents the columns of a range. Do not construct this class directly, use `Range.columns` instead.

#### Example

```
import xlwings as xw

rng = xw.Range('A1:C4')

assert len(rng.columns) == 3 # or rng.columns.count

rng.columns[0].value = 'a'

assert rng.columns[2] == xw.Range('C1:C4')
assert rng.columns(2) == xw.Range('B1:B4')

for c in rng.columns:
    print(c.address)
```

**autofit** ()

Autofits the width of the columns.

**count**

Returns the number of columns.

New in version 0.9.0.

### Shapes

**class** `xlwings.main.Shapes` (*impl*)

A collection of all *shape* objects on the specified sheet:

```
>>> import xlwings as xw
>>> xw.books['Book1'].sheets[0].shapes
Shapes([<Shape 'Oval 1' in <Sheet [Book1]Sheet1>>, <Shape 'Rectangle 1'
↳in <Sheet [Book1]Sheet1>>])
```

New in version 0.9.0.

**api**

Returns the native object (pywin32 or appscript obj) of the engine being used.

**count**

Returns the number of objects in the collection.

## Shape

**class** `xlwings.Shape` (\*args, \*\*options)

The shape object is a member of the `shapes` collection:

```
>>> import xlwings as xw
>>> sht = xw.books['Book1'].sheets[0]
>>> sht.shapes[0] # or sht.shapes['ShapeName']
<Shape 'Rectangle 1' in <Sheet [Book1]Sheet1>>
```

Changed in version 0.9.0.

**activate** ()

Activates the shape.

New in version 0.5.0.

**delete** ()

Deletes the shape.

New in version 0.5.0.

**height**

Returns or sets the number of points that represent the height of the shape.

New in version 0.5.0.

**left**

Returns or sets the number of points that represent the horizontal position of the shape.

New in version 0.5.0.

**name**

Returns or sets the name of the shape.

New in version 0.5.0.

**parent**

Returns the parent of the shape.

New in version 0.9.0.

**top**

Returns or sets the number of points that represent the vertical position of the shape.

New in version 0.5.0.

**type**

Returns the type of the shape.

New in version 0.9.0.

**width**

Returns or sets the number of points that represent the width of the shape.

New in version 0.5.0.

### Charts

**class** `xlwings.main.Charts` (*impl*)

A collection of all *chart* objects on the specified sheet:

```
>>> import xlwings as xw
>>> xw.books['Book1'].sheets[0].charts
Charts([<Chart 'Chart 1' in <Sheet [Book1]Sheet1>>, <Chart 'Chart 1' in
↳<Sheet [Book1]Sheet1>>])
```

New in version 0.9.0.

**add** (*left=0, top=0, width=355, height=211*)

Creates a new chart on the specified sheet.

#### Parameters

- **left** (*float, default 0*) – left position in points
- **top** (*float, default 0*) – top position in points
- **width** (*float, default 355*) – width in points
- **height** (*float, default 211*) – height in points

#### Returns

Return type *Chart*

### Examples

```
>>> import xlwings as xw
>>> sht = xw.Book().sheets[0]
>>> sht.range('A1').value = [['Foo1', 'Foo2'], [1, 2]]
>>> chart = sht.charts.add()
>>> chart.set_source_data(sht.range('A1').expand())
>>> chart.chart_type = 'line'
>>> chart.name
'Chart1'
```

#### **api**

Returns the native object (`pywin32` or `appscript` obj) of the engine being used.

#### **count**

Returns the number of objects in the collection.

### Chart

**class** `xlwings.Chart` (*name\_or\_index=None, impl=None*)

The chart object is a member of the *charts* collection:

```
>>> import xlwings as xw
>>> sht = xw.books['Book1'].sheets[0]
>>> sht.charts[0] # or sht.charts['ChartName']
<Chart 'Chart 1' in <Sheet [Book1]Sheet1>>
```

**api**

Returns the native object (pywin32 or appscript obj) of the engine being used.

New in version 0.9.0.

**chart\_type**

Returns and sets the chart type of the chart.

New in version 0.1.1.

**delete()**

Deletes the chart.

**height**

Returns or sets the number of points that represent the height of the chart.

**left**

Returns or sets the number of points that represent the horizontal position of the chart.

**name**

Returns or sets the name of the chart.

**parent**

Returns the parent of the chart.

New in version 0.9.0.

**set\_source\_data(source)**

Sets the source data range for the chart.

**Parameters source** (*Range*) – Range object, e.g. `xw.books['Book1'].sheets[0].range('A1')`

**top**

Returns or sets the number of points that represent the vertical position of the chart.

**width**

Returns or sets the number of points that represent the width of the chart.

## Pictures

**class xlwings.main.Pictures** (*impl*)

A collection of all *picture* objects on the specified sheet:

```
>>> import xlwings as xw
>>> xw.books['Book1'].sheets[0].pictures
Pictures([<Picture 'Picture 1' in <Sheet [Book1]Sheet1>>, <Picture
↳ 'Picture 2' in <Sheet [Book1]Sheet1>>])
```

New in version 0.9.0.

**add** (*image*, *link\_to\_file=False*, *save\_with\_document=True*, *left=0*, *top=0*, *width=None*, *height=None*, *name=None*, *update=False*)  
Adds a picture to the specified sheet.

### Parameters

- **image** (*str* or *matplotlib.figure.Figure*) – Either a filepath or a Matplotlib figure object.
- **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.
- **name** (*str*, *default None*) – Excel picture name. Defaults to Excel standard name if not provided, e.g. 'Picture 1'.
- **update** (*bool*, *default False*) – Replace an existing picture with the same name. Requires name to be set.

### Returns

Return type *Picture*

## Examples

### 1. Picture

```
>>> import xlwings as xw
>>> sht = xw.Book().sheets[0]
>>> sht.pictures.add(r'C:\path\to\file.jpg')
<Picture 'Picture 1' in <Sheet [Book1]Sheet1>>
```

### 2. Matplotlib

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.plot([1, 2, 3, 4, 5])
>>> sht.pictures.add(fig, name='MyPlot', update=True)
<Picture 'MyPlot' in <Sheet [Book1]Sheet1>>
```

### api

Returns the native object (pywin32 or appsript obj) of the engine being used.



**count**

Returns the number of objects in the collection.

**Picture**

**class** `xlwings.Picture` (*impl=None*)

The picture object is a member of the `pictures` collection:

```
>>> import xlwings as xw
>>> sht = xw.books['Book1'].sheets[0]
>>> sht.pictures[0] # or sht.charts['PictureName']
<Picture 'Picture 1' in <Sheet [Book1]Sheet1>>
```

Changed in version 0.9.0.

**api**

Returns the native object (pywin32 or appscript obj) of the engine being used.

New in version 0.9.0.

**delete()**

Deletes the picture.

New in version 0.5.0.

**height**

Returns or sets the number of points that represent the height of the picture.

New in version 0.5.0.

**left**

Returns or sets the number of points that represent the horizontal position of the picture.

New in version 0.5.0.

**name**

Returns or sets the name of the picture.

New in version 0.5.0.

**parent**

Returns the parent of the picture.

New in version 0.9.0.

**top**

Returns or sets the number of points that represent the vertical position of the picture.

New in version 0.5.0.

**update** (*image*)

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

**Parameters** `image` (*str* or *matplotlib.figure.Figure*) – Either a filepath or a Matplotlib figure object.

New in version 0.5.0.

### **width**

Returns or sets the number of points that represent the width of the picture.

New in version 0.5.0.

## Names

**class** `xlwings.main.Names` (*impl*)

A collection of all *name* objects in the workbook:

```
>>> import xlwings as xw
>>> sht = xw.books['Book1'].sheets[0]
>>> sht.names
[<Name 'MyName': =Sheet1!$A$3>]
```

New in version 0.9.0.

**add** (*name*, *refers\_to*)

Defines a new name for a range of cells.

### **Parameters**

- **name** (*str*) – Specifies the text to use as the name. Names cannot include spaces and cannot be formatted as cell references.
- **refers\_to** (*str*) – Describes what the name refers to, in English, using A1-style notation.

### **Returns**

**Return type** *Name*

New in version 0.9.0.

**api**

Returns the native object (pywin32 or appscript obj) of the engine being used.

New in version 0.9.0.

**count**

Returns the number of objects in the collection.

## Name

**class** `xlwings.Name` (*impl*)

The name object is a member of the *names* collection:

```
>>> import xlwings as xw
>>> sht = xw.books['Book1'].sheets[0]
>>> sht.names[0] # or sht.names['MyName']
<Name 'MyName': =Sheet1!$A$3>
```

New in version 0.9.0.

**api**

Returns the native object (`pywin32` or `appscript` obj) of the engine being used.

New in version 0.9.0.

**delete ()**

Deletes the name.

New in version 0.9.0.

**name**

Returns or sets the name of the name object.

New in version 0.9.0.

**refers\_to**

Returns or sets the formula that the name is defined to refer to, in A1-style notation, beginning with an equal sign.

New in version 0.9.0.

**refers\_to\_range**

Returns the Range object referred to by a Name object.

New in version 0.9.0.

## UDF decorators

`xlwings.func` (*category*="xlwings", *volatile*=False, *call\_in\_wizard*=True)

Functions decorated with `xlwings.func` will be imported as `Function` to Excel when running "Import Python UDFs".

**category** [int or str, default "xlwings"] 1-14 represent built-in categories, for user-defined categories use strings

New in version 0.10.3.

**volatile** [bool, default False] Marks a user-defined function as volatile. A volatile function must be recalculated whenever calculation occurs in any cells on the worksheet. A nonvolatile function is recalculated only when the input variables change. This method has no effect if it's not inside a user-defined function used to calculate a worksheet cell.

New in version 0.10.3.

**call\_in\_wizard** [bool, default True] Set to False to suppress the function call in the function wizard.

New in version 0.10.3.

`xlwings.sub` ()

Functions decorated with `xlwings.sub` will be imported as `Sub` (i.e. macro) to Excel when running "Import Python UDFs".

`xlwings.arg` (*arg*, *convert=None*, *\*\*options*)

Apply converters and options to arguments, see also `Range.options()`.

### Examples:

Convert `x` into a 2-dimensional numpy array:

```
import xlwings as xw
import numpy as np

@xw.func
@xw.arg('x', np.array, ndim=2)
def add_one(x):
    return x + 1
```

`xlwings.ret` (*convert=None*, *\*\*options*)

Apply converters and options to return values, see also `Range.options()`.

### Examples

1. Suppress the index and header of a returned DataFrame:

```
import pandas as pd

@xw.func
@xw.ret(index=False, header=False)
def get_dataframe(n, m):
    return pd.DataFrame(np.arange(n * m).reshape((n, m)))
```

2. Dynamic array:

`expand='table'` turns the UDF into a dynamic array. Currently you must not use volatile functions as arguments of a dynamic array, e.g. you cannot use `=TODAY()` as part of a dynamic array. Also note that a dynamic array needs an empty row and column at the bottom and to the right and will overwrite existing data without warning.

Unlike standard Excel arrays, dynamic arrays are being used from a single cell like a standard function and auto-expand depending on the dimensions of the returned array:

```
import xlwings as xw
import numpy as np

@xw.func
@xw.ret(expand='table')
def dynamic_array(n, m):
    return np.arange(n * m).reshape((n, m))
```

New in version 0.10.0.

**A**

activate() (xlwings.App method), 61  
activate() (xlwings.Book method), 64  
activate() (xlwings.Shape method), 79  
activate() (xlwings.Sheet method), 67  
active (xlwings.main.Apps attribute), 60  
active (xlwings.main.Books attribute), 63  
active (xlwings.main.Sheets attribute), 67  
add() (xlwings.main.Apps method), 60  
add() (xlwings.main.Books method), 63  
add() (xlwings.main.Charts method), 80  
add() (xlwings.main.Names method), 84  
add() (xlwings.main.Pictures method), 82  
add() (xlwings.main.Sheets method), 67  
add\_hyperlink() (xlwings.Range method), 70  
address (xlwings.Range attribute), 70  
api (xlwings.App attribute), 61  
api (xlwings.Book attribute), 64  
api (xlwings.Chart attribute), 81  
api (xlwings.main.Charts attribute), 80  
api (xlwings.main.Names attribute), 84  
api (xlwings.main.Pictures attribute), 82  
api (xlwings.main.Shapes attribute), 78  
api (xlwings.Name attribute), 85  
api (xlwings.Picture attribute), 83  
api (xlwings.Range attribute), 70  
api (xlwings.Sheet attribute), 68  
App (class in xlwings), 60  
app (xlwings.Book attribute), 64  
Apps (class in xlwings.main), 60  
autofit() (xlwings.Range method), 70  
autofit() (xlwings.RangeColumns method), 78  
autofit() (xlwings.RangeRows method), 77  
autofit() (xlwings.Sheet method), 68

**B**

Book (class in xlwings), 64  
book (xlwings.Sheet attribute), 68  
Books (class in xlwings.main), 63  
books (xlwings.App attribute), 61

**C**

calculate() (xlwings.App method), 61  
calculation (xlwings.App attribute), 61  
caller() (xlwings.Book class method), 64  
cells (xlwings.Sheet attribute), 68  
Chart (class in xlwings), 80  
chart\_type (xlwings.Chart attribute), 81  
Charts (class in xlwings.main), 80  
charts (xlwings.Sheet attribute), 68  
clear() (xlwings.Range method), 70  
clear() (xlwings.Sheet method), 68  
clear\_contents() (xlwings.Range method), 70  
clear\_contents() (xlwings.Sheet method), 68  
close() (xlwings.Book method), 65  
color (xlwings.Range attribute), 70  
column (xlwings.Range attribute), 71  
column\_width (xlwings.Range attribute), 71  
columns (xlwings.Range attribute), 71  
count (xlwings.main.Apps attribute), 60  
count (xlwings.main.Charts attribute), 80  
count (xlwings.main.Names attribute), 84  
count (xlwings.main.Pictures attribute), 82  
count (xlwings.main.Shapes attribute), 78  
count (xlwings.Range attribute), 71  
count (xlwings.RangeColumns attribute), 78  
count (xlwings.RangeRows attribute), 77  
current\_region (xlwings.Range attribute), 71

## D

delete() (xlwings.Chart method), 81  
delete() (xlwings.Name method), 85  
delete() (xlwings.Picture method), 83  
delete() (xlwings.Shape method), 79  
delete() (xlwings.Sheet method), 68  
display\_alerts (xlwings.App attribute), 61

## E

end() (xlwings.Range method), 72  
expand() (xlwings.Range method), 72

## F

formula (xlwings.Range attribute), 72  
formula\_array (xlwings.Range attribute), 72  
fullname (xlwings.Book attribute), 65

## G

get\_address() (xlwings.Range method), 72

## H

height (xlwings.Chart attribute), 81  
height (xlwings.Picture attribute), 83  
height (xlwings.Range attribute), 73  
height (xlwings.Shape attribute), 79  
hwnd (xlwings.App attribute), 61  
hyperlink (xlwings.Range attribute), 73

## I

index (xlwings.Sheet attribute), 68

## K

kill() (xlwings.App method), 62

## L

last\_cell (xlwings.Range attribute), 74  
left (xlwings.Chart attribute), 81  
left (xlwings.Picture attribute), 83  
left (xlwings.Range attribute), 74  
left (xlwings.Shape attribute), 79

## M

macro() (xlwings.App method), 62  
macro() (xlwings.Book method), 65

## N

Name (class in xlwings), 84  
name (xlwings.Book attribute), 65

name (xlwings.Chart attribute), 81  
name (xlwings.Name attribute), 85  
name (xlwings.Picture attribute), 83  
name (xlwings.Range attribute), 74  
name (xlwings.Shape attribute), 79  
name (xlwings.Sheet attribute), 68  
Names (class in xlwings.main), 84  
names (xlwings.Book attribute), 65  
names (xlwings.Sheet attribute), 69  
number\_format (xlwings.Range attribute), 74

## O

offset() (xlwings.Range method), 75  
open() (xlwings.main.Books method), 63  
open\_template() (xlwings.Book static method), 65  
options() (xlwings.Range method), 75

## P

parent (xlwings.Chart attribute), 81  
parent (xlwings.Picture attribute), 83  
parent (xlwings.Shape attribute), 79  
Picture (class in xlwings), 83  
Pictures (class in xlwings.main), 81  
pictures (xlwings.Sheet attribute), 69  
pid (xlwings.App attribute), 62

## Q

quit() (xlwings.App method), 62

## R

Range (class in xlwings), 69  
range() (xlwings.App method), 62  
range() (xlwings.Sheet method), 69  
RangeColumns (class in xlwings), 78  
RangeRows (class in xlwings), 77  
raw\_value (xlwings.Range attribute), 75  
refers\_to (xlwings.Name attribute), 85  
refers\_to\_range (xlwings.Name attribute), 85  
resize() (xlwings.Range method), 75  
row (xlwings.Range attribute), 76  
row\_height (xlwings.Range attribute), 76  
rows (xlwings.Range attribute), 76

## S

save() (xlwings.Book method), 66  
screen\_updating (xlwings.App attribute), 62  
select() (xlwings.Range method), 76  
select() (xlwings.Sheet method), 69

selection (xlwings.App attribute), 63  
selection (xlwings.Book attribute), 66  
set\_mock\_caller() (xlwings.Book method), 66  
set\_source\_data() (xlwings.Chart method), 81  
Shape (class in xlwings), 79  
shape (xlwings.Range attribute), 76  
Shapes (class in xlwings.main), 78  
shapes (xlwings.Sheet attribute), 69  
Sheet (class in xlwings), 67  
sheet (xlwings.Range attribute), 76  
Sheets (class in xlwings.main), 67  
sheets (xlwings.Book attribute), 67  
size (xlwings.Range attribute), 76

## T

top (xlwings.Chart attribute), 81  
top (xlwings.Picture attribute), 83  
top (xlwings.Range attribute), 76  
top (xlwings.Shape attribute), 79  
type (xlwings.Shape attribute), 79

## U

update() (xlwings.Picture method), 83

## V

value (xlwings.Range attribute), 77  
version (xlwings.App attribute), 63  
view() (in module xlwings), 59  
visible (xlwings.App attribute), 63

## W

width (xlwings.Chart attribute), 81  
width (xlwings.Picture attribute), 84  
width (xlwings.Range attribute), 77  
width (xlwings.Shape attribute), 79

## X

xlwings (module), 59  
xlwings.arg() (in module xlwings), 85  
xlwings.func() (in module xlwings), 85  
xlwings.ret() (in module xlwings), 86  
xlwings.sub() (in module xlwings), 85