

---

# **xlwings - Make Excel Fly!**

*Release dev*

**Zoomer Analytics LLC**

**Dec 16, 2021**



## GETTING STARTED

<b>1</b>	<b>Video course</b>	<b>1</b>
<b>2</b>	<b>Installation</b>	<b>3</b>
2.1	Prerequisites . . . . .	3
2.2	Installation . . . . .	3
2.3	Add-in . . . . .	4
2.4	Dependencies . . . . .	4
2.5	How to activate xlwings PRO . . . . .	4
2.6	Optional Dependencies . . . . .	5
2.7	Update . . . . .	5
2.8	Uninstall . . . . .	6
<b>3</b>	<b>Quickstart</b>	<b>7</b>
3.1	1. Interacting with Excel from a Jupyter notebook . . . . .	7
3.2	2. Scripting: Automate/interact with Excel from Python . . . . .	7
3.3	3. Macros: Call Python from Excel . . . . .	8
3.4	4. 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	App context manager . . . . .	14
5.4	Range indexing/slicing . . . . .	14
5.5	Range Shortcuts . . . . .	14
5.6	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 . . . . .	20
6.6	Pandas Series . . . . .	20
6.7	Chunking: Read/Write big DataFrames etc. . . . .	21
<b>7</b>	<b>Add-in &amp; Settings</b>	<b>23</b>
7.1	Run main . . . . .	23
7.2	Installation . . . . .	24
7.3	User Settings . . . . .	24
7.4	Environment Variables . . . . .	25
7.5	User Config: Ribbon/Config File . . . . .	25
7.6	Workbook Directory Config: Config file . . . . .	26
7.7	Workbook Config: xlwings.conf Sheet . . . . .	26
7.8	Alternative: Standalone VBA module . . . . .	27
<b>8</b>	<b>RunPython</b>	<b>29</b>
8.1	xlwings add-in . . . . .	29
8.2	Call Python with “RunPython” . . . . .	29
8.3	Function Arguments and Return Values . . . . .	30
<b>9</b>	<b>User Defined Functions (UDFs)</b>	<b>31</b>
9.1	One-time Excel preparations . . . . .	31
9.2	Workbook preparation . . . . .	31
9.3	A simple UDF . . . . .	32
9.4	Array formulas: Get efficient . . . . .	33
9.5	Array formulas with NumPy and Pandas . . . . .	34
9.6	@xw.arg and @xw.ret decorators . . . . .	34
9.7	Dynamic Array Formulas . . . . .	35
9.8	Docstrings . . . . .	37
9.9	The “caller” argument . . . . .	37
9.10	The “vba” keyword . . . . .	37
9.11	Macros . . . . .	38
9.12	Call UDFs from VBA . . . . .	38
9.13	Asynchronous UDFs . . . . .	39
<b>10</b>	<b>Matplotlib &amp; Plotly Charts</b>	<b>41</b>
10.1	Matplotlib . . . . .	41
10.2	Plotly static charts . . . . .	44
<b>11</b>	<b>Jupyter Notebooks: Interact with Excel</b>	<b>47</b>
11.1	The view function . . . . .	47
11.2	The load function . . . . .	48
<b>12</b>	<b>Command Line Client (CLI)</b>	<b>49</b>
<b>13</b>	<b>Deployment</b>	<b>51</b>
13.1	Zip files . . . . .	51
13.2	RunFrozenPython . . . . .	51
<b>14</b>	<b>OneDrive and SharePoint</b>	<b>53</b>

14.1	OneDrive (Personal)	53
14.2	OneDrive for Business	54
14.3	SharePoint (Online and On-Premises)	54
14.4	Implementation Details & Limitations	54
<b>15</b>	<b>Troubleshooting</b>	<b>57</b>
15.1	Issue: dll not found	57
15.2	Issue: Files that are saved on OneDrive or SharePoint cause an error to pop up	57
<b>16</b>	<b>Converters and Options</b>	<b>59</b>
16.1	Default Converter	60
16.2	Built-in Converters	63
16.3	Custom Converter	67
<b>17</b>	<b>Debugging</b>	<b>71</b>
17.1	RunPython	72
17.2	UDF debug server	72
<b>18</b>	<b>Extensions</b>	<b>75</b>
18.1	In-Excel SQL	75
<b>19</b>	<b>Custom Add-ins</b>	<b>77</b>
19.1	Quickstart	77
19.2	Changing the Ribbon menu	79
19.3	Importing UDFs	79
19.4	Configuration	79
19.5	Installation	79
19.6	Renaming your add-in	80
19.7	Deployment	80
<b>20</b>	<b>Threading and Multiprocessing</b>	<b>81</b>
20.1	Threading	81
20.2	Multiprocessing	82
<b>21</b>	<b>Missing Features</b>	<b>85</b>
21.1	Example: Workaround to use VBA's Range.WrapText	85
<b>22</b>	<b>xlwings with other Office Apps</b>	<b>87</b>
22.1	How To	87
22.2	Config	88
<b>23</b>	<b>xlwings PRO Overview</b>	<b>89</b>
23.1	PRO Features	89
23.2	More Infos	90
<b>24</b>	<b>xlwings Reports</b>	<b>91</b>
24.1	Quickstart	92
24.2	DataFrames	94
24.3	Excel Tables	98

24.4	Excel Charts . . . . .	100
24.5	Images . . . . .	103
24.6	Matplotlib and Plotly Plots . . . . .	106
24.7	Text . . . . .	107
24.8	Date and Time . . . . .	110
24.9	Number Format . . . . .	110
24.10	Frames: Multi-column Layout . . . . .	110
24.11	PDF Layout . . . . .	112
<b>25</b>	<b>Markdown Formatting</b>	<b>115</b>
<b>26</b>	<b>Releasing xlwings Tools</b>	<b>119</b>
26.1	Step 1: One-Click Installer . . . . .	119
26.2	Step 2: Release Command (CLI) . . . . .	122
26.3	Updating a Release . . . . .	123
26.4	Embedded Code Explained . . . . .	124
<b>27</b>	<b>Permissioning of Code Execution</b>	<b>127</b>
27.1	Prerequisites . . . . .	127
27.2	Configuration . . . . .	128
27.3	GET request . . . . .	128
27.4	POST request . . . . .	129
27.5	Implementation Details & Limitations . . . . .	130
<b>28</b>	<b>Python API</b>	<b>131</b>
28.1	Top-level functions . . . . .	131
28.2	Object model . . . . .	132
28.3	UDF decorators . . . . .	176
28.4	Reports . . . . .	178
<b>29</b>	<b>REST API</b>	<b>181</b>
29.1	Quickstart . . . . .	181
29.2	Run the server . . . . .	183
29.3	Indexing . . . . .	183
29.4	Range Options . . . . .	183
29.5	Endpoint overview . . . . .	183
29.6	Endpoint details . . . . .	184
	<b>Index</b>	<b>213</b>

## **VIDEO COURSE**

Those who prefer a didactically structured video course over this documentation should have a look at our video course:

<https://training.xlwings.org/p/xlwings>

It's also a great way to support the ongoing development of xlwings :)



## INSTALLATION

### 2.1 Prerequisites

- xlwings requires an **installation of Excel** and therefore only works on **Windows** and **macOS**. Note that macOS currently does not support UDFs.
- xlwings requires at least Python 3.6.

Here are the last versions of xlwings to support:

- Python 3.5: 0.19.5
- Python 2.7: 0.16.6

### 2.2 Installation

xlwings comes pre-installed with

- [Anaconda](#) (Windows and macOS)
- [WinPython](#) (Windows only) Make sure **not** to take the **dot** version as this only contains Python.

If you are new to Python or have trouble installing xlwings, one of these distributions is highly recommended. Otherwise, you can also install it manually with pip:

```
pip install xlwings
```

or conda:

```
conda install xlwings
```

Note that the official conda package might be a few releases behind. You can, however, use the `conda-forge` channel (replace `install` with `upgrade` if xlwings is already installed):

```
conda install -c conda-forge xlwings
```

**Note:** When you are on macOS 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. This is done automatically if you install the addin via `$ xlwings addin install`.

---

## 2.3 Add-in

To install the add-in, run the following command:

```
xlwings addin install
```

To call Excel from Python, you don't need an add-in. Also, you can use a single file VBA module (*standalone workbook*) instead of the add-in. For more details, see [Add-in & Settings](#).

**Note:** The add-in needs to be the same version as the Python package. Make sure to re-install the add-in after upgrading the xlwings package.

---

## 2.4 Dependencies

- **Windows:** pywin32
- **Mac:** psutil, applescript

The dependencies are automatically installed via conda or pip.

## 2.5 How to activate xlwings PRO

xlwings PRO offers access to *additional functionality*. All PRO features are marked with xlwings *PRO* in the docs.

**Note:** To get access to the additional functionality of xlwings PRO, you need a license key and at least xlwings v0.19.0. Everything under the `xlwings.pro` subpackage is distributed under a commercial license. See [xlwings PRO Overview](#) for more details.

---

To activate the license key, run the following command:

```
xlwings license update -k LICENSE_KEY
```

Make sure to replace `LICENSE_KEY` with your personal key. This will store the license key under your `xlwings.conf` file (see [User Config: Ribbon/Config File](#) for where this is on your system). Alternatively, you can also store the license key as an environment variable with the name `XLWINGS_LICENSE_KEY`.

xlwings PRO requires additionally the `cryptography` and `Jinja2` packages which come preinstalled with Anaconda and WinPython. Otherwise, install them via `pip` or `conda`.

With `pip`, you can also run `pip install "xlwings[pro]"` which will take care of the extra dependencies for xlwings PRO.

## 2.6 Optional Dependencies

- NumPy
- Pandas
- Matplotlib
- Pillow/PIL
- Flask (for REST API)
- `cryptography` (for `xlwings.pro`)
- `Jinja2` (for `xlwings.pro.reports`)
- `requests` (for permissioning)

These packages are not required but highly recommended as they play very nicely with xlwings. They are all pre-installed with Anaconda. With `pip`, you can install xlwings with all optional dependencies as follows:

```
pip install "xlwings[all]"
```

## 2.7 Update

To update to the latest xlwings version, run the following in a command prompt:

```
pip install --upgrade xlwings
```

or:

```
conda update -c conda-forge xlwings
```

Make sure to keep your version of the Excel add-in in sync with your Python package by running the following (make sure to close Excel first):

```
xlwings addin install
```

## 2.8 Uninstall

To uninstall xlwings completely, first uninstall the add-in, then uninstall the xlwings package using the same method (pip or conda) that you used for installing it:

```
xlwings addin remove
```

Then

```
pip uninstall xlwings
```

or:

```
conda remove xlwings
```

Finally, manually remove the `.xlwings` directory in your home folder if it exists.

## QUICKSTART

This guide assumes you have `xlwings` already installed. If that's not the case, head over to [Installation](#).

### 3.1 1. Interacting with Excel from a Jupyter notebook

If you're just interested in getting a pandas DataFrame in and out of your Jupyter notebook, you can use the view and load functions, see *Jupyter Notebooks: Interact with Excel*.

### 3.2 2. 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 a file that is open or 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. You will find your app instance key (the PID) via `xw.apps.keys()`:

```
>>> xw.apps[10559].books['FileName.xlsx']
```

Instantiate a sheet object:

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

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

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

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

```
>>> sheet.range('A1').value = [['Foo 1', 'Foo 2', 'Foo 3'], [10.0, 20.0, 30.0]]
>>> sheet.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'])
>>> sheet.range('A1').value = df
>>> sheet.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>]
>>> sheet.pictures.add(fig, name='MyPlot', update=True)
<Picture 'MyPlot' in <Sheet [Workbook4]Sheet1>>
```

### 3.3 3. Macros: Call Python from Excel

You can call Python functions either by clicking the Run button (new in v0.16) in the add-in or from VBA using the `RunPython` function:

The Run button expects a function called `main` in a Python module with the same name as your workbook. The great thing about that approach is that you don't need your workbooks to be macro-enabled, you can save it as `xlsx`.

If you want to call any Python function no matter in what module it lives or what name it has, use `RunPython`:

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

---

**Note:** Per default, `RunPython` expects `hello.py` in the same directory as the Excel file with the same name, **but you can change both of these things**: if your Python file is in a different folder, add that folder to the `PYTHONPATH` in the config. If the file has a different name, change the `RunPython` command accordingly.

---

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 add-in installed or have the workbooks setup in the standalone mode. 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`.

For details about the addin, see [Add-in & Settings](#).

## 3.4 4. 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 f'Hello {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 a step-by-step tutorial, see [User Defined Functions \(UDFs\)](#).



## 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'
```

### 4.1 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[10559] for existing apps, get
↳ the available PIDs via xw.apps.keys()
>>> app.books['Book1']
```

Note that you usually should use `App` as a context manager as this will make sure that the Excel instance is closed and cleaned up again properly:

```
with xw.App() as app:
    book = 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`.

---

## 4.2 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.

## 4.3 User Defined Functions (UDFs)

Unlike `RunPython`, UDFs don't need a call to `xw.Book.caller()`, see *User Defined Functions (UDFs)*. You'll usually use the `caller` argument which returns the xlwings range object from where you call the function.

## SYNTAX OVERVIEW

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
```

### 5.1 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
>>> sheet = xw.sheets.active # in active book
>>> sheet = 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'))
```

## 5.2 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[763].books[0].sheets[0].range('A1')
xw.apps(10559).books(1).sheets(1).range('A1')
xw.apps[763].books['Book1'].sheets['Sheet1'].range('A1')
xw.apps(10559).books('Book1').sheets('Sheet1').range('A1')
```

Note that the apps keys are different for you as they are the process IDs (PID). You can get the list of your PIDs via `xw.apps.keys()`.

## 5.3 App context manager

If you want to open a new Excel instance via `App()`, you usually should use `App` as a context manager as this will make sure that the Excel instance is closed and cleaned up again properly:

```
with xw.App() as app:
    book = app.books['Book1']
```

## 5.4 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>
```

## 5.5 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:

```
>>> sheet = xw.Book().sheets['Sheet1']
>>> sheet['A1']
```

(continues on next page)

(continued from previous page)

```
<Range [Book1]Sheet1!$A$1>
>>> sheet['A1:B5']
<Range [Book1]Sheet1!$A$1:$B$5>
>>> sheet[0, 1]
<Range [Book1]Sheet1!$B$1>
>>> sheet[:, 10, :10]
<Range [Book1]Sheet1!$A$1:$J$10>
```

## 5.6 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[10559].books[0].sheets[0].range('A1')
>>> rng.sheet.book.app
<Excel App 10559>
```



## 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
```

### 6.1 Single Cells

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

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

## 6.2 Lists

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

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

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

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

---

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

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

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

---

## 6.3 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:

```
>>> sheet = xw.Book().sheets[0]
>>> sheet.range('A1').value = [[1,2], [3,4]]
>>> rng1 = sheet.range('A1').expand('table') # or just .expand()
>>> rng2 = sheet.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]]
>>> sheet.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 its size without having to adjust your code, e.g. by using something like `sheet.range('NamedRange').expand().value`.

## 6.4 NumPy arrays

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

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

## 6.5 Pandas DataFrames

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

## 6.6 Pandas Series

```
>>> import pandas as pd
>>> import numpy as np
>>> sheet = 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
>>> sheet.range('A1').value = s
>>> sheet.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.: `sheet.range('A1').value = np.eye(10)`

---

## 6.7 Chunking: Read/Write big DataFrames etc.

When you read and write from or to big ranges, you may have to chunk them or you will hit a timeout or a memory error. The ideal chunksize will depend on your system and size of the array, so you will have to try out a few different chunksizes to find one that works well:

```
import pandas as pd
import numpy as np
sheet = xw.Book().sheets[0]
data = np.arange(75_000 * 20).reshape(75_000, 20)
df = pd.DataFrame(data=data)
sheet['A1'].options(chunksize=10_000).value = df
```

And the same for reading:

```
# As DataFrame
df = sheet['A1'].expand().options(pd.DataFrame, chunksize=10_000).value
# As list of list
df = sheet['A1'].expand().options(chunksize=10_000).value
```



## ADD-IN & SETTINGS



The xlwings add-in is the preferred way to be able to use the `Run main` button, `RunPython` or UDFs. Note that you don't need an add-in if you just want to manipulate Excel by running a Python script.

---

**Note:** The ribbon of the add-in is compatible with Excel  $\geq 2007$  on Windows and  $\geq 2016$  on Mac. On Mac, all UDF related functionality is not available.

---

---

**Note:** The add-in is password protected with the password `xlwings`. For debugging or to add new extensions, you need to unprotect it. Alternatively, you can also install the add-in via `xlwings addin install --unprotected`.

---

### 7.1 Run main

New in version 0.16.0.

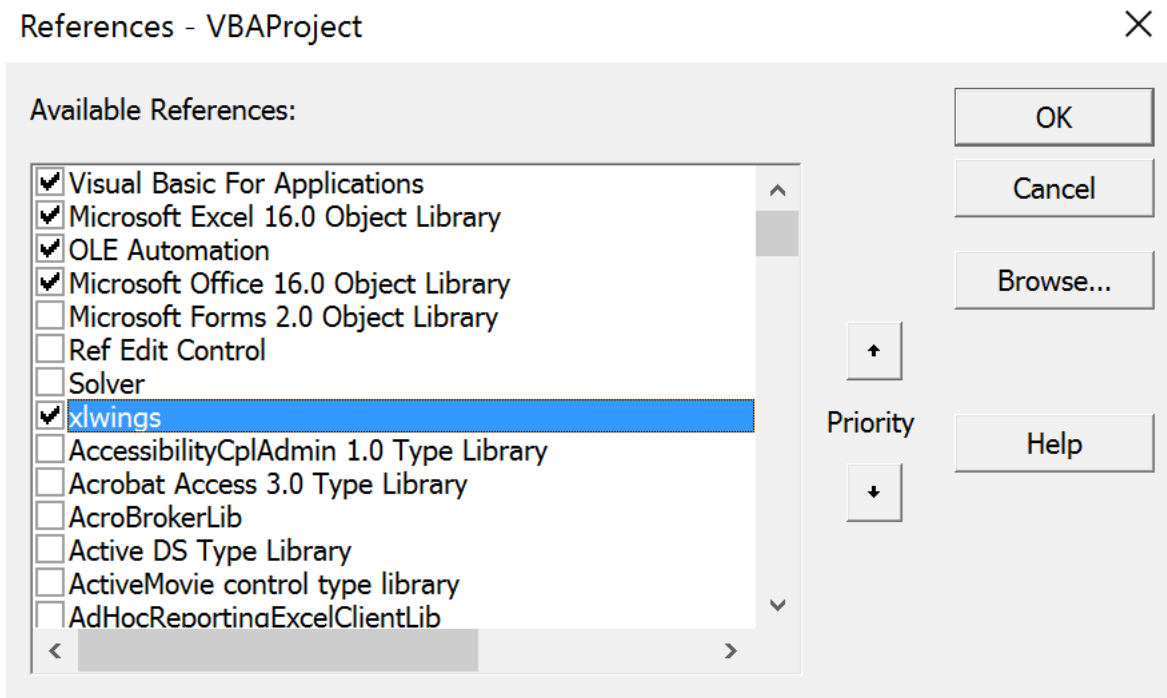
The `Run main` button is the easiest way to run your Python code: It runs a function called `main` in a Python module that has the same name as your workbook. This allows you to save your workbook as `xlsx` without enabling macros. The `xlwings quickstart` command will create a workbook that will automatically work with the `Run` button.

## 7.2 Installation

To install the add-in, use the command line client:

```
xlwings addin install
```

Technically, this copies the add-in from Python's installation directory to Excel's XLSTART folder. Then, to use RunPython or UDFs in a workbook, you need to set a reference to xlwings in the VBA editor, see screenshot (Windows: Tools > References..., Mac: it's on the lower left corner of the VBA editor). Note that when you create a workbook via xlwings quickstart, the reference should already be set.



## 7.3 User Settings

When you install the add-in for the first time, it will get auto-configured and therefore, a quickstart project should work out of the box. For fine-tuning, here are the available settings:

- **Interpreter:** This is the path to the Python interpreter. This works also with virtual or conda envs on Mac. If you use conda envs on Windows, then leave this empty and use Conda Path and Conda Env below instead. Examples: "C:\Python39\pythonw.exe" or "/usr/local/bin/python3.9". Note that in the settings, this is stored as Interpreter\_Win or Interpreter\_Mac, respectively, see below!
- **PYTHONPATH:** If the source file of your code is not found, add the path to its directory here.
- **Conda Path:** If you are on Windows and use Anaconda or Miniconda, then type here the path to your installation, e.g. C:\Users\Username\Miniconda3 or %USERPROFILE%\Anaconda. NOTE that you need at least conda 4.6! You also need to set Conda Env, see next point.

- **Conda Env:** If you are on Windows and use Anaconda or Miniconda, type here the name of your conda env, e.g. `base` for the base installation or `myenv` for a conda env with the name `myenv`.
- **UDF Modules:** Names of Python modules (without `.py` extension) from which the UDFs are being imported. Separate multiple modules by “;”. Example: `UDF_MODULES = "common_udfs;myproject"`  
The default imports a file in the same directory as the Excel spreadsheet with the same name but ending in `.py`.
- **Debug UDFs:** Check this box if you want to run the xlwings COM server manually for debugging, see [Debugging](#).
- **RunPython: Use UDF Server:** Uses the same COM Server for RunPython as for UDFs. This will be faster, as the interpreter doesn’t shut down after each call.
- **Restart UDF Server:** This restarts the UDF Server/Python interpreter.
- **Show Console:** Check the box in the ribbon or set the config to `TRUE` if you want the command prompt to pop up. This currently only works on Windows.

### 7.3.1 Anaconda/Miniconda

If you use Anaconda or Miniconda on Windows, you will need to set your `Conda Path` and `Conda Env` settings, as you will otherwise get errors when using NumPy etc. In return, leave `Interpreter` empty.

## 7.4 Environment Variables

With environment variables, you can set dynamic paths e.g. to your interpreter or `PYTHONPATH`:

- On Windows, you can use all environment variables like so: `%USERPROFILE%\Anaconda`.
- On macOS, the following special variables are supported: `$HOME`, `$APPLICATIONS`, `$DOCUMENTS`, `$DESKTOP`.

## 7.5 User Config: Ribbon/Config File

The settings in the xlwings Ribbon are stored in a config file that can also be manipulated externally. The location is

- Windows: `.xlwings\xlwings.conf` in your home folder, that is usually `C:\Users\<username>`
- macOS: `~/Library/Containers/com.microsoft.Excel/Data/xlwings.conf`

The format is as follows (currently the keys are required to be all caps) - note the OS specific Interpreter settings!

```
"INTERPRETER_WIN", "C:\path\to\python.exe"
"INTERPRETER_MAC", "/path/to/python"
"PYTHONPATH", ""
"CONDA_PATH", ""
```

(continues on next page)

(continued from previous page)

```
"CONDA ENV", ""
"UDF MODULES", ""
"DEBUG UDFS", ""
"USE UDF SERVER", ""
"SHOW CONSOLE", ""
"ONEDRIVE_WIN", ""
"ONEDRIVE_MAC", ""
```

---

**Note:** The ONEDRIVE\_WIN/\_MAC setting has to be edited directly in the file, there is currently no possibility to edit it via the ribbon. Usually, it is only required if you are either on macOS or if your environment variables on Windows are not correctly set or if you have a private and corporate location and don't want to go with the default one. ONEDRIVE\_WIN/\_MAC has to point to the root folder of your local OneDrive folder.

---

## 7.6 Workbook Directory Config: Config file

The global settings of the Ribbon/Config file can be overridden for one or more workbooks by creating a `xlwings.conf` file in the workbook's directory.

---

**Note:** Workbook directory config files are not supported if your workbook is stored on SharePoint.

---

## 7.7 Workbook Config: xlwings.conf Sheet

Workbook specific settings will override global (Ribbon) and workbook directory config files: Workbook specific settings are set by listing the config key/value pairs in a sheet with the name `xlwings.conf`. When you create a new project with `xlwings quickstart`, it'll already have such a sheet but you need to rename it to `xlwings.conf` to make it active.

	A	B	
1	Interpreter	pythonw	
2	PYTHONPATH		
3	UDF Modules		
4	Debug UDFs	FALSE	
5	Log File		
6	Use UDF Server	FALSE	
-			

## 7.8 Alternative: Standalone VBA module

Sometimes, it might be useful to run xlwings code without having to install an add-in first. To do so, you need to use the `standalone` option when creating a new project: `xlwings quickstart myproject --standalone`.

This will add the content of the add-in as a single VBA module so you don't need to set a reference to the add-in anymore. It will also include `Dictionary.cls` as this is required on macOS. It will still read in the settings from your `xlwings.conf` if you don't override them by using a sheet with the name `xlwings.conf`.



## RUNPYTHON

### 8.1 xlwings add-in

To get access to `Run main` (new in v0.16) button or the `RunPython` VBA function, you'll need the `xlwings` addin (or VBA module), see *Add-in & Settings*.

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

```
$ xlwings quickstart myproject
```

### 8.2 Call Python with “RunPython”

In the VBA Editor (Alt-F11), write the code below into a VBA module. `xlwings quickstart` automatically adds a new module with a sample call. If you rather want to start from scratch, you can add a new module via `Insert > Module`.

```
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 `Use UDF Server = True`.

---

## 8.3 Function Arguments and Return Values

While it's technically possible to include arguments in the function call within `RunPython`, it's not very convenient. Also, `RunPython` does not allow you to return values. To overcome these issues, use UDFs, see *User Defined Functions (UDFs)* - however, this is currently limited to Windows only.

## USER DEFINED FUNCTIONS (UDFS)

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*.
- 

### 9.1 One-time Excel preparations

- 1) Enable Trust access to the VBA project object model under File > Options > Trust Center > Trust Center Settings > Macro Settings. You only need to do this once. Also, this is only required for importing the functions, i.e. end users won't need to bother about this.
- 2) Install the add-in via command prompt: `xlwings addin install` (see *Add-in & Settings*).

### 9.2 Workbook preparation

The easiest way to start a new project is to run `xlwings quickstart myproject` on a command prompt (see *Command Line Client (CLI)*). This automatically adds the xlwings reference to the generated workbook.

## 9.3 A simple UDF

The default addin 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`.

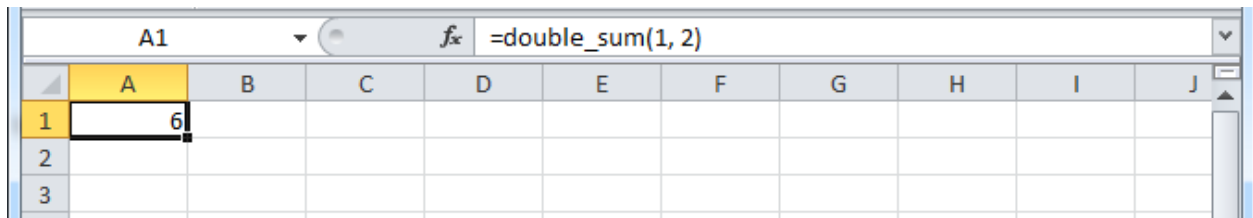
Alternatively, you can point to a specific module via **UDF Modules** in the xlwings ribbon.

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`.
- Enter the formula `=double_sum(1, 2)` into a cell and you will see the correct result:



- 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. If you want to make sure everything is in a fresh state, click **Restart UDF Server**.
  - 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.
-

## 9.4 Array formulas: Get efficient

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

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

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

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

To use this formula in Excel,

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

	A	B	C	D	E	F	G	H	I	J
1	1	2		2	3					
2	3	4		4	5					
3										

### 9.4.1 Number of array dimensions: ndim

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

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

```
@xw.func
@xw.arg('data', ndim=2)
def add_one(data):
    return [[cell + 1 for cell in row] for row in data]
```

## 9.5 Array formulas with NumPy and Pandas

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

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

```
import xlwings as xw
import numpy as np

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

---

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

---

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

```
import xlwings as xw
import pandas as pd

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

## 9.6 @xw.arg and @xw.ret decorators

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

```
@xw.func
@xw.arg('x', pd.DataFrame)
@xw.ret(index=False)
def myfunction(x):
```

(continues on next page)

(continued from previous page)

```
# x is a DataFrame, do something with it
return x
```

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

## 9.7 Dynamic Array Formulas

**Note:** If your version of Excel supports the new native dynamic arrays, then you don't have to do anything special, and you shouldn't use the `expand` decorator! To check if your version of Excel supports it, see if you have the `=UNIQUE()` formula available. Native dynamic arrays were introduced in Office 365 Insider Fast at the end of September 2018.

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**. In practice, it often turns out to be a cumbersome process, 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))
```

**Note:**

- Expanding array formulas will overwrite cells without prompting
- Pre v0.15.0 doesn't allow to have volatile functions as arguments, e.g. you cannot use functions like `=TODAY()` as arguments. Starting with v0.15.0, you can use volatile functions as input, but the UDF will be called more than 1x.
- Dynamic Arrays have been refactored with v0.15.0 to be proper legacy arrays: To edit a dynamic array with xlwings `>= v0.15.0`, you need to hit **Ctrl-Shift-Enter** while in the top left cell. Note that you don't have to do that when you enter the formula for the first time.

File Home Insert Page Layout Formulas Data Review					
B4		✕ ✓ <i>fx</i>		=dynamic_array(B2,C2)	
	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					

File Home Insert Page Layout Formulas Data Review View xlwings							
B4		✕ ✓ <i>fx</i>		=dynamic_array(B2,C2)			
	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							

## 9.8 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)
```

## 9.9 The “caller” argument

You often need to know which cell called the UDF. For this, xlwings offers the reserved argument `caller` which returns the calling cell as xlwings range object:

```
@xw.func
def get_caller_address(caller):
    # caller will not be exposed in Excel, so use it like so:
    # =get_caller_address()
    return caller.address
```

Note that `caller` will not be exposed in Excel but will be provided by xlwings behind the scenes.

## 9.10 The “vba” keyword

By using the `vba` keyword, you can get access to any Excel VBA object in the form of a `pywin32` object. For example, if you wanted to pass the sheet object in the form of its `CodeName`, you can do it as follows:

```
@xw.func
@xw.arg('sheet1', vba='Sheet1')
def get_name(sheet1):
    # call this function in Excel with:
    # =get_name()
    return sheet1.Name
```

Note that `vba` arguments are not exposed in the UDF but automatically provided by xlwings.

## 9.11 Macros

On Windows, as an 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`.

## 9.12 Call UDFs from VBA

Imported functions can also be used from VBA. For example, for a function returning a 2d array:

```
Sub MySub()

Dim arr() As Variant
Dim i As Long, j As Long

    arr = my_imported_function(...)

    For j = LBound(arr, 2) To UBound(arr, 2)
        For i = LBound(arr, 1) To UBound(arr, 1)
            Debug.Print "(" & i & "," & j & ")", arr(i, j)
        Next i
    Next j

End Sub
```

## 9.13 Asynchronous UDFs

---

**Note:** This is an experimental feature

---

New in version v0.14.0.

xlwings offers an easy way to write asynchronous functions in Excel. Asynchronous functions return immediately with #N/A `waiting...`. While the function is waiting for its return value, you can use Excel to do other stuff and whenever the return value is available, the cell value will be updated.

The only available mode is currently `async_mode='threading'`, meaning that it's useful for I/O-bound tasks, for example when you fetch data from an API over the web.

You make a function asynchronous simply by giving it the respective argument in the function decorator. In this example, the time consuming I/O-bound task is simulated by using `time.sleep`:

```
import xlwings as xw
import time

@xw.func(async_mode='threading')
def myfunction(a):
    time.sleep(5)  # long running tasks
    return a
```

You can use this function like any other xlwings function, simply by putting `=myfunction("abcd")` into a cell (after you have imported the function, of course).

Note that xlwings doesn't use the native asynchronous functions that were introduced with Excel 2010, so xlwings asynchronous functions are supported with any version of Excel.



## MATPLOTLIB & PLOTLY CHARTS

### 10.1 Matplotlib

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

#### 10.1.1 Getting started

The easiest sample boils down to:

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

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

sheet = xw.Book().sheets[0]
sheet.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.

---

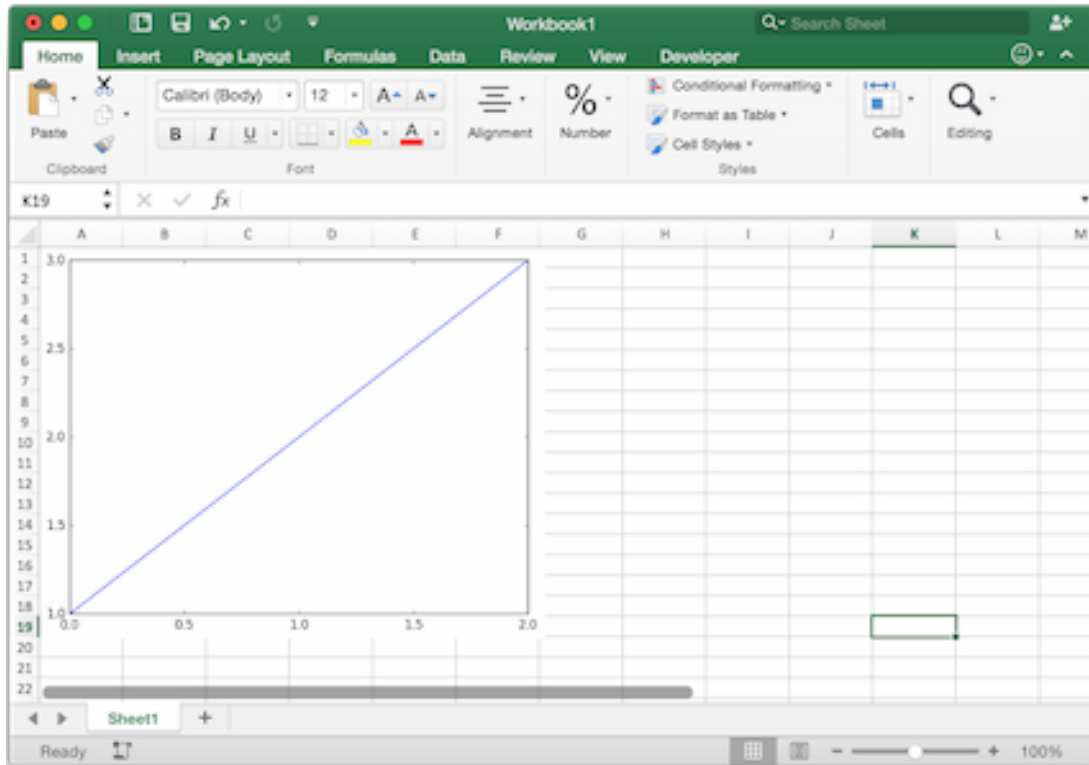
#### 10.1.2 Full integration with Excel

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

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

```
@xw.func
def myplot(n, caller):
    fig = plt.figure()
```

(continues on next page)



(continued from previous page)

```
plt.plot(range(int(n)))
caller.sheet.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:

### 10.1.3 Properties

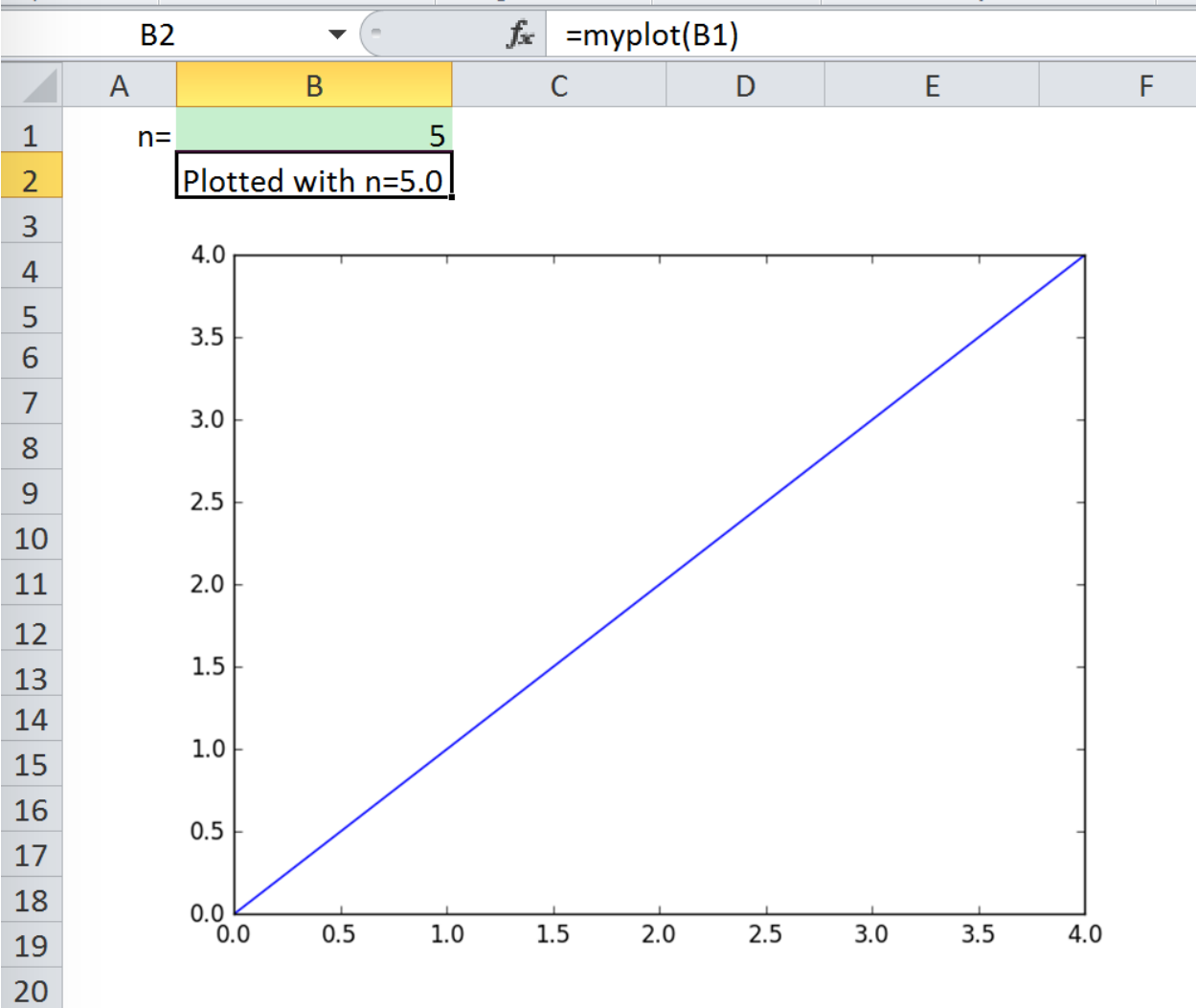
Size, position and other properties can either be set as arguments within `pictures.add()`, or by manipulating the picture object that is returned, see `xlwings.Picture()`.

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
```



## 10.1.4 Getting a Matplotlib figure

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

- via PyPlot interface:

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

or:

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

- via object oriented interface:

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

- via Pandas:

```
import pandas as pd
import numpy as np

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

## 10.2 Plotly static charts

### 10.2.1 Prerequisites

In addition to plotly, you will need kaleido, psutil, and requests. The easiest way to get it is via pip:

```
$ pip install kaleido psutil requests
```

or conda:

```
$ conda install -c conda-forge python-kaleido psutil requests
```

See also: <https://plotly.com/python/static-image-export/>

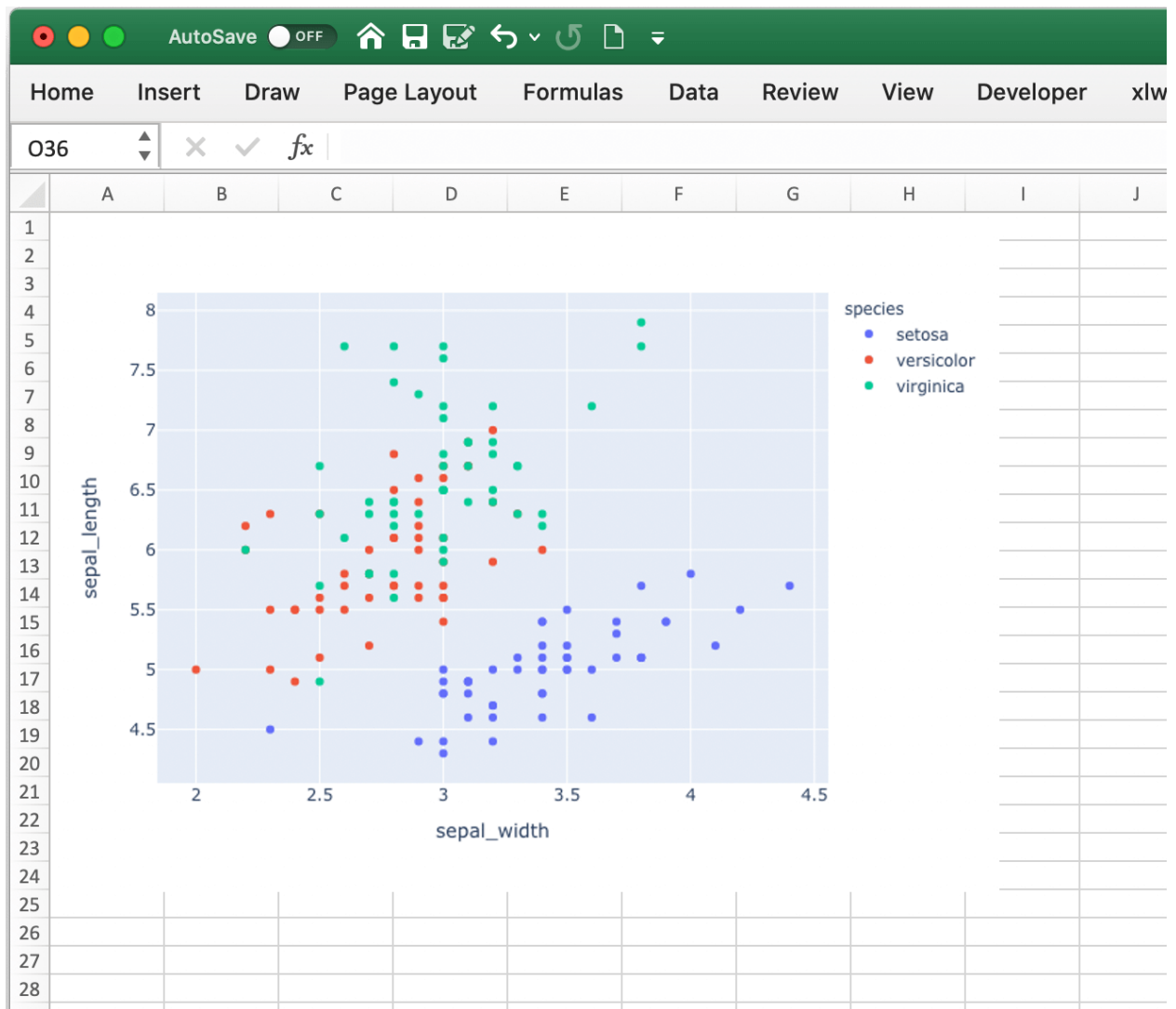
## 10.2.2 How to use

It works the same as with Matplotlib, however, rendering a Plotly chart takes slightly longer. Here is a sample:

```
import xlwings as xw
import plotly.express as px

# Plotly chart
df = px.data.iris()
fig = px.scatter(df, x="sepal_width", y="sepal_length", color="species")

# Add it to Excel
wb = xw.Book()
wb.sheets[0].pictures.add(fig, name='IrisScatterPlot', update=True)
```





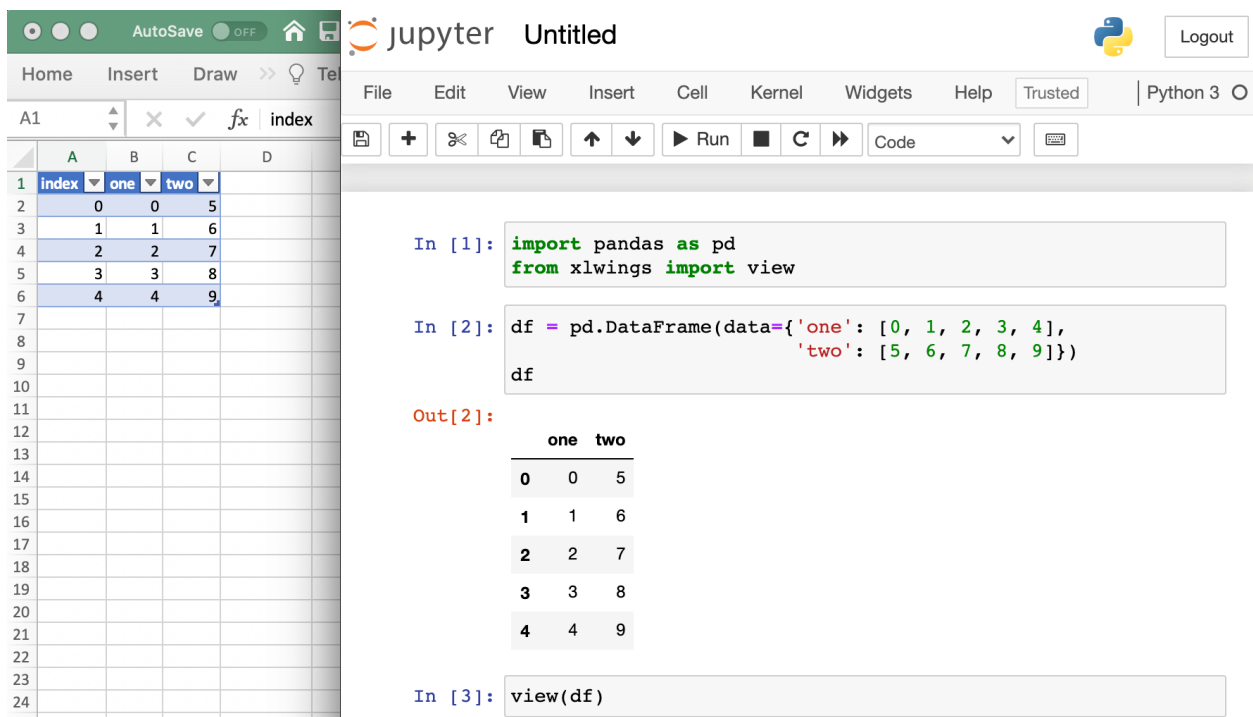
## JUPYTER NOTEBOOKS: INTERACT WITH EXCEL

When you work with Jupyter notebooks, you may use Excel as an interactive data viewer or scratchpad from where you can load DataFrames. The two convenience functions `view` and `load` make this really easy.

**Note:** The `view` and `load` functions should exclusively be used for interactive work. If you write scripts, use the `xlwings` API as introduced under *Quickstart* and *Syntax Overview*.

### 11.1 The view function

The `view` function accepts pretty much any object of interest, whether that's a number, a string, a nested list or a NumPy array or a pandas DataFrame. By default, it writes the data into an Excel table in a new workbook. If you wanted to reuse the same workbook, provide a sheet object, e.g. `view(df, sheet=xw.sheets.active)`, for further options see `view`.



```
In [1]: import pandas as pd
        from xlwings import view

In [2]: df = pd.DataFrame(data={'one': [0, 1, 2, 3, 4],
                                'two': [5, 6, 7, 8, 9]})
        df

Out[2]:
```

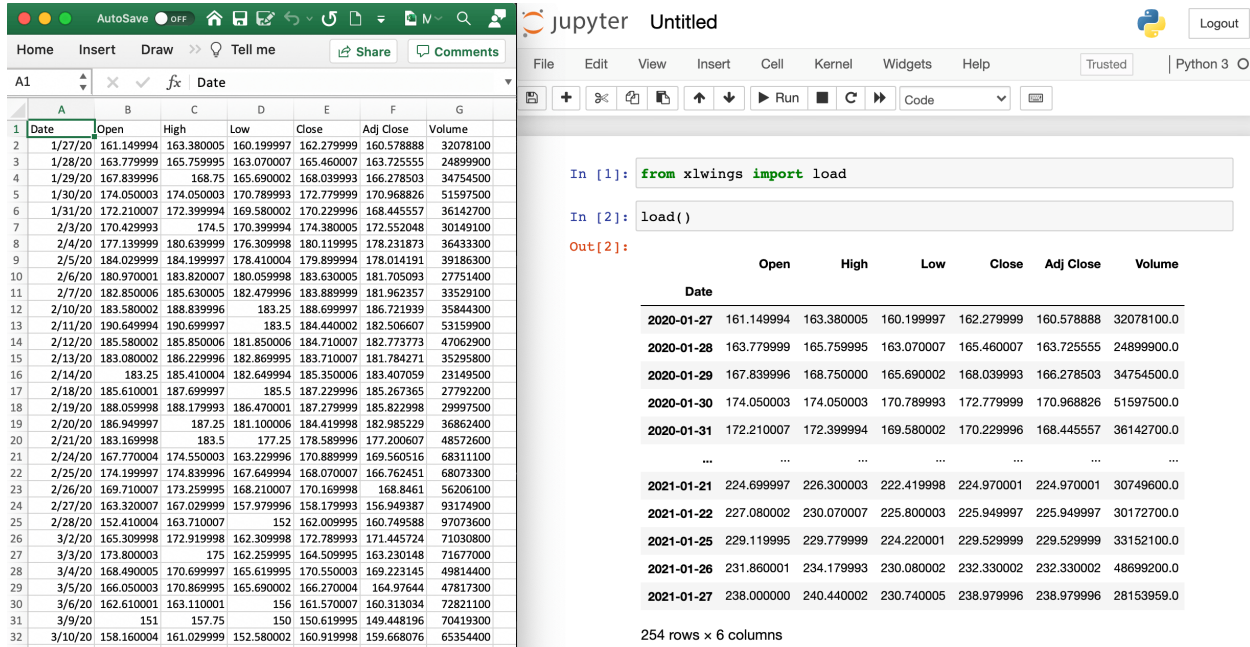
	one	two
0	0	5
1	1	6
2	2	7
3	3	8
4	4	9

```
In [3]: view(df)
```

Changed in version 0.22.0: Earlier versions were not formatting the output as Excel table

## 11.2 The load function

To load in a range in an Excel sheet as pandas DataFrame, use the `load` function. If you only select one cell, it will auto-expand to cover the whole range. If, however, you select a specific range that is bigger than one cell, it will load in only the selected cells. If the data in Excel does not have an index or header, set them to `False` like this: `xw.load(index=False)`, see also [load](#).



The screenshot shows a Jupyter Notebook interface with a Jupyter Notebook titled 'Untitled'. The left pane displays an Excel spreadsheet with columns A through G. The right pane shows the Jupyter Notebook code and output.

**Code:**

```
In [1]: from xlwings import load

In [2]: load()
```

**Output:**

```
Out[2]:
```

Date	Open	High	Low	Close	Adj Close	Volume
2020-01-27	161.149994	163.380005	160.199997	162.279999	160.578888	32078100.0
2020-01-28	163.779999	165.759995	163.070007	165.460007	163.725555	24899900.0
2020-01-29	167.839996	168.750000	165.690002	168.039993	166.278503	34754500.0
2020-01-30	174.050003	174.050003	170.789993	172.779999	170.968826	51597500.0
2020-01-31	172.210007	172.399994	169.580002	170.229996	168.445557	36142700.0
...	...	...	...	...	...	...
2021-01-21	224.699997	226.300003	222.419998	224.970001	224.970001	30749600.0
2021-01-22	227.080002	230.070007	225.800003	225.949997	225.949997	30172700.0
2021-01-25	229.119995	229.779999	224.220001	229.529999	229.529999	33152100.0
2021-01-26	231.860001	234.179993	230.080002	232.330002	232.330002	48699200.0
2021-01-27	238.000000	240.440002	230.740005	238.979996	238.979996	28153959.0

254 rows x 6 columns

New in version 0.22.0.

## COMMAND LINE CLIENT (CLI)

xlwings comes with a command line client. On Windows, type the commands into a Command Prompt or Anaconda Prompt, on Mac, type them into a Terminal. To get an overview of all commands, simply type `xlwings` and hit Enter:

<code>addin</code>	Run <code>"xlwings addin install"</code> to install the Excel add-in (will be copied to the XLSTART folder). Instead of "install" you can also use "update", "remove" or "status". Note that this command may take a while. Use the <code>"--unprotected"</code> flag to install the add-in without password protection. You can install your custom add-in by providing the name or path via the <code>--file</code> flag, e.g. <code>"xlwings add-in install --file custom.xlam"</code> (New in 0.6.0, the unprotected flag was added in 0.20.4)
<code>quickstart</code>	Run <code>"xlwings quickstart myproject"</code> to create a folder called "myproject" in the current directory with an Excel file and a Python file, ready to be used. Use the <code>"--standalone"</code> flag to embed all VBA code in the Excel file and make it work without the xlwings add-in.
<code>runpython</code>	macOS only: run <code>"xlwings runpython install"</code> if you want to enable the RunPython calls without installing the add-in. This will create the following file: ~/Library/Application Scripts/com.microsoft.Excel/xlwings.applescript (new in 0.7.0)
<code>restapi</code>	Use <code>"xlwings restapi run"</code> to run the xlwings REST API via Flask dev server. Accepts <code>"--host"</code> and <code>"--port"</code> as optional arguments.
<code>license</code>	xlwings PRO: Use <code>"xlwings license update -k KEY"</code> where "KEY" is your personal (trial) license key. This will update <code>~/.xlwings/xlwings.conf</code> with the <code>LICENSE_KEY</code> entry. If you have a paid license, you can run <code>"xlwings license deploy"</code> to create a deploy key. This is not available for trial keys.
<code>config</code>	Run <code>"xlwings config create"</code> to create the user config

(continues on next page)

(continued from previous page)

	<p>file (<code>~/xlwings/xlwings.conf</code>) which is where the settings from the Ribbon add-in are stored. It will configure the Python interpreter that you are running this command with. To reset your configuration, run this with the <code>--force</code> flag which will overwrite your current configuration.</p> <p>(New in 0.19.5)</p>
code	<p>Run <code>"xlwings code embed"</code> to embed all Python modules of the workbook's dir in your active Excel file. Use the <code>--file</code> flag to only import a single file by providing its path. Requires xlwings PRO.</p> <p>(Changed in 0.23.4)</p>
permission	<p><code>"xlwings permission cwd"</code> prints a JSON string that can be used to permission the execution of all modules in the current working directory via GET request.</p> <p><code>"xlwings permission book"</code> does the same for code that is embedded in the active workbook.</p> <p>(New in 0.23.4)</p>
release	<p>Run <code>"xlwings release"</code> to configure your active workbook to work with a one-click installer for easy deployment. Requires xlwings PRO.</p> <p>(New in 0.23.4)</p>

## DEPLOYMENT

### 13.1 Zip files

New in version 0.15.2.

To make it easier to distribute, you can zip up your Python code into a zip file. If you use UDFs, this will disable the automatic code reload, so this is a feature meant for distribution, not development. In practice, this means that when your code is inside a zip file, you'll have to click on re-import to get any changes.

If you name your zip file like your Excel file (but with `.zip` extension) and place it in the same folder as your Excel workbook, xlwings will automatically find it (similar to how it works with a single python file).

If you want to use a different directory, make sure to add it to the `PYTHONPATH` in your config (Ribbon or config file):

```
PYTHONPATH, "C:\path\to\myproject.zip"
```

### 13.2 RunFrozenPython

Changed in version 0.15.2.

You can use a freezer like PyInstaller, cx\_Freeze, py2exe etc. to freeze your Python module into an executable so that the recipient doesn't have to install a full Python distribution.

---

**Note:**

- This does not work with UDFs.
  - Currently only available on Windows, but support for Mac should be easy to add.
  - You need at least 0.15.2 to support arguments whereas the syntax changed in 0.15.6
- 

Use it as follows:

```
Sub MySample()  
    RunFrozenPython "C:\path\to\dist\myproject\myproject.exe", "arg1 arg2"  
End Sub
```



## ONEDRIVE AND SHAREPOINT

Since v0.25.0, `xlwings` works with files that are stored on OneDrive, OneDrive for Business, and SharePoint—as long as they are synced locally. Some constellations will work out-of-the-box, while others require you to edit the configuration via the `xlwings.conf` file (see [User Config](#)) or the workbook’s `xlwings.conf` sheet (see [Workbook Config](#)).

**Warning: `xlwings` only works with OneDrive and SharePoint files that are synced to a local folder!**  
This means that both, the Excel and Python file, need to show the green check mark in the File Explorer/Finder as status—a cloud icon will not work.

### 14.1 OneDrive (Personal)

Default setups work out-of-the-box on Windows and macOS. If you get an error message, add the following setting with the correct path to the local root directory of your OneDrive. If possible, make use of environment variables (as shown in the examples) so the configuration will work across different users with the same setup:

- **Windows** (Example):

ONEDRIVE_CONSUMER_WIN	%USERPROFILE%\OneDrive
-----------------------	------------------------

- **macOS** (Example):

ONEDRIVE_CONSUMER_MAC	\$HOME/OneDrive
-----------------------	-----------------

## 14.2 OneDrive for Business

- **Windows:** Default setups work out-of-the-box. If you get an error message, add the following setting with the correct path to the local root directory of your OneDrive for Business. If possible, make use of environment variables (as shown in the examples) so the configuration will work across different users with the same setup:

ONEDRIVE_COMMERCIAL_WIN	%USERPROFILE%\OneDrive - My Company LLC
-------------------------	---

- **macOS:** macOS *always* requires the following setting with the correct path to the local root directory of your OneDrive for Business. If possible, make use of environment variables (as shown in the examples) so the configuration will work across different users with the same setup:

ONEDRIVE_COMMERCIAL_MAC	\$HOME/OneDrive - My Company LLC
-------------------------	----------------------------------

## 14.3 SharePoint (Online and On-Premises)

On Windows, the location of the local root folder of SharePoint can *sometimes* be derived from the OneDrive environment variables. Most of the time though, you'll have to provide the following setting (on macOS this is a must):

- **Windows:**

SHAREPOINT_WIN	%USERPROFILE%\My Company LLC
----------------	------------------------------

- **macOS:**

SHAREPOINT_MAC	\$HOME/My Company LLC
----------------	-----------------------

## 14.4 Implementation Details & Limitations

A lot of the xlwings functionality depends on the workbook's `FullName` property (via VBA/COM) that returns the local path of the file unless it is saved on OneDrive, OneDrive for Business or SharePoint **with AutoSave enabled**. In this case, it returns a URL instead.

URLs for OneDrive and OneDrive for Business can be translated fairly straight forward to the local equivalent. You will need to know the root directory of the local drive though: on Windows, these are usually provided via environment variables for OneDrive. On macOS they don't exist, which is the reason why you

need to provide the root directory for OneDrive. On Windows, the root directory for SharePoint can sometimes be derived from the env vars, too, but this is not guaranteed. On macOS, you'll need to provide it always anyway.

SharePoint, unfortunately, allows you to map the drives locally in any way you want and there's no way to reliably get the local path for these files. xlwings therefore first checks if the local path is mapped by using the defaults and if the file can't be found, it checks all existing local files on SharePoint. If it finds one with the same name, it'll use this. If, however, it finds more than one with the same name, you will get an error message. In this case, you can either rename the file to something unique across all the locally synced SharePoint files or you can change the `SHAREPOINT_WIN/MAC` setting to not stop at the root folder but include additional folders. As an example, assume you have the following file structure on your local SharePoint:

```
My Company LLC/  
└─ sitename1/  
    └─ myfile.xlsx  
└─ sitename2 - Documents/  
    └─ myfile.xlsx
```

In this case, you could either rename one of the files, or you could add a path that goes beyond the root folder (preferably under the `xlwings.conf` sheet):

SHAREPOINT_WIN	%USERPROFILE%/My Company LLC/sitename2 - Documents
----------------	--



## TROUBLESHOOTING

### 15.1 Issue: dll not found

Solution:

- 1) `xlwings32-<version>.dll` and `xlwings64-<version>.dll` are both in the same directory as your `python.exe`. If not, something went wrong with your installation. Reinstall it with `pip` or `conda`, see *Installation*.
- 2) Check your Interpreter in the add-in or config sheet. If it is empty, then you need to be able to open a windows command prompt and type `python` to start an interactive Python session. If you get the error '`python`' is not recognized as an internal or external command, operable program or batch file., then you have two options: Either add the path of where your `python.exe` lives to your Windows path (see <https://www.computerhope.com/issues/ch000549.htm>) or set the full path to your interpreter in the add-in or your config sheet, e.g. `C:\Users\MyUser\anaconda\pythonw.exe`

### 15.2 Issue: Files that are saved on OneDrive or SharePoint cause an error to pop up

Solution:

See the dedicated page about how to configure OneDrive and Sharepoint: *OneDrive and SharePoint*.



## 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
```

## 16.1 Default Converter

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

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

The following options can be set:

- **ndim**

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

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

---

- **chunksize**

When you read and write from or to big ranges, you may have to chunk them or you will hit a timeout or a memory error. The ideal chunksize will depend on your system and size of the array, so you will have to try out a few different chunksizes to find one that works well:

```
import pandas as pd
import numpy as np
sheet = xw.Book().sheets[0]
data = np.arange(75_000 * 20).reshape(75_000, 20)
df = pd.DataFrame(data=data)
sheet['A1'].options(chunksize=10_000).value = df
```

And the same for reading:

```
# As DataFrame
df = sheet['A1'].expand().options(pd.DataFrame, chunksize=10_000).value
# As list of list
df = sheet['A1'].expand().options(chunksize=10_000).value
```

## 16.2 Built-in Converters

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

It is also possible to write and register a 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.

### 16.2.1 Dictionary converter

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

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

```
>>> 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}
```

Note: instead of `dict`, you can also use `OrderedDict` from `collections`.

### 16.2.2 Numpy array converter

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

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

**Example:**

```
>>> import numpy as np
>>> 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.]])
```

### 16.2.3 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
```

## 16.2.4 Pandas DataFrame converter

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

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

### index: int or Boolean

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

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

### header: int or Boolean

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

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

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

### Example:

	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
```

(continues on next page)

(continued from previous page)

```

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

```

## 16.2.5 xw.Range and 'raw' converters

Technically speaking, these are “no-converters”.

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

```

@xw.func
@xw.arg('x', '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))

```

## 16.3 Custom Converter

Here are the steps to implement your own converter:

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

The `options` dictionary will contain all keyword arguments specified in the `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):
```

(continues on next page)

(continued from previous page)

```

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 lists (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.

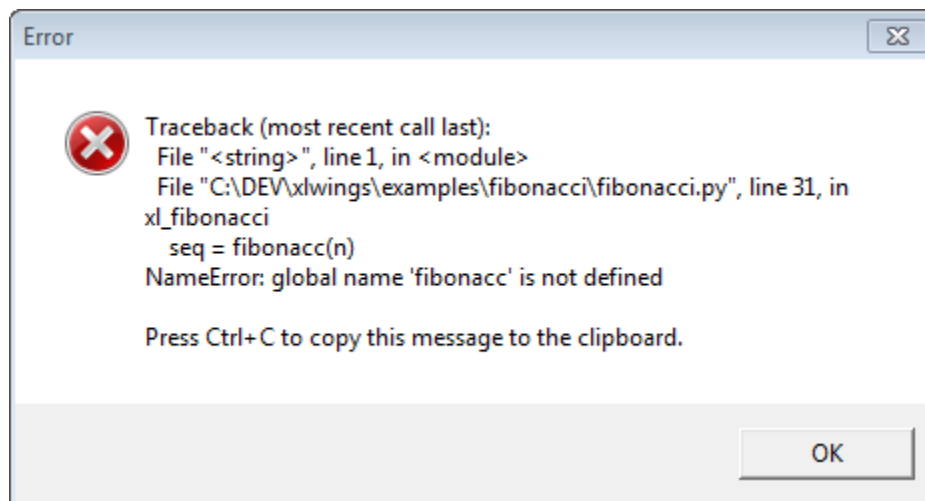


## DEBUGGING

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

- **RunPython:** When calling Python through RunPython, you can set a `mock_caller` to make it easy to switch back and forth between calling 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 (`/Users/<User>/Library/Containers/com.microsoft.Excel/Data/xlwings.log`).

---

## 17.1 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.xlsm').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
```

## 17.2 UDF debug server

Windows only: To debug UDFs, just check the `Debug UDFs` in the *Add-in & 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()
```

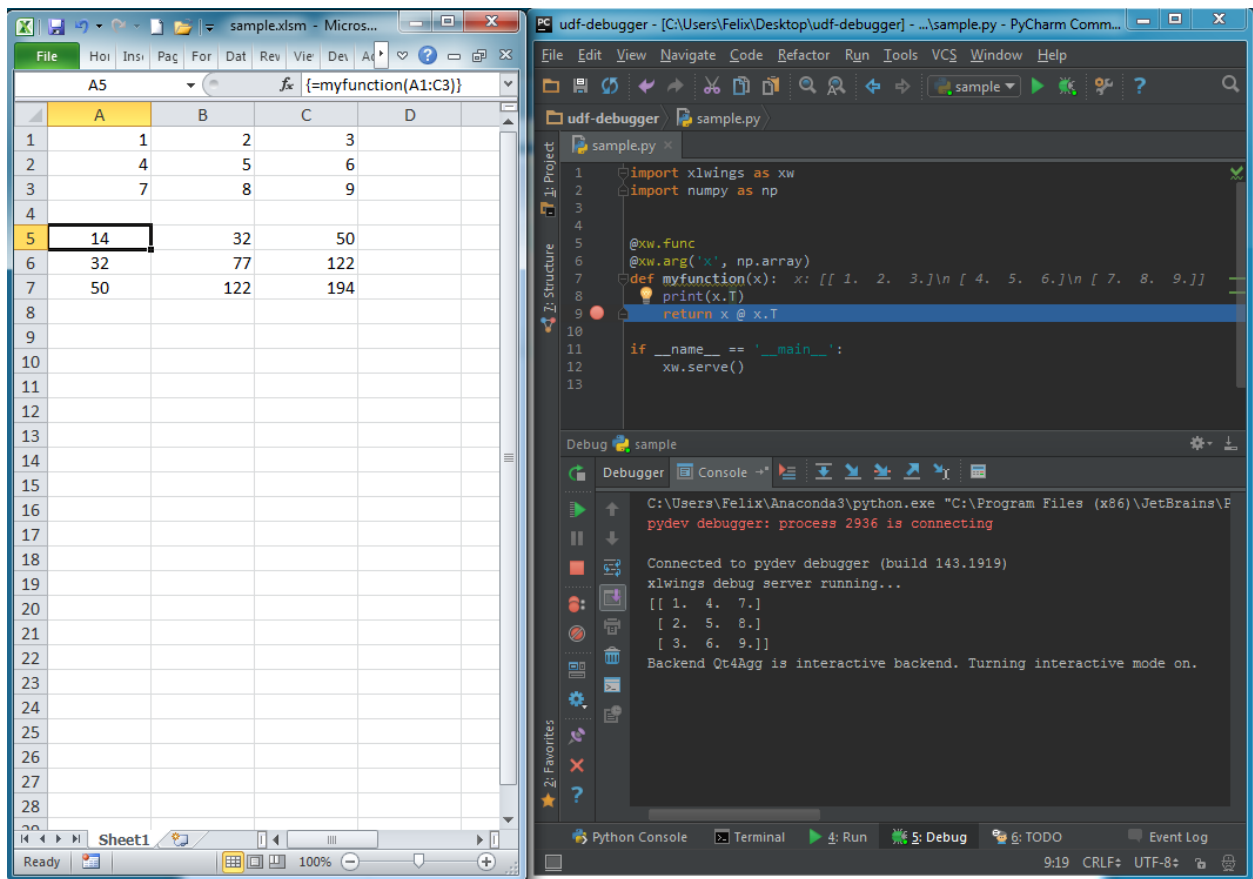
When you recalculate the Sheet (Ctrl-Alt-F9), the code will stop at breakpoints or output any print calls that you may have.

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.

---





## EXTENSIONS

It's easy to extend the xlwings add-in with own code like UDFs or RunPython macros, so that they can be deployed without end users having to import or write the functions themselves. Just add another VBA module to the xlwings addin with the respective code.

UDF extensions can be used from every workbook without having to set a reference.

### 18.1 In-Excel SQL

The xlwings addin comes with a built-in extension that adds in-Excel SQL syntax (sqlite dialect):

```
=sql(SQL Statement, table a, table b, ...)
```

As this extension uses UDFs, it's only available on Windows right now.

A16
:
✖
✔
fx
=sql(A14,A1:D11,G1:H8)

	A	B	C	D	E	F	G	H
1	<b>id</b>	<b>first_name</b>	<b>last_name</b>	<b>age</b>			<b>id</b>	<b>email</b>
2	1	Mariam	Alt	12			1	Mariam@Alt
3	2	Shenita	Truelove	55			2	Shenita@Truelove
4	3	Evelyn	Braddy	30			3	Evelyn@Braddy
5	4	Shery	Sam	35			5	Rogello@Mote
6	5	Rogello	Mote	88			6	Solomon@Okamura
7	6	Solomon	Okamura	33			8	Latashia@Alire
8	7	Jessica	Buelow	10			9	Roselee@Tarwater
9	8	Latashia	Alire	19				
10	9	Roselee	Tarwater	28				
11	10	Kiera	Saulsbury	55				
12								
13								
14	SELECT a.id, a.first_name, a.last_name, b.email FROM a INNER JOIN b ON a.id = b.id							
15								
16	<b>id</b>	<b>first_name</b>	<b>last_name</b>	<b>email</b>				
17	1	Mariam	Alt	Mariam@Alt				
18	2	Shenita	Truelove	Shenita@Truelove				
19	3	Evelyn	Braddy	Evelyn@Braddy				
20	5	Rogello	Mote	Rogello@Mote				
21	6	Solomon	Okamura	Solomon@Okamura				
22	8	Latashia	Alire	Latashia@Alire				
23	9	Roselee	Tarwater	Roselee@Tarwater				

## CUSTOM ADD-INS

New in version 0.22.0.

Custom add-ins work on Windows and macOS and are white-labeled xlwings add-ins that include all your RunPython functions and UDFs (as usual, UDFs work on Windows only). You can build add-ins with and without an Excel ribbon.

The useful thing about add-in is that UDFs and RunPython calls will be available in all workbooks right out of the box without having to add any references via the VBA editor's Tools > References.... You can also work with standard `xlsx` files rather than `xlsm` files. This tutorial assumes you're familiar with how xlwings and its configuration works.

### 19.1 Quickstart

Start by running the following command on a command line (to create an add-in without a ribbon, you would leave away the `--ribbon` flag):

```
$ xlwings quickstart myproject --addin --ribbon
```

This will create the familiar quickstart folder with a Python file and an Excel file, but this time, the Excel file is in the `xlam` format.

- Double-click the Excel add-in to open it in Excel
- Add a new empty workbook (Ctrl+N on Windows or Command+N on macOS)

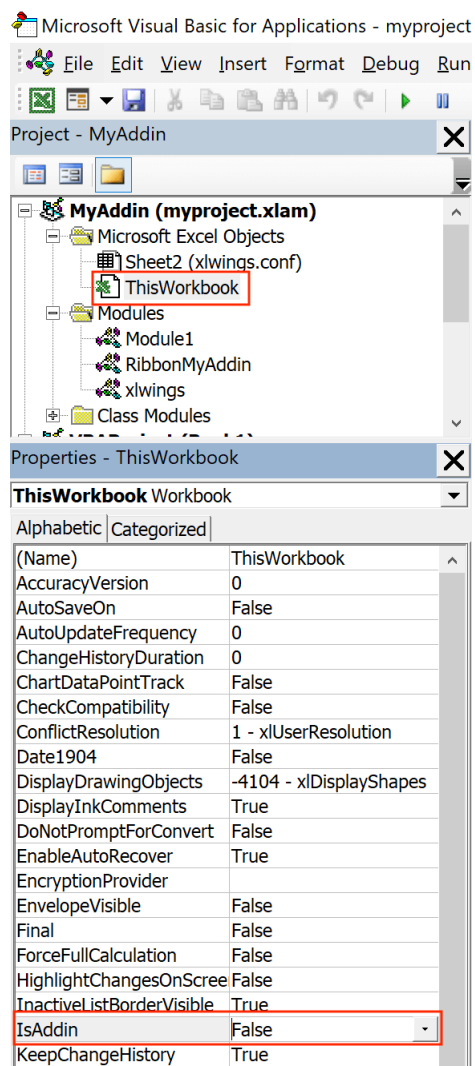
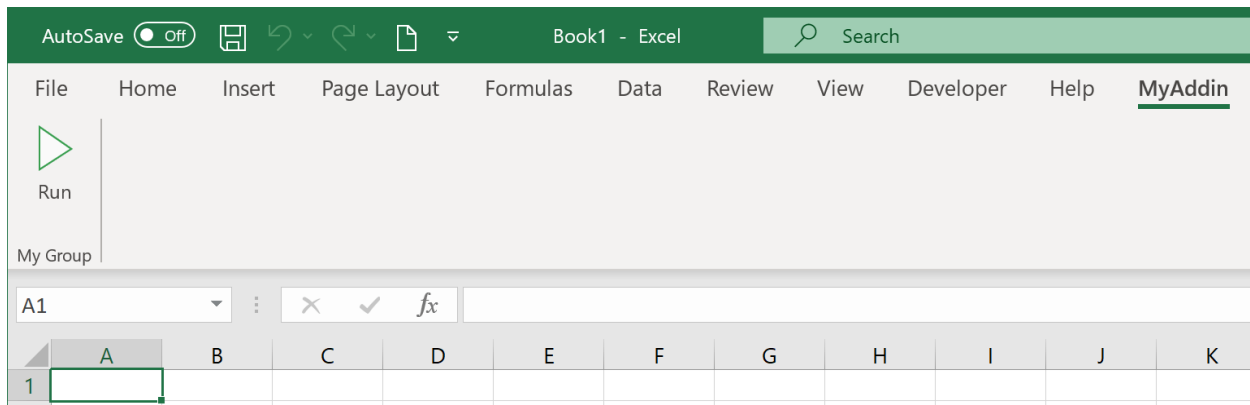
You should see a new ribbon tab called `MyAddin` like this:

The add-in and VBA project are currently always called `myaddin`, no matter what name you chose in the quickstart command. We'll see towards the end of this tutorial how we can change that, but for now we'll stick to it.

Compared to the xlwings add-in, the custom add-in offers an additional level of configuration: the configuration sheet of the add-in itself which is the easiest way to configure simple add-ins with a static configuration.

Let's open the VBA editor by clicking on Alt+F11 (Windows) or Option+F11 (macOS). In our project, select `ThisWorkbook`, then change the Property `IsAddin` from `True` to `False`, see the following screenshot:

This will make the sheet `_myaddin.conf` visible (again, we'll see how to change the name of `myaddin` at the end of this tutorial):



- Activate the sheet config by renaming it from `_myaddin.conf` to `myaddin.conf`
- Set your `Interpreter_Win/_Mac` or `Conda` settings (you may want to take them over from the `xlwings` settings for now)

Once done, switch back to the VBA editor, select `ThisWorkbook` again, and change `IsAddin` back to `True` before you save your add-in from the VBA editor. Switch back to Excel and click the `Run` button under the `My Addin` ribbon tab and if you've configured the Python interpreter correctly, it will print `Hello xlwings!` into cell `A1` of the active workbook.

## 19.2 Changing the Ribbon menu

To change the buttons and items in the ribbon menu or the Backstage View, download and install the [Office RibbonX Editor](#). While it is only available for Windows, the created ribbons will also work on macOS. Open your add-in with it so you can change the XML code that defines your buttons etc. You will find a good tutorial [here](#). The callback function for the demo `Run` button is in the `RibbonMyAddin` VBA module that you'll find in the VBA editor.

## 19.3 Importing UDFs

To import your UDFs into the custom add-in, run the `ImportPythonUDFsToAddin` Sub towards the end of the `xlwings` module (click into the Sub and hit `F5`). Remember, you only have to do this whenever you change the function name, argument or decorator, so your end users won't have to deal with this.

If you are only deploying UDFs via your add-in, you probably don't need a Ribbon menu and can leave away the `--ribbon` flag in the `quickstart` command.

## 19.4 Configuration

As mentioned before, configuration works the same as with `xlwings`, so you could have your users override the default configuration we did above by adding a `myaddin.conf` sheet on their workbook or you could use the `myaddin.conf` file in the user's home directory. For details see [Add-in & Settings](#).

## 19.5 Installation

If you want to permanently install your add-in, you can do so by using the `xlwings` CLI:

```
$ xlwings addin install --file C:\path\to\your\myproject.xlam
```

This, however, means that you will need to adjust the `PYTHONPATH` for it to find your Python code (or move your Python code to somewhere where Python looks for it—more about that below under deployment). The command will copy your add-in to the `XLSTART` folder, a special folder from where Excel will open all files everytime you start it.

## 19.6 Renaming your add-in

Admittedly, this part is a bit cumbersome for now. Let's assume, we would like to rename the addin from `MyAddin` to `Demo`:

- In the xlwings VBA module, change `Public Const PROJECT_NAME As String = "myaddin"` to `Public Const PROJECT_NAME As String = "demo"`. You'll find this line at the top, right after the `Declare` statements.
- If you rely on the `myaddin.conf` sheet for your configuration, rename it to `demo.conf`
- Right-click the VBA project, select `MyAddin Properties...` and rename the `Project Name` from `MyAddin` to `Demo`.
- If you use the ribbon, you want to rename the `RibbonMyAddin` VBA module to `RibbonDemo`. To do this, select the module in the VBA editor, then rename it in the `Properties` window. If you don't see the `Properties` window, hit `F4`.
- Open the add-in in the Office RibbonX Editor (see above) and replace all occurrences of `MyAddin` with `Demo` in the XML code.

And finally, you may want to rename your `myproject.xlam` file in the Windows explorer, but I assume you have already run the `quickstart` command with the correct name, so this won't be necessary.

## 19.7 Deployment

By far the easiest way to deploy your add-in to your end-users is to build an installer via the xlwings `PRO` offering. This will take care of everything and your end users literally just need to double-click the installer and they are all set (no existing Python installation required and no manual installation of the add-in or adjusting of settings required).

If you want it the free (but hard) way, you either need to build an installer yourself or you need your users to install Python and the add-in and take care of placing the Python code in the correct directory. This normally involves tweaking the following settings, for example in the `myaddin.conf` sheet:

- `Interpreter_Win/_Mac`: if your end-users have a working version of Python, you can use environment variables to dynamically resolve to the correct path. For example, if they have Anaconda installed in the default location, you could use the following configuration:

<pre>Conda Path: %USERPROFILE%\anaconda3 Conda Env: base Interpreter_Mac: \$HOME/opt/anaconda3/bin/python</pre>
---

- `PYTHONPATH`: since you can't have your Python source code in the `XLSTART` folder next to the add-in, you'll need to adjust the `PYTHONPATH` setting and add the folder to where the Python code will be. You could point this to a shared drive or again make use of environment variables so the users can place the file into a folder called `MyAddin` in their home directory, for example. However, you can also place your Python code where Python looks for it, for example by placing them in the `site-packages` directory of the Python distribution—an easy way to achieve this is to build a Python package that you can install via `pip`.

## THREADING AND MULTIPROCESSING

New in version 0.13.0.

### 20.1 Threading

While xlwings is not technically thread safe, it's still easy to use it in threads as long as you have at least v0.13.0 and stick to a simple rule: Do not pass xlwings objects to threads. This rule isn't a requirement on macOS, but it's still recommended if you want your programs to be cross-platform.

Consider the following example that will **NOT** work:

```
import threading
from queue import Queue
import xlwings as xw

num_threads = 4

def write_to_workbook():
    while True:
        rng = q.get()
        rng.value = rng.address
        print(rng.address)
        q.task_done()

q = Queue()

for i in range(num_threads):
    t = threading.Thread(target=write_to_workbook)
    t.daemon = True
    t.start()

for cell in ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7', 'A8', 'A9', 'A10']:
    # THIS DOESN'T WORK - passing xlwings objects to threads will fail!
```

(continues on next page)

(continued from previous page)

```
rng = xw.Book('Book1.xlsx').sheets[0].range(cell)
q.put(rng)

q.join()
```

To make it work, you simply have to fully qualify the cell reference in the thread instead of passing a Book object:

```
import threading
from queue import Queue
import xlwings as xw

num_threads = 4

def write_to_workbook():
    while True:
        cell_ = q.get()
        xw.Book('Book1.xlsx').sheets[0].range(cell_).value = cell_
        print(cell_)
        q.task_done()

q = Queue()

for i in range(num_threads):
    t = threading.Thread(target=write_to_workbook)
    t.daemon = True
    t.start()

for cell in ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7', 'A8', 'A9', 'A10']:
    q.put(cell)

q.join()
```

## 20.2 Multiprocessing

---

**Note:** Multiprocessing is only supported on Windows!

---

The same rules apply to multiprocessing as for threading, here's a working example:

```
from multiprocessing import Pool
import xlwings as xw
```

(continues on next page)

(continued from previous page)

```
def write_to_workbook(cell):
    xw.Book('Book1.xlsx').sheets[0].range(cell).value = cell
    print(cell)

if __name__ == '__main__':
    with Pool(4) as p:
        p.map(write_to_workbook,
              ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7', 'A8', 'A9', 'A10'])
```



## MISSING FEATURES

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

- 1) Most importantly, open an issue on [GitHub](#). 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:

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

This works accordingly for the other objects like `sheet.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).

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

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

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



## XLWINGS WITH OTHER OFFICE APPS

xlwings can also be used to call Python functions from VBA within Office apps other than Excel (like Outlook, Access etc.).

---

**Note:** This is an experimental feature and may be removed in the future. Currently, this functionality is only available on Windows for UDFs. The RunPython functionality is currently not supported.

---

### 22.1 How To

- 1) As usual, write your Python function and import it into Excel (see *User Defined Functions (UDFs)*).
- 2) Press Alt-F11 to get into the VBA editor, then right-click on the xlwings\_udfs VBA module and select **Export File...** Save the xlwings\_udfs.bas file somewhere.
- 3) Switch into the other Office app, e.g. Microsoft Access and click again Alt-F11 to get into the VBA editor. Right-click on the VBA Project and **Import File...**, then select the file that you exported in the previous step. Once imported, replace the app name in the first line to the one that you are using, i.e. Microsoft Access or Microsoft Outlook etc. so that the first line then reads: `#Const App = "Microsoft Access"`
- 4) Now import the standalone xlwings VBA module (xlwings.bas). You can find it in your xlwings installation folder. To know where that is, do:

```
>>> import xlwings as xw
>>> xlwings.__path__
```

And finally do the same as in the previous step and replace the App name in the first line with the name of the corresponding app that you are using. You are now able to call the Python function from VBA.

## 22.2 Config

The other Office apps will use the same global config file as you are editing via the Excel ribbon add-in. When it makes sense, you'll be able to use the directory config file (e.g. you can put it next to your Access or Word file) or you can hardcode the path to the config file in the VBA standalone module, e.g. in the function `GetDirectoryConfigFilePath` (e.g. suggested when using Outlook that doesn't really have the same concept of files like the other Office apps). NOTE: For Office apps without file concept, you need to make sure that the `PYTHONPATH` points to the directory with the Python source file. For details on the different config options, see [Config](#).

## XLWINGS PRO OVERVIEW

The purpose of xlwings PRO is to fund the continued maintenance and enhancement of xlwings. This will allow you to rely on the package without being left with the dreaded “this library currently has no active maintainers” message that happens to too many open-source packages after a couple of years.

xlwings PRO offers access to additional functionality. All PRO features are marked with xlwings *PRO* in the docs.

---

**Note:** To get access to the additional functionality of xlwings PRO, you need a (trial) license key and at least xlwings v0.19.0. Everything under the `xlwings.pro` subpackage is distributed under a commercial license. To make use of xlwings PRO functionality beyond the trial, you will need to subscribe to one of our [paid plans](#).

---

### 23.1 PRO Features

- *One-click Installer*: Easily build your own Python installer including all dependencies—your end users don’t need to know anything about Python.
- *Embedded code*: Store your Python source code directly in Excel for easy deployment.
- *xlwings Reports*: A template-based reporting mechanism, allowing business users to change the layout of the report without having to touch the Python code.
- *Markdown Formatting*: Support for Markdown formatting of text in cells and shapes like e.g., text boxes.
- *Permissioning of Code Execution*: Control which users can run which Python modules via xlwings.

## 23.2 More Infos

- Pricing: <https://www.xlwings.org/pricing>
- Trial license key: <https://www.xlwings.org/trial>

## XLWINGS REPORTS

This feature requires xlwings *PRO*.

xlwings Reports is a solution for template-based Excel and PDF reporting, making the generation of pixel-perfect factsheets really simple. xlwings Reports allows business users without Python knowledge to create and maintain Excel templates without having to rely on a Python developer after the initial setup has been done: xlwings Reports separates the Python code (pre- and post-processing) from the Excel template (layout/formatting).

xlwings Reports supports all commonly required components:

- **Text:** Easily format your text via Markdown syntax.
- **Tables (dynamic):** Write pandas DataFrames to Excel cells and Excel tables and format them dynamically based on the number of rows.
- **Charts:** Use your favorite charting engine: Excel charts, Matplotlib, or Plotly.
- **Images:** You can include both raster (e.g., png) or vector (e.g., svg) graphics, including dynamically generated ones, e.g., QR codes or plots.
- **Multi-column Layout:** Split your content up into e.g. a classic two column layout by using Frames.
- **Single Template:** Generate reports in various languages, for various funds etc. based on a single template.
- **PDF Report:** Generate PDF reports automatically and “print” the reports on PDFs in your corporate layout for pixel-perfect results including headers, footers, backgrounds and borderless graphics.
- **Easy Pre-processing:** Since everything is based on Python, you can connect with literally any data source and clean it with pandas or some other library.
- **Easy Post-processing:** Again, with Python you’re just a few lines of code away from sending an email with the reports as attachment or uploading the reports to your web server, S3 bucket etc.

## 24.1 Quickstart

You can work on the sheet, book or app level:

- `mysheet.render_template(**data)`: replaces the placeholders in `mysheet`
- `mybook.render_template(**data)`: replaces the placeholders in all sheets of `mybook`
- `myapp.render_template(template, output, **data)`: convenience wrapper that copies a template book before replacing the placeholder with the values. Since this approach allows you to work with hidden Excel instances, it is the most commonly used method for production.

Let's go through a typical example: start by creating the following Python script `report.py`:

```
# report.py
from pathlib import Path

import pandas as pd
import xlwings as xw

# We'll place this file in the same directory as the Excel template
this_dir = Path(__file__).resolve().parent

data = dict(
    title='MyTitle',
    df=pd.DataFrame(data={'one': [1, 2], 'two': [3, 4]})
)

# Change visible=False to run this in a hidden Excel instance
with xw.App(visible=True) as app:
    book = app.render_template(this_dir / 'mytemplate.xlsx',
                              this_dir / 'myreport.xlsx',
                              **data)
    book.to_pdf(this_dir / 'myreport.pdf')
```

Then create the following Excel file called `mytemplate.xlsx`:

	A	B	C	D
1	{{ title }}			
2				
3	My DataFrame			
4	{{ df }}			
5				
6				

Run the Python script (or run the code from a Jupyter notebook):

```
python report.py
```

This will copy the template and create the following output by replacing the variables in double curly braces with the value from the Python variable:

	A	B
1	MyTitle	
2		
3	My DataFrame	
4	one	two
5	1	3
6	2	4
7		

If you like, you could also create a classic xlwings tool to call this script or you could design a GUI app by using a framework like PySimpleGUI and turn it into an executable by using a freezer (e.g., PyInstaller). This, however, is beyond the scope of this tutorial.

**Note:** By default, xlwings Reports overwrites existing values in templates if there is not enough free space for your variable. If you want your rows to dynamically shift according to the height of your array, use [Frames](#).

**Note:** Unlike xlwings, xlwings Reports never writes out the index of pandas DataFrames. If you need the index to appear in Excel, use `df.reset_index()`, see [DataFrames](#).

See also [render\\_templates](#) ([API reference](#)).

### 24.1.1 Render Books and Sheets

Sometimes, it's useful to render a single book or sheet instead of using the `myapp.render_template` method. This is a workbook stored as `Book1.xlsx`:

Running the following code:

```
import xlwings as xw

book = xw.Book('Book1.xlsx')
sheet = book.sheets['template'].copy(name='report')
```

(continues on next page)

	A	B	C
1	{{ title }}		
2			
3	{{ table }}		
4			
5			
6			
7			
8			

template +

(continued from previous page)

```
sheet.render_template(title='A Demo!', table=[[1, 2], [3, 4]])
book.to_pdf()
```

Copies the template sheet first and then fills it in:

	A	B	C
1	A Demo!		
2			
3	1	2	
4	3	4	
5			
6			
7			
8			

template report

See also the [mysheet.render\\_template \(API reference\)](#) and [mybook.render\\_template \(API reference\)](#).

New in version 0.22.0.

## 24.2 DataFrames

To write DataFrames in a consistent manner to Excel, xlwings Reports ignores the DataFrame indices. If you need to pass the index over to Excel, reset the index before passing in the DataFrame to `render_template` or `render_template`: `df.reset_index()`.

When working with pandas DataFrames, the report designer often needs to tweak the data. Thanks to filters, they can do the most common operations directly in the template without the need to write Python code. A filter is added to the placeholder in Excel by using the pipe character: `{{ myplaceholder | myfilter }}`. You can combine multiple filters by using multiple pipe characters: they are applied from left to right,

i.e. the result from the first filter will be the input for the next filter. Let's start with an example before listing each filter with its details:

```
import xlwings as xw
import pandas as pd

book = xw.Book('Book1.xlsx')
sheet = book.sheets['template'].copy(name='report')
df = pd.DataFrame({'one': [1, 2, 3], 'two': [4, 5, 6], 'three': [7, 8, 9]})
sheet.render_template(df=df)
```

	A	B	C	D
1	{{ df }}			
2				
3				
4				
5				
6				
7	{{ df   noheader }}			
8				
9				
10				
11				
12				
13	{{ df   sortdesc(1)   columns(0, None, 1) }}			
14				
15				
16				
17				

	A	B	C	D
1	one	two	three	
2	1	4	7	
3	2	5	8	
4	3	6	9	
5				
6				
7	1	4	7	
8	2	5	8	
9	3	6	9	
10				
11				
12				
13	one		two	
14	3		6	
15	2		5	
16	1		4	
17				

Available filters for DataFrames:

- **noheader**: Hide the column headers

Example:

```
{{ df | noheader }}
```

- **header**: Only return the header

Example:

```
{{ df | header }}
```

- **sortasc**: Sort in ascending order (indices are zero-based)

Example: sort by second, then by first column:

```
{{ df | sortasc(1, 0) }}
```

- **sortdesc**: Sort in descending order (indices are zero-based)

Example: sort by first, then by second column in descending order:

```
{{ df | sortdesc(0, 1) }}
```

- **columns:** Select/reorder columns and insert empty columns (indices are zero-based)

See also: `colslice`

Example: introduce an empty column (`None`) as the second column and switch the order of the second and third column:

```
{{ df | columns(0, None, 2, 1) }}
```

---

**Note:** Merged cells: you'll also have to introduce empty columns if you are using merged cells in your Excel template.

---

- **mul, div, sum, sub:** Apply an arithmetic operation (multiply, divide, sum, subtract) on a column (indices are zero-based)

Syntax:

```
{{ df | operation(value, col_ix[, fill_value]) }}
```

`fill_value` is optional and determines whether empty cells are included in the operation or not. To include empty values and thus make it behave like in Excel, set it to `0`.

Example: multiply the first column by 100:

```
{{ df | mul(100, 0) }}
```

Example: multiply the first column by 100 and the second column by 2:

```
{{ df | mul(100, 0) | mul(2, 1) }}
```

Example: add 100 to the first column including empty cells:

```
{{ df | add(100, 0, 0) }}
```

- **maxrows:** Maximum number of rows (currently, only `sum` is supported as aggregation function)

If your DataFrame has 12 rows and you use `maxrows(10, "Other")` as filter, you'll get a table that shows the first 9 rows as-is and sums up the remaining 3 rows under the label `Other`. If your data is unsorted, make sure to call `sortasc/sortdesc` first to make sure the correct rows are aggregated.

See also: `aggsmall`, `head`, `tail`, `rowslice`

Syntax:

```
{{ df | maxrows(number_rows, label[, label_col_ix]) }}
```

label\_col\_ix is optional: if left away, it will label the first column of the DataFrame (index is zero-based)

Examples:

```
{{ df | maxrows(10, "Other") }}
{{ df | sortasc(1) | maxrows(5, "Other") }}
{{ df | maxrows(10, "Other", 1) }}
```

- **aggsmall:** Aggregate rows with values below a certain threshold (currently, only sum is supported as aggregation function)

If the values in the specified row are below the threshold values, they will be summed up in a single row.

See also: maxrows, head, tail, rowslice

Syntax:

```
{{ df | aggsmall(threshold, threshold_col_ix, label[, label_col_ix][, min_
↪rows]) }}
```

label\_col\_ix and min\_rows are optional: if label\_col\_ix is left away, it will label the first column of the DataFrame (indices are zero-based). min\_rows has the effect that it skips rows from aggregating if it otherwise the number of rows falls below min\_rows. This prevents you from ending up with only one row called “Other” if you only have a few rows that are all below the threshold. NOTE that this parameter only makes sense if the data is sorted!

Examples:

```
{{ df | aggsmall(0.1, 2, "Other") }}
{{ df | sortasc(1) | aggsmall(0.1, 2, "Other") }}
{{ df | aggsmall(0.5, 1, "Other", 1) }}
{{ df | aggsmall(0.5, 1, "Other", 1, 10) }}
```

- **head:** Only show the top n rows

See also: maxrows, aggsmall, tail, rowslice

Example:

```
{{ df | head(3) }}
```

- **tail:** Only show the bottom n rows

See also: maxrows, aggsmall, head, rowslice

Example:

```
{{ df | tail(5) }}
```

- **rowslice:** Slice the rows

See also: maxrows, aggsmall, head, tail

Syntax:

```
{{ df | rowslice(start_index[, stop_index]) }}
```

`stop_index` is optional: if left away, it will stop at the end of the DataFrame

Example: Show rows 2 to 4 (indices are zero-based and interval is half-open, i.e. the start is including and the end is excluding):

```
{{ df | rowslice(2, 5) }}
```

Example: Show rows 2 to the end of the DataFrame:

```
{{ df | rowslice(2) }}
```

- **colslice**: Slice the columns

See also: `columns`

Syntax:

```
{{ df | colslice(start_index[, stop_index]) }}
```

`stop_index` is optional: if left away, it will stop at the end of the DataFrame

Example: Show columns 2 to 4 (indices are zero-based and interval is half-open, i.e. the start is including and the end is excluding):

```
{{ df | colslice(2, 5) }}
```

Example: Show columns 2 to the end of the DataFrame:

```
{{ df | colslice(2) }}
```

## 24.3 Excel Tables

Using Excel tables is the recommended way to format tables as the styling can be applied dynamically across columns and rows. You can also use themes and apply alternating colors to rows/columns. Go to **Insert > Table** and make sure that you activate **My table has headers** before clicking on OK. Add the placeholder as usual on the top-left of your Excel table (note that this example makes use of *Frames*):

Running the following script:

```
from xlwings.pro.reports import render_template
import pandas as pd

nrows, ncols = 3, 3
df = pd.DataFrame(data=nrows * [ncols * ['test']],
                  columns=[f'col {i}' for i in range(ncols)])
```

(continues on next page)

	A	B	C
1	Title	<b>&lt;frame&gt;</b>	
2			
3	<b>{{ df }}</b>		
4			
5			
6	Some static text.		
7			

(continued from previous page)

```
render_template('template.xlsx', 'output.xlsx', df=df)
```

Will produce the following report:

	A	B	C
1	Title		
2			
3	<b>col 0</b>	<b>col 1</b>	<b>col 2</b>
4	test	test	test
5	test	test	test
6	test	test	test
7			
8	Some static text.		
9			

Headers of Excel tables are relatively strict, e.g. you can't have multi-line headers or merged cells. To get around these limitations, uncheck the Header Row checkbox under Table Design and use the noheader filter (see DataFrame filters). This will allow you to design your own headers outside of the Excel Table.

---

**Note:**

- At the moment, you can only assign pandas DataFrames to tables
-

## 24.4 Excel Charts

To use Excel charts in your reports, follow this process:

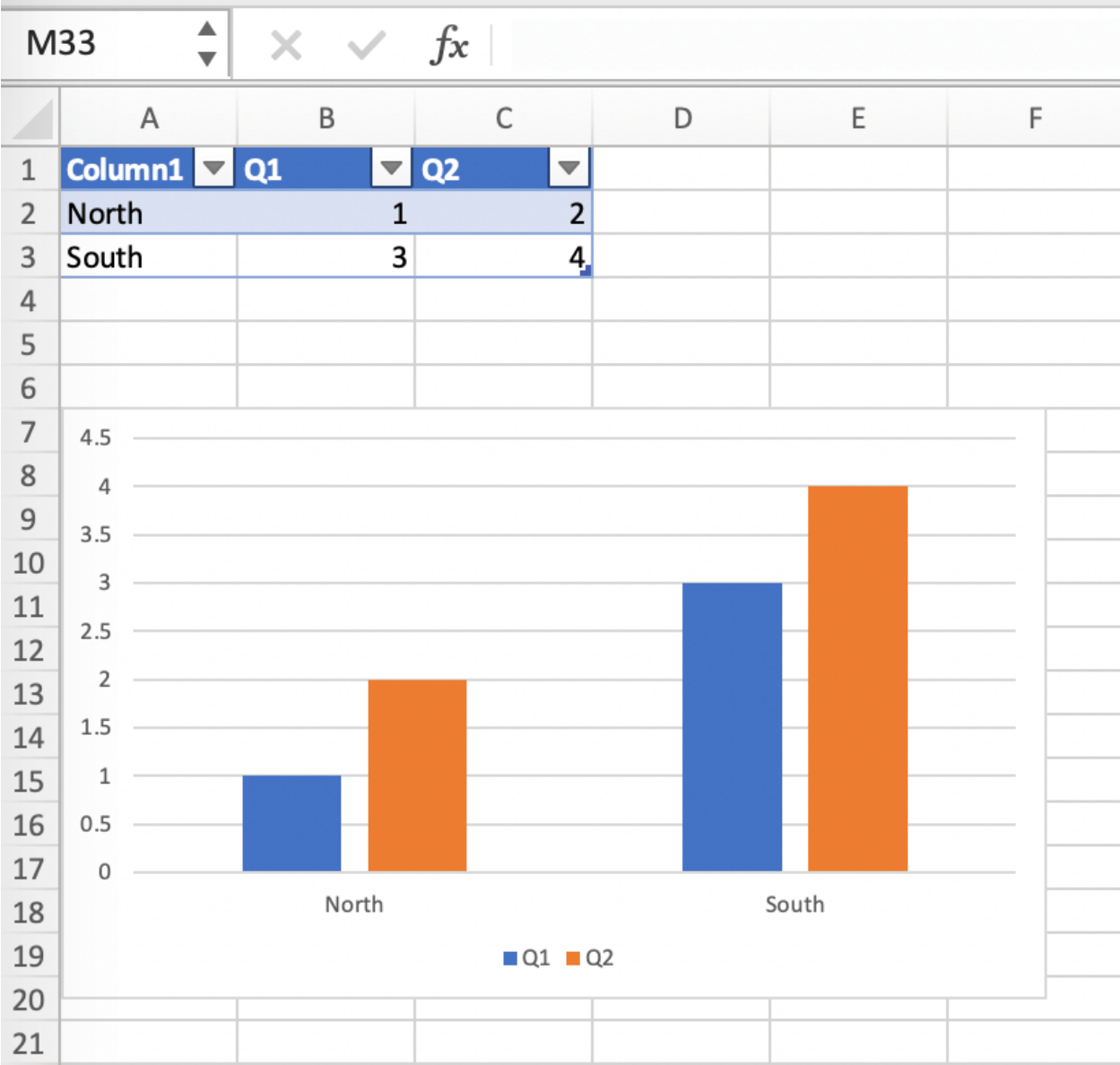
1. Add some sample/dummy data to your Excel template:

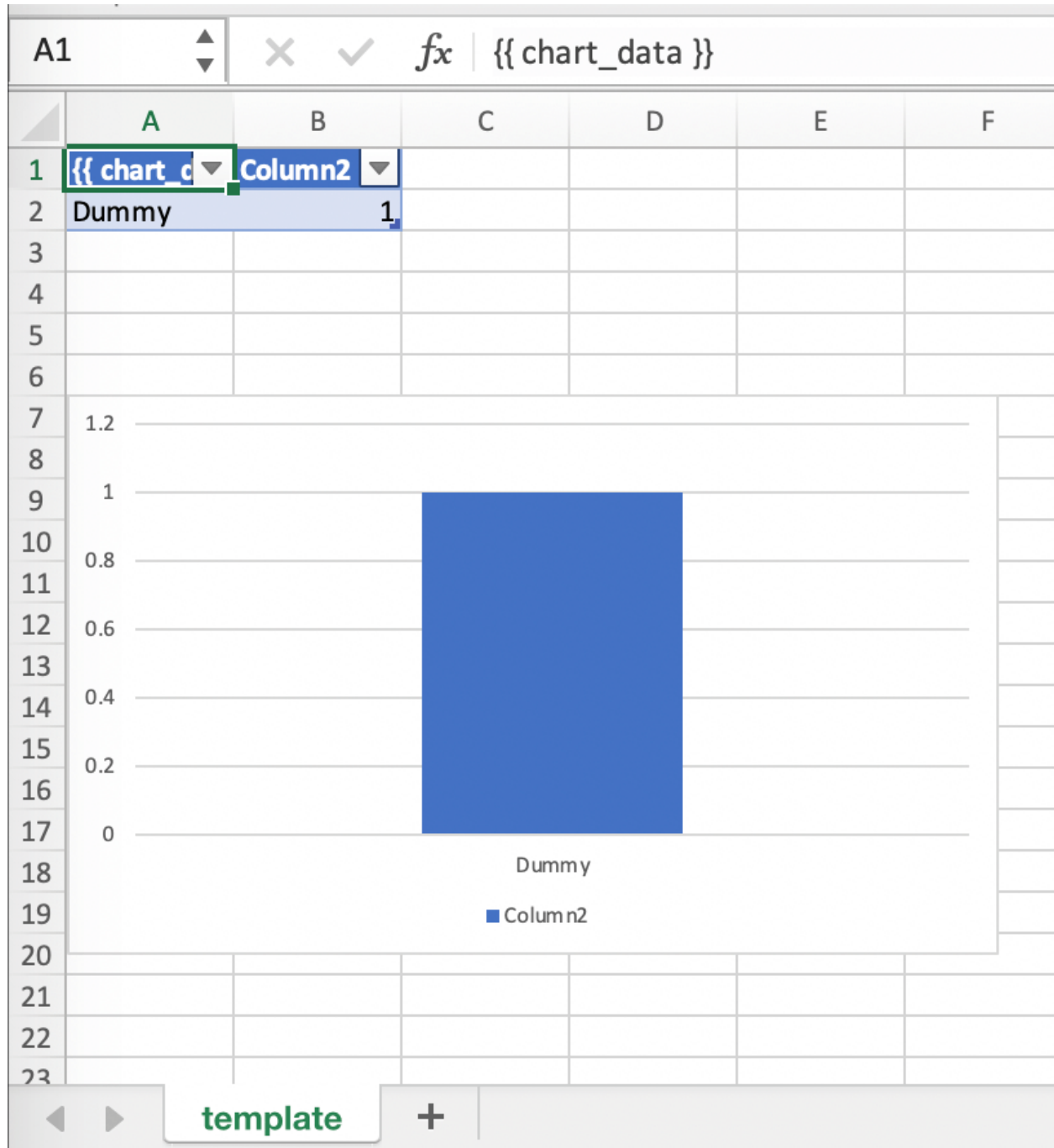
	A	B	C
1		Q1	Q2
2	North	1	2
3	South	3	4
4			
5			

2. If your data source is dynamic, turn it into an Excel Table (Insert > Table). Make sure you do this *before* adding the chart in the next step.

	A	B	C	D
1	Column1	Q1	Q2	
2	North	1	2	
3	South	3	4	
4				
5				

3. Add your chart and style it:
4. Reduce the Excel table to a 2 x 2 range and add the placeholder in the top-left corner (in our example `{{ chart_data }}`). You can leave in some dummy data or clear the values of the Excel table:





5. Assuming your file is called `mytemplate.xlsx` and your sheet `template` like on the previous screenshot, you can run the following code:

```
import xlwings as xw
import pandas as pd

df = pd.DataFrame(data={'Q1': [1000, 2000, 3000],
                        'Q2': [4000, 5000, 6000],
                        'Q3': [7000, 8000, 9000]},
                  index=['North', 'South', 'West'])

book = xw.Book("mytemplate.xlsx")
sheet = book.sheets['template'].copy(name='report')
sheet.render_template(chart_data=df.reset_index())
```

This will produce the following report, with the chart source correctly adjusted:

---

**Note:** If you don't want the source data on your report, you can place it on a separate sheet. It's easiest if you add and design the chart on the separate sheet, before cutting the chart and pasting it on your report template. To prevent the data sheet from being printed when calling `to_pdf`, you can give it a name that starts with `#` and it will be ignored. NOTE that if you start your sheet name with `##`, it won't be printed but also not rendered!

---

## 24.5 Images

Images are inserted so that the cell with the placeholder will become the top-left corner of the image. For example, write the following placeholder into your desired cell: `{{ logo }}`, then run the following code:

```
import xlwings as xw
from xlwings.pro.reports import Image

book = xw.Book('Book1.xlsx')
sheet = book.sheets['template'].copy(name='report')
sheet.render_template(logo=Image(r'C:\path\to\logo.png'))
```

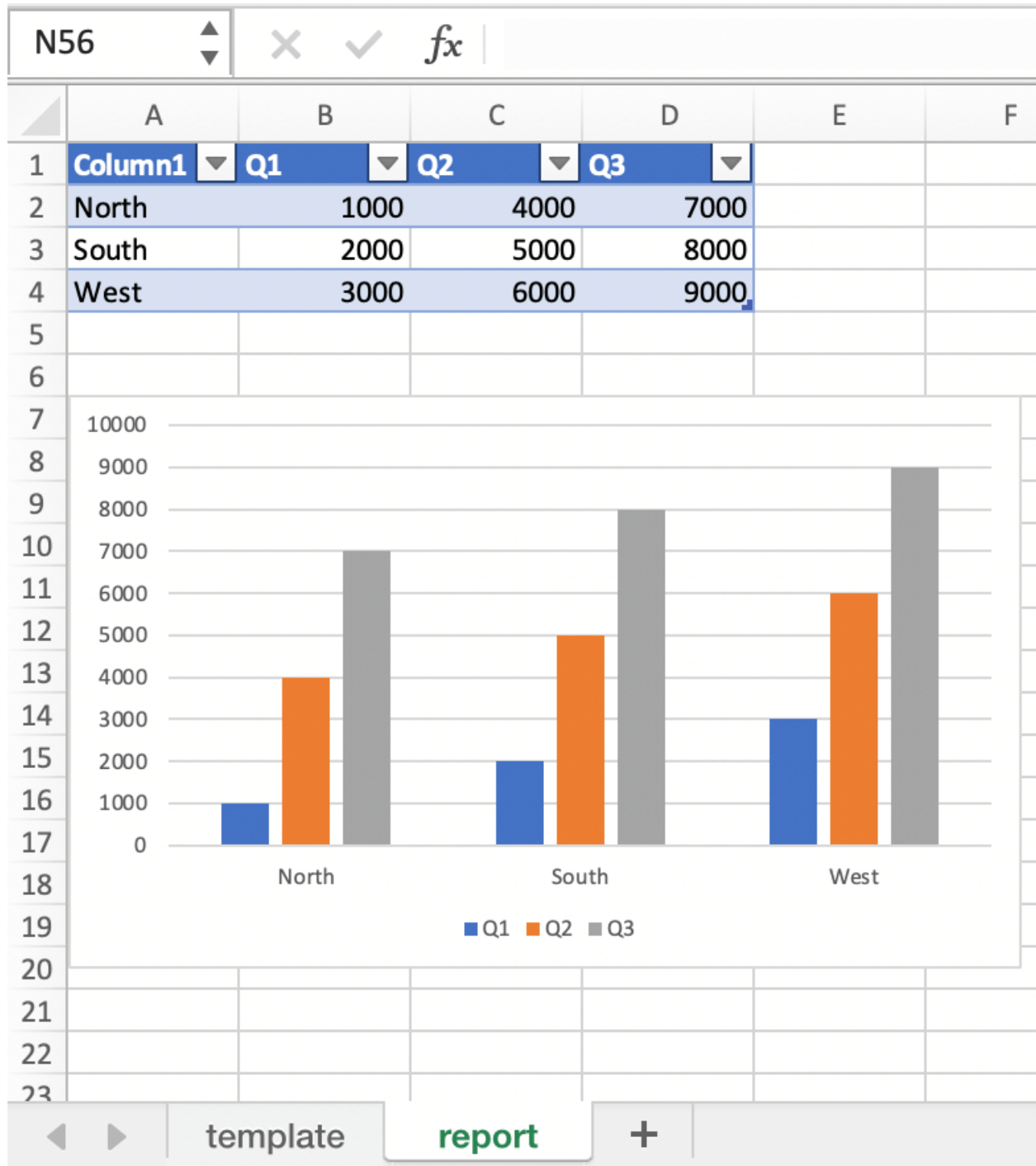
---

**Note:** `Image` also accepts a `pathlib.Path` object instead of a string.

---

If you want to use vector-based graphics, you can use `svg` on Windows and `pdf` on macOS. You can control the appearance of your image by applying filters on your placeholder.

Available filters for Images:



- **width:** Set the width in pixels (height will be scaled proportionally).

Example:

```
{{ logo | width(200) }}
```

- **height:** Set the height in pixels (width will be scaled proportionally).

Example:

```
{{ logo | height(200) }}
```

- **width and height:** Setting both width and height will distort the proportions of the image!

Example:

```
{{ logo | height(200) | width(200) }}
```

- **scale:** Scale your image using a factor (height and width will be scaled proportionally).

Example:

```
{{ logo | scale(1.2) }}
```

- **top:** Top margin. Has the effect of moving the image down (positive pixel number) or up (negative pixel number), relative to the top border of the cell. This is very handy to fine-tune the position of graphics object.

See also: `left`

Example:

```
{{ logo | top(5) }}
```

- **left:** Left margin. Has the effect of moving the image right (positive pixel number) or left (negative pixel number), relative to the left border of the cell. This is very handy to fine-tune the position of graphics object.

See also: `top`

Example:

```
{{ logo | left(5) }}
```

## 24.6 Matplotlib and Plotly Plots

For a general introduction on how to handle Matplotlib and Plotly, see also: *Matplotlib & Plotly Charts*. There, you'll also find the prerequisites to be able to export Plotly charts as pictures.

### 24.6.1 Matplotlib

Write the following placeholder in the cell where you want to paste the Matplotlib plot: `{{ lineplot }}`. Then run the following code to get your Matplotlib Figure object:

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

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

book = xw.Book('Book1.xlsx')
sheet = book.sheets['template'].copy(name='report')
sheet.render_template(lineplot=fig)
```

### 24.6.2 Plotly

Plotly works practically the same:

```
import plotly.express as px
import xlwings as xw

fig = px.line(x=["a","b","c"], y=[1,3,2], title="A line plot")
book = xw.Book('Book1.xlsx')
sheet = book.sheets['template'].copy(name='report')
sheet.render_template(lineplot=fig)
```

To change the appearance of the Matplotlib or Plotly plot, you can use the same filters as with images. Additionally, you can use the following filter:

- **format**: allows to change the default image format from `png` to e.g., `vector`, which will export the plot as vector graphics (svg on Windows and pdf on macOS). As an example, to make the chart smaller and use the vector format, you would write the following placeholder:

```
{{ lineplot | scale(0.8) | format("vector") }}
```

## 24.7 Text

You can work with placeholders in text that lives in cells or shapes like text boxes. If you have more than just a few words, text boxes usually make more sense as they won't impact the row height no matter how you style them. Using the same grid formatting across worksheets is key to getting a consistent multi-page report.

### 24.7.1 Simple Text without Formatting

New in version 0.21.4.

You can use any shapes like rectangles or circles, not just text boxes:

```
from xlwings.pro.reports import render_template

render_template('template.xlsx', 'output.xlsx', temperature=12.3)
```

This code turns this template:

	A	B	C	D	E	F	G
1							
2							
3							
4		The temperature today is {{ temperature }} degrees celsius.					
5							
6							
7							

into this report:

	A	B	C	D	E	F	G
1							
2							
3							
4		The temperature today is 12.3 degrees celsius.					
5							
6							
7							

While this works for simple text, you will lose the formatting if you have any. To prevent that, use a **Markdown** object, as explained in the next section.

If you will be printing on a *PDF Layout* with a dark background, you may need to change the font color to white. This has the nasty side effect that you won't see anything on the screen anymore. To solve that issue, use the `fontcolor` filter:

- **fontcolor:** Change the color of the whole (!) cell or shape. The primary purpose of this filter is to make white fonts visible in Excel. For most other colors, you can just change the color in Excel itself.

Note that this filter changes the font of the whole cell or shape and only has an effect if there is just a single placeholder—if you need to manipulate single words, use Markdown instead, see below. Black and white can be used as word, otherwise use a hex notation of your desired color.

Example:

```
{{ mytitle | fontcolor("white") }}  
{{ mytitle | fontcolor("#efefef") }}
```

### 24.7.2 Markdown Formatting

New in version 0.23.0.

You can format text in cells or shapes via Markdown syntax. Note that you can also use placeholders in the Markdown text that will take the values from the variables you supply via the `render_template` method:

```
import xlwings as xw  
from xlwings.pro import Markdown  
  
mytext = """\  
# Title  
  
Text bold and italic  
  
* A first bullet  
* A second bullet  
  
# {{ second_title }}  
  
This paragraph has a line break.  
Another line.  
"""  
  
# The first sheet requires a shape as shown on the screenshot  
sheet = xw.sheets.active  
sheet.render_template(myplaceholder=Markdown(mytext),  
                      second_title='Another Title')
```

This will render this template with the placeholder in a cell and a shape:

Like this (this uses the default formatting):

For more details about Markdown, especially about how to change the styling, see [Markdown Formatting](#).

	A	B	C	D	E	F
	1 2 3	{{ myplaceholder }}				

	A	B	C	D	E	F
	1 2	<b>Title</b>				
		Text <b>bold</b> and <i>italic</i>				
		<ul style="list-style-type: none"><li>• A first bullet</li><li>• A second bullet</li></ul>				
		<b>Another Title</b>				
		This paragraph has a line break. Another line.				

## 24.8 Date and Time

If a placeholder corresponds to a Python `datetime` object, by default, Excel will format that cell as a date-formatted cell. This isn't always desired as the formatting depends on the user's regional settings. To prevent that, format the cell in the `Text` format or use a `TextBox` and use the `datetime` filter to format the date in the desired format. The `datetime` filter accepts the `strftime` syntax—for a good reference, see e.g., [strftime.org](https://strftime.org).

To control the language of month and weekday names, you'll need to set the `locale` in your Python code. For example, for German, you would use the following:

```
import locale
locale.setlocale(locale.LC_ALL, 'de_DE')
```

Example: The default formatting is December 1, 2020:

```
{{ mydate | datetime }}
```

Example: To apply a specific formatting, provide the desired format as filter argument. For example, to get it in the 12/31/20 format:

```
{{ mydate | datetime("%m/%d/%y") }}
```

## 24.9 Number Format

The `format` filter allows you to format numbers by using the same mechanism as offered by Python's f-strings. For example, to format the placeholder `performance=0.13` as `13.0%`, you would do the following:

```
{{ performance | format(".1%") }}
```

This corresponds to the following f-string in Python: `f"{performance:0.1%}"`. To get an introduction to the formatting string syntax, have a look at the [Python String Format Cookbook](#).

## 24.10 Frames: Multi-column Layout

Frames are vertical containers in which content is being aligned according to their height. That is, within Frames:

- Variables do not overwrite existing cell values as they do without Frames.
- Formatting is applied dynamically, depending on the number of rows your object uses in Excel

To use Frames, insert a Note with the text `<frame>` into **row 1** of your Excel template wherever you want a new dynamic column to start. Frames go from one `<frame>` to the next `<frame>` or the right border of the used range.

How Frames behave is best demonstrated with an example: The following screenshot defines two frames. The first one goes from column A to column E and the second one goes from column F to column I, since this is the last column that is used.

	A	B	C	D	E	F	G	H	I
1	Table 1	<frame>				Table 3	<frame>		
2	{{ df1 }}					{{ df2 }}			
3									
4									
5	Table 2					Table 4			
6	{{ df2 }}					{{ df1 }}			
7									

You can define and format DataFrames by formatting

- one header and
- one data row

If you use the `noheader` filter for DataFrames, you can leave the header away and format a single data row. Alternatively, you could also use Excel Tables, as they can make formatting easier.

Running the following code:

```
from xlwings.pro.reports import render_template
import pandas as pd

df1 = pd.DataFrame([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
df2 = pd.DataFrame([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15]])

data = dict(df1=df1.reset_index(), df2=df2.reset_index())

render_template('my_template.xlsx',
                'my_report.xlsx',
                **data)
```

will generate this report:

	A	B	C	D	E	F	G	H	I
1	Table 1					Table 3			
2		0	1	2			0	1	2
3		0	1	2			0	1	2
4		1	4	5			1	4	5
5		2	7	8			2	7	8
6							3	10	11
7	Table 2						4	13	14
8		0	1	2					
9		0	1	2		Table 4			
10		1	4	5			0	1	2
11		2	7	8			0	1	2
12		3	10	11			1	4	5
13		4	13	14			2	7	8

## 24.11 PDF Layout

Using the `layout` parameter in the `to_pdf()` command, you can “print” your Excel workbook on professionally designed PDFs for pixel-perfect reports in your corporate layout including headers, footers, backgrounds and borderless graphics:

```
from xlwings.pro.reports import render_template
import pandas as pd

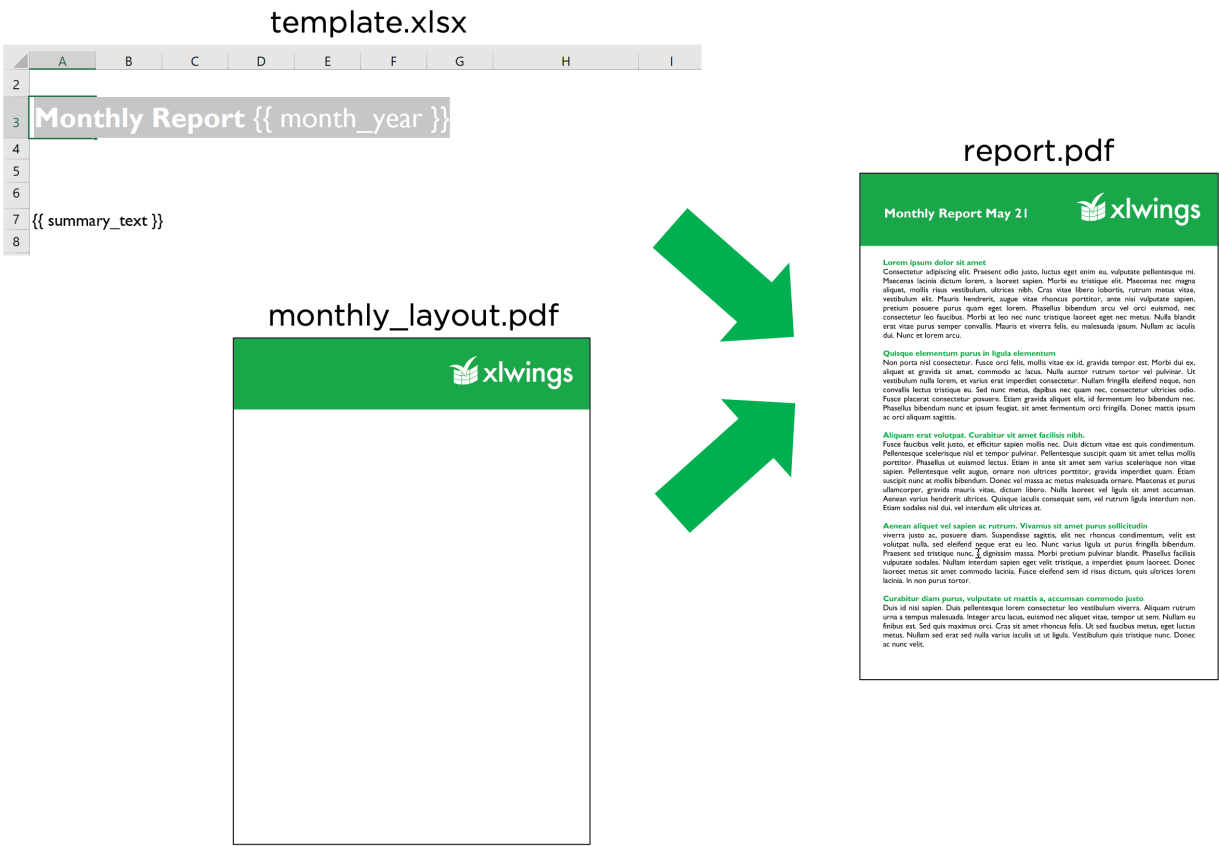
df = pd.DataFrame([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

book = render_template('template.xlsx',
                      'report.xlsx',
                      month_year = 'May 21',
                      summary_text = '...')

book.to_pdf('report.pdf', layout='monthly_layout.pdf')
```

Note that the layout PDF either needs to consist of a single page (will be used for each reporting page) or will need to have the same number of pages as the report (each report page will be printed on the corresponding layout page).

To create your layout PDF, you can use any program capable of exporting a file in PDF format such as PowerPoint or Word, but for the best results consider using a professional desktop publishing software such as Adobe InDesign.





## MARKDOWN FORMATTING

This feature requires xlwings *PRO*.

New in version 0.23.0.

Markdown offers an easy and intuitive way of styling text components in your cells and shapes. For an introduction to Markdown, see e.g., [Mastering Markdown](#).

Markdown support is in an early stage and currently only supports:

- First-level headings
- Bold (i.e., strong)
- Italic (i.e., emphasis)
- Unordered lists

It doesn't support nested objects yet such as 2nd-level headings, bold/italic within bullet points or nested bullet points.

Let's go through an example to see how everything works!

```
from xlwings.pro import Markdown, MarkdownStyle

mytext = """\
# Title

Text bold and italic

* A first bullet
* A second bullet

# Another Title

This paragraph has a line break.
Another line.
"""

sheet = xw.Book("Book1.xlsx").sheets[0]
```

(continues on next page)

(continued from previous page)

```
# Range
sheet['A1'].clear()
sheet['A1'].value = Markdown(mytext)

# Shape: The following expects a shape like a Rectangle on the sheet
sheet.shapes[0].text = ""
sheet.shapes[0].text = Markdown(mytext)
```

Running this code will give you this nicely formatted text:

	A	B	C	D	E	F
	<b>Title</b>  Text <b>bold</b> and <i>italic</i>  <ul style="list-style-type: none"> <li>• A first bullet</li> <li>• A second bullet</li> </ul> <b>Another Title</b>  This paragraph has a line break. 1 Another line. 2	<div> <b>Title</b>   Text <b>bold</b> and <i>italic</i>   <ul style="list-style-type: none"> <li>• A first bullet</li> <li>• A second bullet</li> </ul>   <b>Another Title</b>   This paragraph has a line break.  Another line. </div>				

But why not make things a tad more stylish? By providing a `MarkdownStyle` object, you can define your style. Let's change the previous example like this:

```
from xlwings.pro import Markdown, MarkdownStyle

mytext = ""\
# Title

Text bold and italic

* A first bullet
* A second bullet

# Another Title

This paragraph has a line break.
Another line.
""

sheet = xw.Book("Book1.xlsx").sheets[0]

# Styling
```

(continues on next page)

(continued from previous page)

```

style = MarkdownStyle()
style.h1.font.color = (255, 0, 0)
style.h1.font.size = 14
style.h1.font.name = 'Comic Sans MS' # No, that's not a font recommendation...
style.h1.blank_lines_after = 0
style.unordered_list.bullet_character = '\N{heavy black heart}' # Emojis are fun!
↪ fun!

# Range
sheet['A1'].clear()
sheet['A1'].value = Markdown(mytext, style) # <= provide your style object here

# Shape: The following expects a shape like a Rectangle on the sheet
sheet.shapes[0].text = ""
sheet.shapes[0].text = Markdown(mytext, style)

```

Here is the output of this:

	A	B	C	D	E	F
	<b>Title</b> Text <b>bold</b> and <i>italic</i>  ♥ A first bullet ♥ A second bullet  <b>Another Title</b> This paragraph has a line break. 1 Another line. 2	<b>Title</b> Text <b>bold</b> and <i>italic</i>  ♥ A first bullet ♥ A second bullet  <b>Another Title</b> This paragraph has a line break. Another line.				

You can override all properties, i.e., you can change the emphasis from italic to a red font or anything else you want:

```

>>> style.strong.bold = False
>>> style.strong.color = (255, 0, 0)
>>> style.strong
strong.color: (255, 0, 0)

```

Markdown objects can also be used with template-based reporting, see [xlwings Reports](#).

**Note:** macOS currently doesn't support the formatting (bold, italic, color etc.) of Markdown text due to a bug with AppleScript/Excel. The text will be rendered correctly though, including bullet points.

See also the API reference:

- *Markdown class*
- *MarkdownStyle class*

## RELEASING XLWINGS TOOLS

This feature requires xlwings *PRO*.

xlwings *PRO* offers a simple way to deploy your xlwings tools to your end users without the usual hassle that's involved when installing and configuring Python and xlwings. End users don't need to know anything about Python as they only need to:

- Run an installer (one installer can power many different Excel workbooks)
- Use the Excel workbook as if it was a normal macro-enabled workbook

Advantages:

- **Zero-config:** The end user doesn't have to configure anything throughout the whole process.
- **No add-in required:** No installation of the xlwings add-in required.
- **Easy to update:** If you want to deploy an update of your Python code, it's often good enough to distribute a new version of your workbook.
- **No conflicts:** The installer doesn't touch any environment variables or registry keys and will therefore not conflict with any existing Python installations.
- **Deploy key:** The release command will add a deploy key as your `LICENSE_KEY`. A deploy key won't expire and end users won't need a paid subscription.

You as a developer need to create the one-click installer and run the `xlwings release` command on the workbook. Let's go through these two steps in detail!

There is a video walkthrough at: [https://www.youtube.com/watch?v=yw36VT\\_n1qg](https://www.youtube.com/watch?v=yw36VT_n1qg)

### 26.1 Step 1: One-Click Installer

As a subscriber of one of our [paid plans](#), you will get access to a private GitHub repository, where you can build your one-click installer:

- 1) Update your `requirements.txt` file with your dependencies: in your repository, start by clicking on the `requirements.txt` file. This will open the following screen where you can click on the pencil icon to edit the file (if you know your way around Git, you can also clone the repository and use your local commit/push workflow instead):

```
2 lines (1 sloc) | 22 Bytes
1 xlwings[pro]==0.20.8
2
```

After you're done with your edits, click on the green **Commit changes** button.

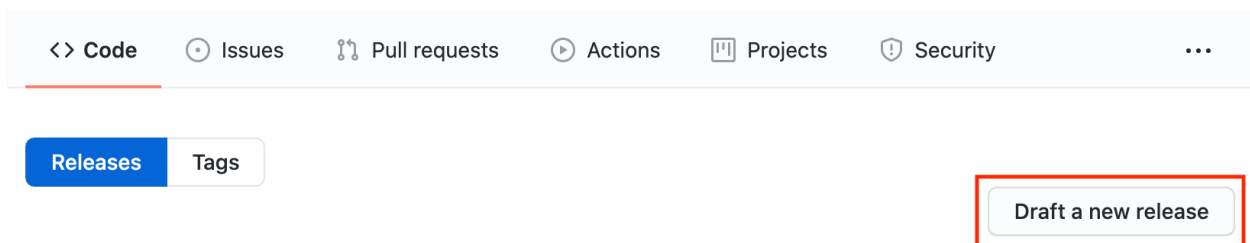
**Note:** If you are unsure about your dependencies, it's best to work locally with a virtual or Conda environment. In the virtual/Conda environment, only install packages that you need, then run: `pip list --format=freeze`.

2) On the right-hand side of the landing page, click on **Releases**:



No releases published  
[Create a new release](#)

On the next screen, click on **Draft a new release** (note, the very first time, you will see a green button called **Create a new release** instead):



This will bring up the following screen, where you'll only have to fill in a **Tag version** (e.g., `1.0.0`), then click on the green button **Publish release**:

After 3-5 minutes (you can follow the progress under the **Actions** tab), you'll find the installer ready for download under **Releases** (ignore the `zip` and `tar.gz` files):

**Note:** The one-click installer is a normal Python installation that you can use with multiple Excel workbooks. Hence, you don't need to create a separate installer for each workbook as long as they all work with the same set of dependencies as defined by the `requirements.txt` file.

Releases

Tags

1.0.0

@ 

🔗 Target: main ▼


Choose an existing tag, or create a new tag on publish

Release title

Write

Preview

Describe this release

Attach files by dragging & dropping, selecting or pasting them. 

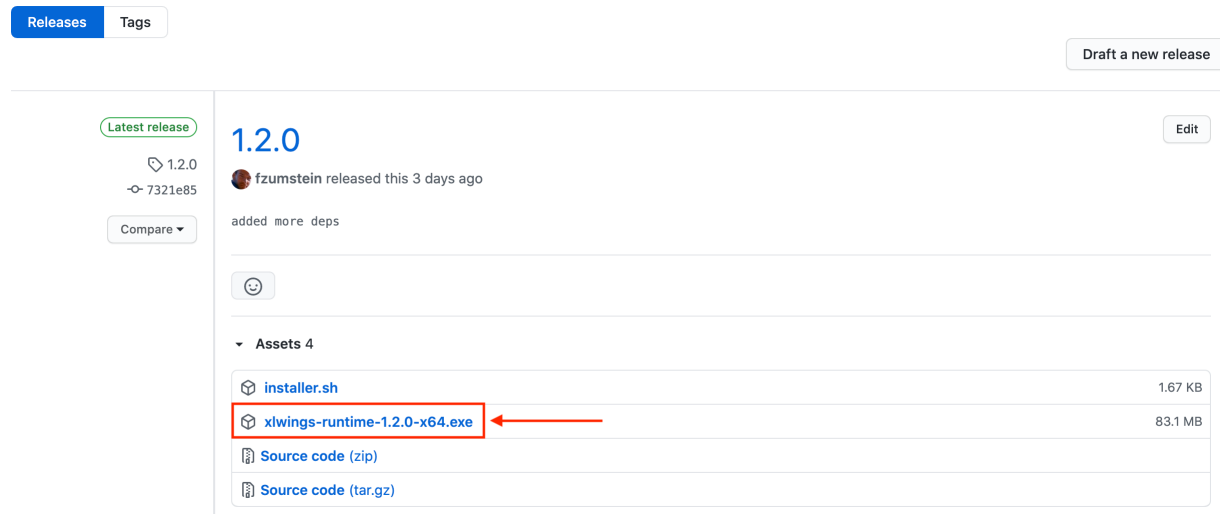
↓

 Attach binaries by dropping them here or selecting them.

☐ **This is a pre-release**  
We'll point out that this release is identified as non-production ready.

Publish release

Save draft



Releases Tags

Draft a new release

Latest release

1.2.0  
7321e85

Compare

1.2.0  
fzumstein released this 3 days ago

added more deps

Assets 4

installer.sh	1.67 KB
xlwings-runtime-1.2.0-x64.exe	83.1 MB
Source code (zip)	
Source code (tar.gz)	

## 26.2 Step 2: Release Command (CLI)

The release command is part of the xlwings CLI (command-line client) and will prepare your Excel file to work with the one-click installer generated in the previous step. Before anything else:

- Make sure that you have enabled Trust access to the VBA project object model under File > Options > Trust Center > Trust Center Settings > Macro Settings. You only need to do this once and since this is a developer setting, your end users won't need to bother about this. This setting is needed so that xlwings can update the Excel file with the correct version of the VBA code.
- Run the installer from the previous step. This will not interfere with your existing Python installation as it won't touch your environment variables or registry. Instead, it will only write to the following folder: %LOCALAPPDATA%\<installer-name>.
- Make sure that your local version of xlwings corresponds to the version of xlwings in the requirements.txt from the installer. The easiest way to double-check this is to run `pip freeze` on a Command Prompt or Anaconda Prompt. If your local version of xlwings differs, install the same version as the installer uses via: `pip install xlwings==<version from installer>`.

To work with the release command, you should have your workbook in the xlsx format and all the Python modules in the same folder:

```
myworkbook.xlsx
mymodule_one.py
mymodule_two.py
...
```

You currently can't organize your code in directories, but you can easily import `mymodule_two` from `mymodule_one`.

Make sure that your Excel workbook is the active workbook, then run the following command on a Command/Anaconda Prompt:

**xlwings release**

If this is the first time you are running this command, you will be asked a few questions. If you are shown a [Y/n], you can hit Enter to accept the default as expressed by the capitalized letter:

- **Name of your one-click installer?** *Type in the name of your one-click installer. If you want to use a different Python distribution (e.g., Anaconda), you can leave this empty (but you will need to update the xlwings.conf sheet with the Conda settings once the release command has been run).*
- **Embed your Python code?** [Y/n] *This will copy the Python code into the sheets of the Excel file. It will respect all Python files that are in the same folder as the Excel workbook.*
- **Hide the config sheet?** [Y/n] *This will hide the xlwings.conf sheet.*
- **Hide the sheets with the embedded Python code?** [Y/n] *If you embed your Python code, this will hide all sheets with a .py ending.*
- **Allow your tool to run without the xlwings add-in?** [Y/n] *This will remove the VBA reference to xlwings and copy in the xlwings VBA modules so that the end users don't need to have the xlwings add-in installed. Note that in this case, you will need to have your RunPython calls bound to a button as you can't use the Ribbon's Run main button anymore.*

Whatever answers you pick, you can always change them later by editing the xlwings.conf sheet or by deleting the xlwings.conf sheet and re-running the xlwings release command. If you go with the defaults, you only need to provide your end users with the one-click installer and the Excel workbook, no external Python files are required.

## 26.3 Updating a Release

To edit your Python code, it's easiest to work with external Python files and not with embedded code. To stop xlwings from using the embedded code, simply delete all sheets with a .py ending and the workbook will again use the external Python modules. Once you are done editing the files, simply run the xlwings release command again, which will embed the updated code. If you haven't done any changes to your dependencies (i.e., you haven't upgraded a package or introduced a new one), you only need to redeploy your Excel workbook to have the end users get the update.

If you did make changes to the requirements.txt and release a new one-click installer, you will need to have the users install the new version of the installer first.

---

**Note:** Every time you change the xlwings version in requirements.txt of your one-click installer, make sure to upgrade your local xlwings installatino to the same version and run xlwings release again!

---

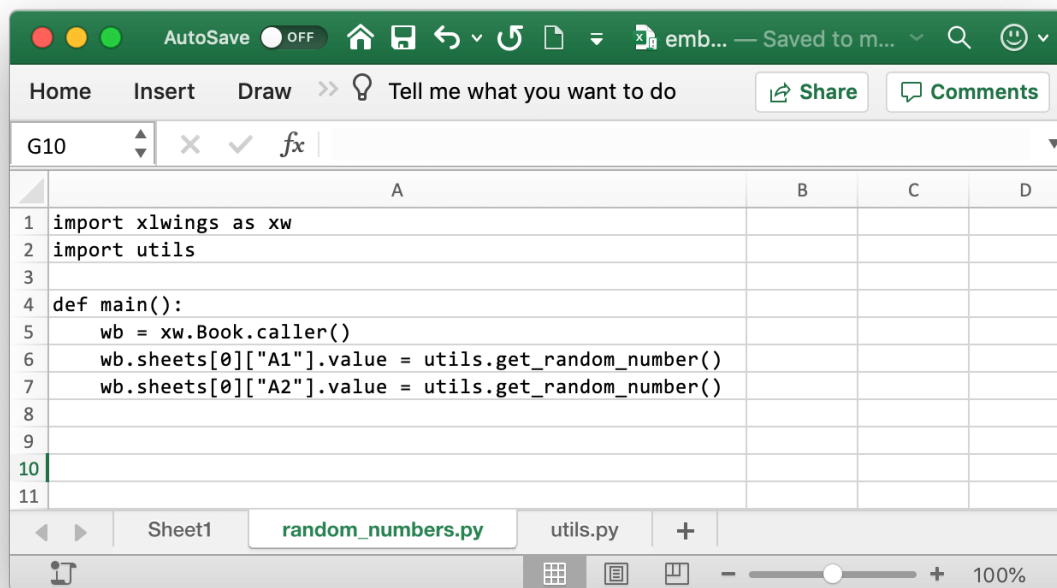
## 26.4 Embedded Code Explained

When you run the `xlwings release` command, your code will be embedded automatically (except if you switch this behavior off). You can, however, also embed code directly: on a command line, run the following command:

```
xlwings code embed
```

This will import all Python files from the current directory and paste them into Excel sheets of the currently active workbook. Now, you can use `RunPython` as usual: `RunPython "import mymodule;mymodule.myfunction()"`.

Note that you can have multiple Excel sheets and import them like normal Python files. Consider this example:



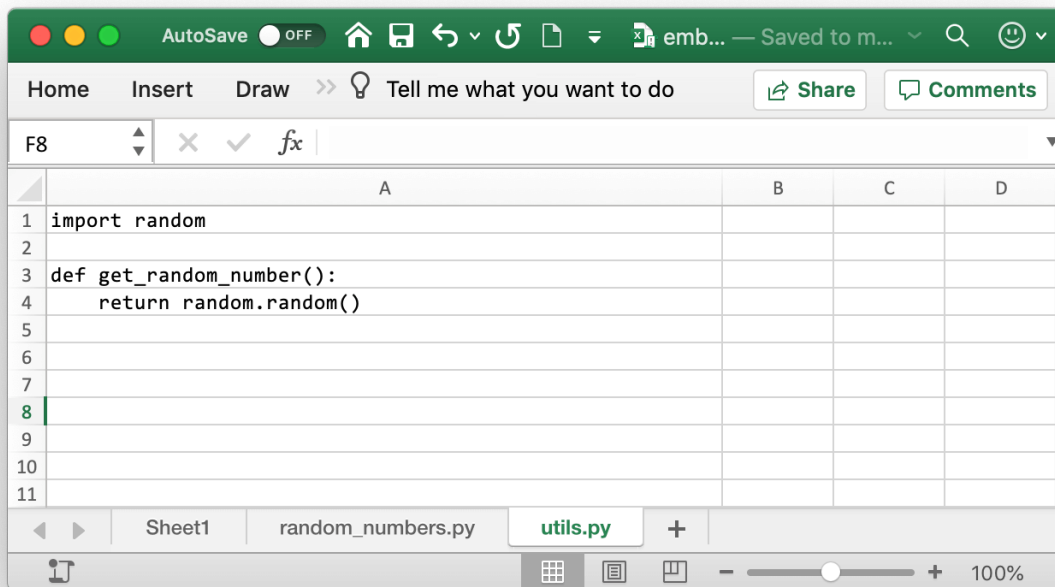
You can call the `main` function from VBA like so:

```
Sub RandomNumbers()
    RunPython "import random_numbers;random_numbers.main()"
End Sub
```

---

### Note:

- UDFs modules don't have to be added to the UDF Modules explicitly when using embedded code. However, in contrast to how it works with external files, you currently need to re-import the functions when you change them.



- While you can hide your sheets with your code, they will be written to a temporary directory in clear text.



## PERMISSIONING OF CODE EXECUTION

This feature requires *xlwings PRO*.

*xlwings* allows you to control which Python modules are allowed to run from Excel. In order to use this functionality, you need to run your own web server. You can choose between an HTTP POST and a GET request for the permissioning:

- **GET:** This is the simpler option as you only need to host a static JSON file that you can generate via the *xlwings* CLI. You can use any web server that is capable of serving static files (e.g., *nginx*) or use a free external service like GitHub pages. However, every permission change requires you to update the JSON file on the server.
- **POST:** This option relies on the web server to validate the incoming payload of the POST request. While this requires custom logic on your end, you are able to connect it with any internal system (such as a database or LDAP server) to dynamically decide whether a user should be able to run a specific Python module through *xlwings*.

Before looking at both of these options in more detail, let's go through the prerequisites and configuration.

---

**Note:** This feature does not stop users from running arbitrary Python code through Python directly. Rather, think of it as a mechanism to prevent accidental execution of Python code from Excel via *xlwings*.

---

### 27.1 Prerequisites

- This functionality requires every end user to have the `requests` and `cryptography` libraries installed. You can install them via `pip`:

```
pip install requests cryptography
```

or via Conda:

```
conda install requests cryptography
```

- You need to have a `LICENSE_KEY` in the form of a [trial key](#), a paid license key or a deploy key.

## 27.2 Configuration

While xlwings offers various ways to configure your workbook (see [Configuration](#)), it will only respect the permissioning settings in the config file in the user's home folder (on Windows, this is %USERPROFILE%\xlwings\xlwings.conf):

- To prevent end users from overwriting xlwings.conf, you'll need to make sure that the file is owned by the Administrator while giving end users read-only permissions.
- Add the following settings while replacing the PERMISSION\_CHECK\_URL and PERMISSION\_CHECK\_METHOD (POST or GET) with the appropriate value for your case:

```
"LICENSE_KEY", "YOUR_LICENSE_OR_DEPLOY_KEY"
"PERMISSION_CHECK_ENABLED", "True"
"PERMISSION_CHECK_URL", "https://myurl.com"
"PERMISSION_CHECK_METHOD", "POST"
```

## 27.3 GET request

You can generate the static JSON file by using the xlwings CLI:

- Print the JSON string for all Python modules in a certain folder:

```
cd myfolder
xlwings permission cwd
```

- Print the JSON string for all embedded modules of the active workbook:

```
xlwings permission book
```

Both commands will print a JSON string similar to this one:

```
{
  "modules": [
    {
      "file_name": "myfile.py",
      "sha256": "cea259922207049a734c88930b5c09109deb6b55f692fd0832f4e57052d85896
→",
      "machine_names": [
        "DESKTOP-QQ27RP3"
      ]
    },
    {
      "file_name": "myfile2.py",
      "sha256": "355200bb9ae00fcec1d7b660e7dd95fb3dbf246a9db397a6daa2471458a8e6cb
→",
      "machine_names": [
```

(continues on next page)

(continued from previous page)

```

        "DESKTOP-QQ27RP3"
    ]
}
]
}

```

All you need to do at this point is:

- Add additional machines names e.g., "machine\_names": [""DESKTOP-QQ27RP3", "DESKTOP-XY12AS2"]. Alternatively, you can use the "\*" wildcard if you want to allow the module to be used on all end user's computers. In case of the wildcard, it will still make sure that the file's content hasn't been changed by looking at its sha256 hash. xlwings uses `import socket;socket.gethostname()` as the machine name.
- Make this JSON file accessible via your web server and update the settings in the `xlwings.conf` file accordingly (see above).

## 27.4 POST request

If you work with POST requests, xlwings will post a payload similar to the following:

```

{
  "machine_name": "DESKTOP-QQ27RP3",
  "modules": [
    {
      "file_name": "myfile.py",
      "sha256":
↪ "cea259922207049a734c88930b5c09109deb6b55f692fd0832f4e57052d85896"
    },
    {
      "file_name": "myfile2.py",
      "sha256":
↪ "355200bb9ae00fcec1d7b660e7dd95fb3dbf246a9db397a6daa2471458a8e6cb"
    }
  ]
}

```

It is now up to you to validate this request and:

- Return the HTTP status code 200 ("success") if the user is allowed to run the code of these modules
- Return the HTTP status code 403 ("forbidden") if the user is not allowed to run the code of these modules

Note that xlwings only checks for HTTP status code 200, so any other status code will fail.

## 27.5 Implementation Details & Limitations

- Currently, RunPython and user-defined functions (UDFs) are supported. RunFrozenPython is not supported.
- Permissions checks are only done when the Python module is run via Excel/xlwings, it has no effect on Python code that is run from Python directly.
- RunPython won't allow you to run code that uses the `from x import y` syntax. Use `import x;x.y` instead.
- The answer of the permissioning server is cached for the duration of the Python session. For UDFs, this means until the functions are re-imported or the **Restart UDF Server** button is clicked or until Excel is restarted. The same is true if you run RunPython with the **Use UDF Server** option. By default, however, RunPython starts a new Python session every time, so it will contact the server whenever you call RunPython.
- Only top-level modules are checked, i.e. modules that are imported as UDFs or run via RunPython call. Any modules that are imported as dependencies of these modules are not checked.
- RunPython with external Python source files depends on logic in the VBA part of xlwings. UDFs and RunPython calls that use embedded code will only rely on Python to perform the permissioning check.

## 28.1 Top-level functions

`xlwings.load(index=1, header=1, chunksize=5000)`

Loads the selected cell(s) of the active workbook into a pandas DataFrame. If you select a single cell that has adjacent cells, the range is auto-expanded (via current region) and turned into a pandas DataFrame. If you don't have pandas installed, it returns the values as nested lists.

---

**Note:** Only use this in an interactive context like e.g. a Jupyter notebook! Don't use this in a script as it depends on the active book.

---

### Parameters

- **index** (*bool or int, default 1*) – Defines the number of columns on the left that will be turned into the DataFrame's index
- **header** (*bool or int, default 1*) – Defines the number of rows at the top that will be turned into the DataFrame's columns
- **chunksize** (*int, default 5000*) – Chunks the loading of big arrays.

### Examples

```
>>> import xlwings as xw
>>> xw.load()
```

See also: [view](#)

Changed in version 0.23.1.

`xlwings.view(obj, sheet=None, table=True, chunksize=5000)`

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.

**Note:** Only use this in an interactive context like e.g. a Jupyter notebook! Don't use this in a script as it depends on the active book.

---

### 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.
- **table** (*bool, default True*) – If your object is a pandas DataFrame, by default it is formatted as an Excel Table
- **chunksize** (*int, default 5000*) – Chunks the loading of big arrays.

### 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)
```

See also: [load](#)

Changed in version 0.22.0.

## 28.2 Object model

### 28.2.1 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>])
```

#### **property** 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.

**property count**

Returns the number of apps.

New in version 0.9.0.

**keys()**

Provides the PIDs of the Excel instances that act as keys in the Apps collection.

New in version 0.13.0.

## 28.2.2 App

**class** `xlwings.App(visible=None, spec=None, add_book=True, impl=None)`

An app corresponds to an Excel instance and should normally be used as context manager to make sure that everything is properly cleaned up again and to prevent zombie processes. New Excel instances can be fired up like so:

```
import xlwings as xw

with xw.App() as app:
    print(app.books)
```

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

```
>>> xw.apps
Apps([<Excel App 1668>, <Excel App 1644>])
>>> xw.apps[1668] # get the available PIDs via xw.apps.keys()
<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 frontmost application and hand over focus from Python to Excel.

New in version 0.9.0.

**property api**

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

New in version 0.9.0.

**property 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.

**property 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.

**property cut\_copy\_mode**

Gets or sets the status of the cut or copy mode. Accepts False for setting and returns None, copy or cut when getting the status.

New in version 0.24.0.

**property 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.

**property enable\_events**

True if events are enabled. Read/write boolean.

New in version 0.24.4.

**property hwnd**

Returns the Window handle (Windows-only).

New in version 0.9.0.

**property interactive**

True if Excel is in interactive mode. If you set this property to False, Excel blocks all input from the keyboard and mouse (except input to dialog boxes that are displayed by your code). Read/write Boolean. Note: Not supported on macOS.

New in version 0.24.4.

**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
```

Types are supported too:

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

However typed arrays are not supported. So the following won't work

**Function MySum(arr() as integer)** *'''* code here

End Function

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

New in version 0.9.0.

### **property pid**

Returns the PID of the app.

New in version 0.9.0.

### **properties(\*\*kwargs)**

Context manager that allows you to easily change the app's properties temporarily. Once the code leaves the with block, the properties are changed back to their previous state. Note: Must be used as context manager or else will have no effect. Also, you can only use app properties that you can both read and write.

## **Examples**

```
import xlwings as xw
app = App()

# Sets app.display_alerts = False
with app.properties(display_alerts=False):
    # do stuff

# Sets app.calculation = 'manual' and app.enable_events = True
with app.properties(calculation='manual', enable_events=True):
    # do stuff

# Makes sure the status bar is reset even if an error happens in the
↪with block
with app.properties(status_bar='Calculating...'):
    # do stuff
```

New in version 0.24.4.

### **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.

**render\_template**(*template=None, output=None, book\_settings=None, \*\*data*)

This function requires xlwings *PRO*.

This is a convenience wrapper around [\*mysheet.render\\_template\*](#)

Writes the values of all key word arguments to the output file according to the `template` and the variables contained in there (Jinja variable syntax). Following variable types are supported: strings, numbers, lists, simple dicts, NumPy arrays, Pandas DataFrames, pictures and Matplotlib/Plotly figures.

#### Parameters

- **template** (*str or path-like object*) – Path to your Excel template, e.g. `r'C:\Path\to\my_template.xlsx'`
- **output** (*str or path-like object*) – Path to your Report, e.g. `r'C:\Path\to\my_report.xlsx'`
- **book\_settings** (*dict, default None*) – A dictionary of xlwings. Book parameters, for details see: [\*xlwings.Book\*](#). For example: `book_settings={'update_links': False}`.
- **data** (*kwargs*) – All key/value pairs that are used in the template.

#### Returns

- **wb** (*xlwings Book*)
- .. *versionadded:: 0.24.4*

#### property 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.

#### property selection

Returns the selected cells as `Range`.

New in version 0.9.0.

#### property startup\_path

Returns the path to XLSTART which is where the xlwings add-in gets copied to by doing `xlwings addin install`.

New in version 0.19.4.

#### property status\_bar

Gets or sets the value of the status bar. Returns `False` if Excel has control of it.

New in version 0.20.0.

#### property version

Returns the Excel version number object.

## Examples

```
>>> import xlwings as xw
>>> xw.App().version
VersionNumber('15.24')
>>> xw.apps[10559].version.major
15
```

Changed in version 0.9.0.

### property visible

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

New in version 0.3.3.

## 28.2.3 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[10559].books # specific app, get the PIDs via xw.apps.keys()
Books([<Book [Book1]>, <Book [Book2]>])
```

New in version 0.9.0.

### property active

Returns the active Book.

### add()

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

**open**(*fullname*, *update\_links=None*, *read\_only=None*, *format=None*, *password=None*,  
*write\_res\_password=None*, *ignore\_read\_only\_recommended=None*, *origin=None*,  
*delimiter=None*, *editable=None*, *notify=None*, *converter=None*, *add\_to\_mru=None*,  
*local=None*, *corrupt\_load=None*)

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* or *path-like object*) – filename or fully qualified filename, e.g. `r'C:\path\to\file.xlsx'` or `'file.xlsm'`. Without a full path, it looks for the file in the current working directory.
- **Parameters (Other)** – see: `xlwings.Book()`

### Returns Book

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

## 28.2.4 Book

```
class xlwings.Book(fullname=None, update_links=None, read_only=None, format=None,
                    password=None, write_res_password=None,
                    ignore_read_only_recommended=None, origin=None, delimiter=None,
                    editable=None, notify=None, converter=None, add_to_mru=None, local=None,
                    corrupt_load=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[10559] for existing apps,
↳ get the PIDs via xw.apps.keys()
>>> 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>

### Parameters

- **fullname** (*str or path-like object, 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.
- **update\_links** (*bool, default None*) – If this argument is omitted, the user is prompted to specify how links will be updated
- **read\_only** (*bool, default False*) – True to open workbook in read-only mode
- **format** (*str*) – If opening a text file, this specifies the delimiter character
- **password** (*str*) – Password to open a protected workbook
- **write\_res\_password** (*str*) – Password to write to a write-reserved workbook
- **ignore\_read\_only\_recommended** (*bool, default False*) – Set to True to mute the read-only recommended message
- **origin** (*int*) – For text files only. Specifies where it originated. Use `XIPlatform` constants.

- **delimiter** (*str*) – If format argument is 6, this specifies the delimiter.
- **editable** (*bool*, *default False*) – This option is only for legacy Microsoft Excel 4.0 addins.
- **notify** (*bool*, *default False*) – Notify the user when a file becomes available. If the file cannot be opened in read/write mode.
- **converter** (*int*) – The index of the first file converter to try when opening the file.
- **add\_to\_mru** (*bool*, *default False*) – Add this workbook to the list of recently added workbooks.
- **local** (*bool*, *default False*) – If True, saves files against the language of Excel, otherwise against the language of VBA. Not supported on macOS.
- **corrupt\_load** (*int*, *default xlNormalLoad*) – Can be one of `xlNormalLoad`, `xlRepairFile` or `xlExtractData`. Not supported on macOS.

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

Activates the book.

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

**property api**

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

New in version 0.9.0.

**property 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.:

```
import xlwings as xw

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

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.

**property 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.

**property name**

Returns the name of the book as str.

**property 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.

**render\_template**(*\*\*data*)

This method requires xlwings *PRO*.

Replaces all Jinja variables (e.g {{ myvar }}) in the book with the keyword argument that has the same name.

New in version 0.25.0.

**Parameters** **data** (*kwargs*) – All key/value pairs that are used in the template.

## Examples

```
>>> import xlwings as xw
>>> book = xw.Book()
>>> book.sheets[0]['A1:A2'].value = '{{ myvar }}'
>>> book.render_template(myvar='test')
```

**save**(*path=None, password=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 or path-like object, default None*) – Full path to the workbook
- **password**(*str, default None*) – Protection password with max. 15 characters

New in version 0.25.1.

## 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.

### property 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():
```

(continues on next page)

(continued from previous page)

```

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.

### property sheets

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

New in version 0.9.0.

**to\_pdf**(*path=None, include=None, exclude=None, layout=None, exclude\_start\_string='#', show=False*)

Exports the whole Excel workbook or a subset of the sheets to a PDF file. If you want to print hidden sheets, you will need to list them explicitly under *include*.

#### Parameters

- **path** (*str or path-like object, default None*) – Path to the PDF file, defaults to the same name as the workbook, in the same directory. For unsaved workbooks, it defaults to the current working directory instead.
- **include** (*int or str or list, default None*) – Which sheets to include: provide a selection of sheets in the form of sheet indices (1-based like in Excel) or sheet names. Can be an int/str for a single sheet or a list of int/str for multiple sheets.
- **exclude** (*int or str or list, default None*) – Which sheets to exclude: provide a selection of sheets in the form of sheet indices (1-based like in Excel) or sheet names. Can be an int/str for a single sheet or a list of int/str for multiple sheets.
- **layout** (*str or path-like object, default None*) – This argument requires xlwings *PRO*.

Path to a PDF file on which the report will be printed. This is ideal for headers and footers as well as borderless printing of graphics/artwork. The PDF file either needs to have only 1 page (every report page uses the same layout) or otherwise needs the same amount of pages as the report (each report page is printed on the respective page in the layout PDF).

New in version 0.24.3.

- **exclude\_start\_string** (*str, default '#'*) – Sheet names that start with this character/string will not be printed.

New in version 0.24.4.

- **show** (*bool, default False*) – Once created, open the PDF file with the default application.

New in version 0.24.6.

### Examples

```
>>> wb = xw.Book()
>>> wb.sheets[0]['A1'].value = 'PDF'
>>> wb.to_pdf()
```

See also `xlwings.Sheet.to_pdf()`

New in version 0.21.1.

## 28.2.5 PageSetup

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

### property `api`

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

New in version 0.24.2.

### property `print_area`

Gets or sets the range address that defines the print area.

### Examples

```
>>> mysheet.page_setup.print_area = '$A$1:$B$3'
>>> mysheet.page_setup.print_area
'$A$1:$B$3'
>>> mysheet.page_setup.print_area = None # clear the print_area
```

New in version 0.24.2.

## 28.2.6 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.Book('Book1').sheets # specific book
Sheets([<Sheet [Book1]Sheet1>, <Sheet [Book1]Sheet2>])
```

New in version 0.9.0.

**property 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.

## 28.2.7 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.

**property 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.

### property book

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

### property 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.

### property 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.

### copy(*before=None, after=None, name=None*)

Copy a sheet to the current or a new Book. By default, it places the copied sheet after all existing sheets in the current Book. Returns the copied sheet.

New in version 0.22.0.

#### Parameters

- **before** (*sheet object, default None*) – The sheet object before which you want to place the sheet
- **after** (*sheet object, default None*) – The sheet object after which you want to place the sheet, by default it is placed after all existing sheets
- **name** (*str, default None*) – The sheet name of the copy

**Returns** Sheet object – The copied sheet

**Return type** [Sheet](#)

## Examples

```
# Create two books and add a value to the first sheet of the first book
first_book = xw.Book()
second_book = xw.Book()
first_book.sheets[0]['A1'].value = 'some value'

# Copy to same Book with the default location and name
first_book.sheets[0].copy()

# Copy to same Book with custom sheet name
first_book.sheets[0].copy(name='copied')

# Copy to second Book requires to use before or after
first_book.sheets[0].copy(after=second_book.sheets[0])
```

### **delete()**

Deletes the Sheet.

### **property index**

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

### **property name**

Gets or sets the name of the Sheet.

### **property names**

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

New in version 0.9.0.

### **property page\_setup**

Returns a PageSetup object.

New in version 0.24.2.

### **property 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.

### **render\_template(\*\*data)**

This method requires xlwings *PRO*.

Replaces all Jinja variables (e.g `{{ myvar }}`) in the sheet with the keyword argument that has the same name. Following variable types are supported:

strings, numbers, lists, simple dicts, NumPy arrays, Pandas DataFrames, PIL Image objects that have a filename and Matplotlib figures.

New in version 0.22.0.

**Parameters** **data** (*kwargs*) – All key/value pairs that are used in the template.

## Examples

```
>>> import xlwings as xw
>>> book = xw.Book()
>>> book.sheets[0]['A1:A2'].value = '{{ myvar }}'
>>> book.sheets[0].render_template(myvar='test')
```

### **select()**

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

New in version 0.9.0.

### **property shapes**

See [Shapes](#)

New in version 0.9.0.

### **property tables**

See [Tables](#)

New in version 0.21.0.

### **to\_pdf(path=None, layout=None, show=False)**

Exports the sheet to a PDF file.

#### **Parameters**

- **path** (*str or path-like object, default None*) – Path to the PDF file, defaults to the name of the sheet in the same directory of the workbook. For unsaved workbooks, it defaults to the current working directory instead.
- **layout** (*str or path-like object, default None*) – This argument requires xlwings *PRO*.

Path to a PDF file on which the report will be printed. This is ideal for headers and footers as well as borderless printing of graphics/artwork. The PDF file either needs to have only 1 page (every report page uses the same layout) or otherwise needs the same amount of pages as the report (each report page is printed on the respective page in the layout PDF).

New in version 0.24.3.

- **show** (*bool, default False*) – Once created, open the PDF file with the default application.

New in version 0.24.6.

## Examples

```
>>> wb = xw.Book()
>>> sheet = wb.sheets[0]
>>> sheet['A1'].value = 'PDF'
>>> sheet.to_pdf()
```

See also `xlwings.Book.to_pdf()`

New in version 0.22.3.

### property `used_range`

Used Range of Sheet.

#### Returns

**Return type** `xw.Range`

New in version 0.13.0.

### property `visible`

Gets or sets the visibility of the Sheet (bool).

New in version 0.21.1.

## 28.2.8 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.

**property 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.

**property 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.

**property 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 hex string like #efefef or an Excel 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) # or '#ffffff'
>>> 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.

### property 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.

### property 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.

### property columns

Returns a [RangeColumns](#) object that represents the columns in the specified range.

New in version 0.9.0.

### copy(*destination=None*)

Copy a range to a destination range or clipboard.

**Parameters** **destination** ([xlwings.Range](#)) – xlwings Range to which the specified range will be copied. If omitted, the range is copied to the Clipboard.

#### Returns

**Return type** None

**copy\_picture**(*appearance='screen', format='picture'*)

Copies the range to the clipboard as picture.

**Parameters**

- **appearance** (*str*, *default 'screen'*) – Either ‘screen’ or ‘printer’.
- **format** (*str*, *default 'picture'*) – Either ‘picture’ or ‘bitmap’.
- **versionadded:** (.) – 0.24.8:

**property count**

Returns the number of cells.

**property 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

**delete**(*shift=None*)

Deletes a cell or range of cells.

**Parameters** **shift** (*str*, *default None*) – Use left or up. If omitted, Excel decides based on the shape of the range.

**Returns**

**Return type** None

**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.

### property formula

Gets or sets the formula for the given Range.

### property formula2

Gets or sets the formula2 for the given Range.

### property 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.

### property `has_array`

Are we part of an Array formula?

### property `height`

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

#### Returns

**Return type** float

New in version 0.4.0.

### property `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.

### `insert(shift=None, copy_origin='format_from_left_or_above')`

Insert a cell or range of cells into the sheet.

#### Parameters

- **shift** (*str*, *default None*) – Use right or down. If omitted, Excel decides based on the shape of the range.
- **copy\_origin** (*str*, *default format\_from\_left\_or\_above*) – Use `format_from_left_or_above` or `format_from_right_or_below`. Note that this is not supported on macOS.

**Returns****Return type** None**property 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.

**property 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.

**merge(*across=False*)**

Creates a merged cell from the specified Range object.

**Parameters** **across** (*bool*, *default False*) – True to merge cells in each row of the specified Range as separate merged cells.**property merge\_area**

Returns a Range object that represents the merged Range containing the specified cell. If the specified cell isn't in a merged range, this property returns the specified cell.

**property merge\_cells**

Returns True if the Range contains merged cells, otherwise False

**property name**

Sets or gets the name of a Range.

New in version 0.4.0.

**property note**

Returns a Note object. Before the introduction of threaded comments, a Note was called a Comment.

New in version 0.24.2.

**property 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'
- **chunksize** (*int*) –

Use a chunksize, e.g. 10000 to prevent timeout or memory issues when reading or writing large amounts of data. Works with all formats, including DataFrames, NumPy arrays, and list of lists.

=> For converter-specific options, see [Converters and Options](#).

**Returns****Return type** Range object

New in version 0.7.0.

**paste**(*paste=None, operation=None, skip\_blanks=False, transpose=False*)

Pastes a range from the clipboard into the specified range.

**Parameters**

- **paste**(*str, default None*) – One of `all_merging_conditional_formats`, `all`, `all_except_borders`, `all_using_source_theme`, `column_widths`, `comments`, `formats`, `formulas`, `formulas_and_number_formats`, `validation`, `values`, `values_and_number_formats`.
- **operation**(*str, default None*) – One of “add”, “divide”, “multiply”, “subtract”.
- **skip\_blanks**(*bool, default False*) – Set to True to skip over blank cells
- **transpose**(*bool, default False*) – Set to True to transpose rows and columns.

**Returns****Return type** None**property 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.

**property row**

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

**Returns****Return type** Integer

New in version 0.3.5.

**property 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.

**property 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.

**property shape**

Tuple of Range dimensions.

New in version 0.3.0.

**property sheet**

Returns the Sheet object to which the Range belongs.

New in version 0.9.0.

**property size**

Number of elements in the Range.

New in version 0.3.0.

**property table**

Returns a Table object if the range is part of one, otherwise None.

New in version 0.21.0.

**to\_png(path=None)**

Exports the range as PNG picture.

**Parameters**

- **path** (*str or path-like, default None*) – Path where you want to store the picture. Defaults to the name of the range in the same directory as the Excel file if the Excel file is stored and to the current working directory otherwise.
- **versionadded:** (.) – 0.24.8:

**property 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.

**unmerge()**

Separates a merged area into individual cells.

**property value**Gets and sets the values for the given Range. See [xlwings.Range.options\(\)](#) about how to set options, e.g. to transform it into a DataFrame or how to set a chunksize.**Returns object****Return type** returned object depends on the converter being used, see [xlwings.Range.options\(\)](#)**property width**

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

**Returns****Return type** float

New in version 0.4.0.

**property wrap\_text**

Returns True if the wrap\_text property is enabled and False if it's disabled. If not all cells have the same value in a range, on Windows it returns None and on macOS False.

New in version 0.23.2.

## 28.2.9 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')
```

(continues on next page)

(continued from previous page)

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

**autofit()**

Autofits the height of the rows.

**property count**

Returns the number of rows.

New in version 0.9.0.

### 28.2.10 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.

**property count**

Returns the number of columns.

New in version 0.9.0.

### 28.2.11 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.

**property** api

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

**property** count

Returns the number of objects in the collection.

### 28.2.12 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.

**property** api

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

New in version 0.19.2.

**delete**()

Deletes the shape.

New in version 0.5.0.

**property** height

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

New in version 0.5.0.

**property** left

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

New in version 0.5.0.

**property name**

Returns or sets the name of the shape.

New in version 0.5.0.

**property parent**

Returns the parent of the shape.

New in version 0.9.0.

**scale\_height**(*factor*, *relative\_to\_original\_size=False*, *scale='scale\_from\_top\_left'*)

**factor** [float] For example 1.5 to scale it up to 150%

**relative\_to\_original\_size** [bool, optional] If `False`, it scales relative to current height (default).  
For `True` must be a picture or OLE object.

**scale** [str, optional] One of `scale_from_top_left` (default), `scale_from_bottom_right`,  
`scale_from_middle`

New in version 0.19.2.

**scale\_width**(*factor*, *relative\_to\_original\_size=False*, *scale='scale\_from\_top\_left'*)

**factor** [float] For example 1.5 to scale it up to 150%

**relative\_to\_original\_size** [bool, optional] If `False`, it scales relative to current width (default).  
For `True` must be a picture or OLE object.

**scale** [str, optional] One of `scale_from_top_left` (default), `scale_from_bottom_right`,  
`scale_from_middle`

New in version 0.19.2.

**property text**

Returns or sets the text of a shape.

New in version 0.21.4.

**property top**

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

New in version 0.5.0.

**property type**

Returns the type of the shape.

New in version 0.9.0.

**property width**

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

New in version 0.5.0.

### 28.2.13 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'
```

#### property **api**

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

#### property **count**

Returns the number of objects in the collection.

## 28.2.14 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>>
```

### property `api`

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

New in version 0.9.0.

### property `chart_type`

Returns and sets the chart type of the chart. The following chart types are available:

3d\_area, 3d\_area\_stacked, 3d\_area\_stacked\_100, 3d\_bar\_clustered,  
3d\_bar\_stacked, 3d\_bar\_stacked\_100, 3d\_column, 3d\_column\_clustered,  
3d\_column\_stacked, 3d\_column\_stacked\_100, 3d\_line, 3d\_pie, 3d\_pie\_exploded,  
area, area\_stacked, area\_stacked\_100, bar\_clustered, bar\_of\_pie,  
bar\_stacked, bar\_stacked\_100, bubble, bubble\_3d\_effect, column\_clustered,  
column\_stacked, column\_stacked\_100, combination, cone\_bar\_clustered,  
cone\_bar\_stacked, cone\_bar\_stacked\_100, cone\_col, cone\_col\_clustered,  
cone\_col\_stacked, cone\_col\_stacked\_100, cylinder\_bar\_clustered,  
cylinder\_bar\_stacked, cylinder\_bar\_stacked\_100, cylinder\_col,  
cylinder\_col\_clustered, cylinder\_col\_stacked, cylinder\_col\_stacked\_100,  
doughnut, doughnut\_exploded, line, line\_markers, line\_markers\_stacked,  
line\_markers\_stacked\_100, line\_stacked, line\_stacked\_100, pie,  
pie\_exploded, pie\_of\_pie, pyramid\_bar\_clustered, pyramid\_bar\_stacked,  
pyramid\_bar\_stacked\_100, pyramid\_col, pyramid\_col\_clustered,  
pyramid\_col\_stacked, pyramid\_col\_stacked\_100, radar, radar\_filled,  
radar\_markers, stock\_hlc, stock\_ohlc, stock\_vhlc, stock\_vohlc, surface,  
surface\_top\_view, surface\_top\_view\_wireframe, surface\_wireframe, xy\_scatter,  
xy\_scatter\_lines, xy\_scatter\_lines\_no\_markers, xy\_scatter\_smooth,  
xy\_scatter\_smooth\_no\_markers

New in version 0.1.1.

### `delete()`

Deletes the chart.

### property `height`

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

### property `left`

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

### property `name`

Returns or sets the name of the chart.

**property 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')`

**to\_png**(*path=None*)

Exports the chart as PNG picture.

**Parameters**

- **path** (*str or path-like, default None*) – Path where you want to store the picture. Defaults to the name of the chart in the same directory as the Excel file if the Excel file is stored and to the current working directory otherwise.
- **versionadded:** (.) – 0.24.8:

**property top**

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

**property width**

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

## 28.2.15 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=None, top=None, width=None, height=None, name=None, update=False, scale=None, format=None, anchor=None*)

Adds a picture to the specified sheet.

**Parameters**

- **image** (*str or path-like object or matplotlib.figure.Figure*) – Either a filepath or a Matplotlib figure object.
- **left** (*float, default None*) – Left position in points, defaults to 0. If you use top/left, you must not provide a value for anchor.
- **top** (*float, default None*) – Top position in points, defaults to 0. If you use top/left, you must not provide a value for anchor.

- **width** (*float, default None*) – Width in points. Defaults to original width.
- **height** (*float, default None*) – Height in points. Defaults to original height.
- **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.
- **scale** (*float, default None*) – Scales your picture by the provided factor.
- **format** (*str, default None*) – Only used if image is a Matplotlib or Plotly plot. By default, the plot is inserted in the “png” format, but you may want to change this to a vector-based format like “svg” on Windows (may require Microsoft 365) or “eps” on macOS for better print quality. If you use 'vector', it will be using 'svg' on Windows and 'eps' on macOS. To find out which formats your version of Excel supports, see: <https://support.microsoft.com/en-us/topic/support-for-eps-images-has-been-turned-off-in-office-a069d664-4bcf-415e-a1b5-cbb0c334a840>
- **anchor** (*xw.Range, default None*) – The xlwings Range object of where you want to insert the picture. If you use anchor, you must not provide values for top/left.

New in version 0.24.3.

## Returns

Return type *Picture*

## Examples

### 1. Picture

```
>>> import xlwings as xw
>>> sht = xw.Book().sheets[0]
>>> sht.pictures.add(r'C:\path\to\file.png')
<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>>
```

**property api**

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

**property count**

Returns the number of objects in the collection.

**28.2.16 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.

**property 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.

**property height**

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

New in version 0.5.0.

**property left**

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

New in version 0.5.0.

**property lock\_aspect\_ratio**

True will keep the original proportion, False will allow you to change height and width independently of each other (read/write).

New in version 0.24.0.

**property name**

Returns or sets the name of the picture.

New in version 0.5.0.

**property parent**

Returns the parent of the picture.

New in version 0.9.0.

**property top**

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

New in version 0.5.0.

**update**(*image*, *format=None*)

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

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

New in version 0.5.0.

**property width**

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

New in version 0.5.0.

## 28.2.17 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.

**property api**

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

New in version 0.9.0.

**property count**

Returns the number of objects in the collection.

### 28.2.18 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.

**property** `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.

**property** `name`

Returns or sets the name of the name object.

New in version 0.9.0.

**property** `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.

**property** `refers_to_range`

Returns the Range object referred to by a Name object.

New in version 0.9.0.

### 28.2.19 Note

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

**property** `api`

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

New in version 0.24.2.

**delete**()

Delete the note.

New in version 0.24.2.

**property** `text`

Gets or sets the text of a note. Keep in mind that the note must already exist!

## Examples

```
>>> sheet = xw.Book(...).sheets[0]
>>> sheet['A1'].note.text = 'mynote'
>>> sheet['A1'].note.text
>>> 'mynote'
```

New in version 0.24.2.

## 28.2.20 Tables

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

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

```
>>> import xlwings as xw
>>> xw.books['Book1'].sheets[0].tables
Tables([<Table 'Table1' in <Sheet [Book1]Sheet1>>, <Table 'Table2' in
↳<Sheet [Book1]Sheet1>>])
```

New in version 0.21.0.

**add**(*source=None, name=None, source\_type=None, link\_source=None, has\_headers=True, destination=None, table\_style\_name='TableStyleMedium2'*)

Creates a Table to the specified sheet.

### Parameters

- **source** (*xlwings range, default None*) – An xlwings range object, representing the data source.
- **name** (*str, default None*) – The name of the Table. By default, it uses the autogenerated name that is assigned by Excel.
- **source\_type** (*str, default None*) – This currently defaults to `xlSrcRange`, i.e. expects an xlwings range object. No other options are allowed at the moment.
- **link\_source** (*bool, default None*) – Currently not implemented as this is only in case `source_type` is `xlSrcExternal`.
- **has\_headers** (*bool or str, default True*) – Indicates whether the data being imported has column labels. Defaults to `True`. Possible values: `True`, `False`, `'guess'`
- **destination** (*xlwings range, default None*) – Currently not implemented as this is used in case `source_type` is `xlSrcExternal`.
- **table\_style\_name** (*str, default 'TableStyleMedium2'*) – Possible strings: `'TableStyleLightN'` (where N is 1-21), `'TableStyleMediumN'` (where N is 1-28), `'TableStyleDarkN'` (where N is 1-11)

### Returns

Return type *Table*

### Examples

```
>>> import xlwings as xw
>>> sheet = xw.Book().sheets[0]
>>> sheet['A1'].value = [['a', 'b'], [1, 2]]
>>> table = sheet.tables.add(source=sheet['A1'].expand(), name='MyTable
↳')
>>> table
<Table 'MyTable' in <Sheet [Book1]Sheet1>>
```

## 28.2.21 Table

**class** xlwings.main.Table(\*args, \*\*options)

The table object is a member of the *tables* collection:

```
>>> import xlwings as xw
>>> sht = xw.books['Book1'].sheets[0]
>>> sht.tables[0] # or sht.tables['TableName']
<Table 'Table 1' in <Sheet [Book1]Sheet1>>
```

New in version 0.21.0.

#### property api

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

#### property data\_body\_range

Returns an xlwings range object that represents the range of values, excluding the header row

#### property display\_name

Returns or sets the display name for the specified Table object

#### property header\_row\_range

Returns an xlwings range object that represents the range of the header row

#### property insert\_row\_range

Returns an xlwings range object representing the row where data is going to be inserted. This is only available for empty tables, otherwise it'll return None

#### property name

Returns or sets the name of the Table.

#### property parent

Returns the parent of the table.

#### property range

Returns an xlwings range object of the table.

**resize**(*range*)

Resize a Table by providing an xlwings range object

New in version 0.24.4.

**property show\_autofilter**

Turn the autofilter on or off by setting it to True or False (read/write boolean)

**property show\_headers**

Show or hide the header (read/write)

**property show\_table\_style\_column\_stripes**

Returns or sets if the Column Stripes table style is used for (read/write boolean)

**property show\_table\_style\_first\_column**

Returns or sets if the first column is formatted (read/write boolean)

**property show\_table\_style\_last\_column**

Returns or sets if the last column is displayed (read/write boolean)

**property show\_table\_style\_row\_stripes**

Returns or sets if the Row Stripes table style is used (read/write boolean)

**property show\_totals**

Gets or sets a boolean to show/hide the Total row.

**property table\_style**

Gets or sets the table style. See [Tables.add](#) for possible values.

**property totals\_row\_range**

Returns an xlwings range object representing the Total row

**update**(*data*, *index=True*)

Updates the Excel table with the provided data. Currently restricted to DataFrames.

Changed in version 0.24.0.

**Parameters**

- **data** (*pandas DataFrame*) – Currently restricted to pandas DataFrames.
- **index** (*bool*, *default True*) – Whether or not the index of a pandas DataFrame should be written to the Excel table.

**Returns**

**Return type** *Table*

## Examples

```
import pandas as pd
import xlwings as xw

sheet = xw.Book('Book1.xlsx').sheets[0]
table_name = 'mytable'

# Sample DataFrame
nrows, ncols = 3, 3
df = pd.DataFrame(data=nrows * [ncols * ['test']],
                  columns=['col ' + str(i) for i in range(ncols)])

# Hide the index, then insert a new table if it doesn't exist yet,
# otherwise update the existing one
df = df.set_index('col 0')
if table_name in [table.name for table in sheet.tables]:
    sheet.tables[table_name].update(df)
else:
    mytable = sheet.tables.add(source=sheet['A1'], name=table_name).
    ↪update(df)
```

### 28.2.22 Font

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

The font object can be accessed as an attribute of the range or shape object.

- `mysheet['A1'].font`
- `mysheet.shapes[0].font`

New in version 0.23.0.

#### property api

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

New in version 0.23.0.

#### property bold

Returns or sets the bold property (boolean).

```
>>> sheet['A1'].font.bold = True
>>> sheet['A1'].font.bold
True
```

New in version 0.23.0.

#### property color

Returns or sets the color property (tuple).

```
>>> sheet['A1'].font.color = (255, 0, 0) # or '#ff0000'
>>> sheet['A1'].font.color
(255, 0, 0)
```

New in version 0.23.0.

**property italic**

Returns or sets the italic property (boolean).

```
>>> sheet['A1'].font.italic = True
>>> sheet['A1'].font.italic
True
```

New in version 0.23.0.

**property name**

Returns or sets the name of the font (str).

```
>>> sheet['A1'].font.name = 'Calibri'
>>> sheet['A1'].font.name
Calibri
```

New in version 0.23.0.

**property size**

Returns or sets the size (float).

```
>>> sheet['A1'].font.size = 13
>>> sheet['A1'].font.size
13
```

New in version 0.23.0.

## 28.2.23 Characters

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

The characters object can be accessed as an attribute of the range or shape object.

- `mysheet['A1'].characters`
- `mysheet.shapes[0].characters`

---

**Note:** On macOS, characters are currently not supported due to bugs/lack of support in AppleScript.

---

New in version 0.23.0.

**property api**

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

New in version 0.23.0.

#### property **font**

Returns or sets the text property of a characters object.

```
>>> sheet['A1'].characters[1:3].font.bold = True
>>> sheet['A1'].characters[1:3].font.bold
True
```

New in version 0.23.0.

#### property **text**

Returns or sets the text property of a characters object.

```
>>> sheet['A1'].value = 'Python'
>>> sheet['A1'].characters[:3].text
Pyt
```

New in version 0.23.0.

## 28.2.24 Markdown

```
class xlwings.pro.Markdown(text, style=<MarkdownStyle> h1.font: .bold: True
                           paragraph.blank_lines_after: 1 unordered_list.bullet_character: •
                           unordered_list.blank_lines_after: 1 strong.bold: True emphasis.italic:
                           True)
```

Markdown objects can be assigned to a single cell or shape via `myrange.value` or `myshape.text`. They accept a string in Markdown format which will cause the text in the cell to be formatted accordingly. They can also be used in `mysheet.render_template()`.

---

**Note:** On macOS, formatting is currently not supported, but things like bullet points will still work.

---

#### Parameters

- **text** (*str*) – The text in Markdown syntax
- **style** (*MarkdownStyle object, optional*) – The `MarkdownStyle` object defines how the text will be formatted.

## Examples

```
>>> mysheet['A1'].value = Markdown("A text with *emphasis* and **strong**  
↪style.")  
>>> myshape.text = Markdown("A text with *emphasis* and **strong** style.")
```

New in version 0.23.0.

### 28.2.25 MarkdownStyle

#### `class xlwings.pro.MarkdownStyle`

MarkdownStyle defines how Markdown objects are being rendered in Excel cells or shapes. Start by instantiating a MarkdownStyle object. Printing it will show you the current (default) style:

```
>>> style = MarkdownStyle()  
>>> style  
<MarkdownStyle>  
h1.font: .bold: True  
h1.blank_lines_after: 1  
paragraph.blank_lines_after: 1  
unordered_list.bullet_character: •  
unordered_list.blank_lines_after: 1  
strong.bold: True  
emphasis.italic: True
```

You can override the defaults, e.g., to make **strong** text red instead of bold, do this:

```
>>> style.strong.bold = False  
>>> style.strong.color = (255, 0, 0)  
>>> style.strong  
strong.color: (255, 0, 0)
```

New in version 0.23.0.

## 28.3 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:

---

**Note:** If your version of Excel supports the new native dynamic arrays, then you don't have to do anything special, and you shouldn't use the `expand` decorator! To check if your version of Excel supports it, see if you have the `=UNIQUE()` formula available. Native dynamic arrays were introduced in Office 365 Insider Fast at the end of September 2018.

---

`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.

## 28.4 Reports

**class** xlwings.pro.reports.**Image**(filename)

Use this class to provide images to either `render_template()`.

**Parameters** **filename** (*str* or *pathlib.Path* object) – The file name or path

**class** xlwings.pro.reports.**Markdown**(text, style=<MarkdownStyle> *h1.font: .bold: True*  
*paragraph.blank\_lines\_after: 1*  
*unordered\_list.bullet\_character: •*  
*unordered\_list.blank\_lines\_after: 1 strong.bold: True*  
*emphasis.italic: True*)

Markdown objects can be assigned to a single cell or shape via `myrange.value` or `myshape.text`. They accept a string in Markdown format which will cause the text in the cell to be formatted accordingly. They can also be used in `mysheet.render_template()`.

---

**Note:** On macOS, formatting is currently not supported, but things like bullet points will still work.

---

### Parameters

- **text** (*str*) – The text in Markdown syntax
- **style** (*MarkdownStyle* object, *optional*) – The *MarkdownStyle* object defines how the text will be formatted.

## Examples

```
>>> mysheet['A1'].value = Markdown("A text with *emphasis* and **strong**↵
↵style.")
>>> myshape.text = Markdown("A text with *emphasis* and **strong** style.")
```

New in version 0.23.0.

### class xlwings.pro.reports.MarkdownStyle

MarkdownStyle defines how Markdown objects are being rendered in Excel cells or shapes. Start by instantiating a MarkdownStyle object. Printing it will show you the current (default) style:

```
>>> style = MarkdownStyle()
>>> style
<MarkdownStyle>
h1.font: .bold: True
h1.blank_lines_after: 1
paragraph.blank_lines_after: 1
unordered_list.bullet_character: •
unordered_list.blank_lines_after: 1
strong.bold: True
emphasis.italic: True
```

You can override the defaults, e.g., to make **\*\*strong\*\*** text red instead of bold, do this:

```
>>> style.strong.bold = False
>>> style.strong.color = (255, 0, 0)
>>> style.strong
strong.color: (255, 0, 0)
```

New in version 0.23.0.

### xlwings.pro.reports.render\_template(template, output, book\_settings=None, app=None, \*\*data)

This function requires xlwings *PRO*.

This is a convenience wrapper around [mysheet.render\\_template](#)

Writes the values of all key word arguments to the output file according to the template and the variables contained in there (Jinja variable syntax). Following variable types are supported:

strings, numbers, lists, simple dicts, NumPy arrays, Pandas DataFrames, pictures and Matplotlib/Plotly figures.

#### Parameters

- **template** (*str*) – Path to your Excel template, e.g. `r'C:\Path\to\my_template.xlsx'`
- **output** (*str*) – Path to your Report, e.g. `r'C:\Path\to\my_report.xlsx'`
- **book\_settings** (*dict*, *default None*) – A dictionary of xlwings. Book parameters, for details see: [xlwings.Book](#). For example:

```
book_settings={'update_links': False}.
```

- **app** (*xlwings App*, *default None*) – By passing in an xlwings App instance, you can control where your report runs and configure things like `visible=False`. For details see [xlwings.App](#). By default, it creates the report in the currently active instance of Excel.
- **data** (*kwargs*) – All key/value pairs that are used in the template.

### Returns

**Return type** xlwings Book

### Examples

In `my_template.xlsx`, put the following Jinja variables in two cells: `{{ title }}` and `{{ df }}`

```
>>> from xlwings.pro.reports import render_template
>>> import pandas as pd
>>> df = pd.DataFrame(data=[[1,2],[3,4]])
>>> mybook = render_template('my_template.xlsx', 'my_report.xlsx', title=
↪ 'MyTitle', df=df)
```

With many template variables it may be useful to collect the data first:

```
>>> data = dict(title='MyTitle', df=df)
>>> mybook = render_template('my_template.xlsx', 'my_report.xlsx', **data)
```

If you need to handle external links or a password, use it like so:

```
>>> mybook = render_template('my_template.xlsx', 'my_report.xlsx',
                             book_settings={'update_links': True, 'password':
↪ 'mypassword'},
                             **data)
```

## REST API

---

**Note:** This is an experimental feature and may be removed in the future.

---

New in version 0.13.0.

### 29.1 Quickstart

xlwings offers an easy way to expose an Excel workbook via REST API both on Windows and macOS. This can be useful when you have a workbook running on a single computer and want to access it from another computer. Or you can build a Linux based web app that can interact with a legacy Excel application while you are in the progress of migrating the Excel functionality into your web app (if you need help with that, [give us a shout](#)).

You can run the REST API server from a command prompt or terminal as follows (this requires Flask>=1.0, so make sure to `pip install Flask`):

```
xlwings restapi run
```

Then perform a GET request e.g. via PowerShell on Windows or Terminal on Mac (while having an unsaved “Book1” open). Note that you need to run the server and the GET request from two separate terminals (or you can use something more convenient like [Postman](#) or [Insomnia](#) for testing the API):

```
$ curl "http://127.0.0.1:5000/book/book1/sheets/0/range/A1:B2"
{
  "address": "$A$1:$B$2",
  "color": null,
  "column": 1,
  "column_width": 10.0,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    [
      "1",
      "2"
    ]
  ]
}
```

(continues on next page)

(continued from previous page)

```
    ],
    [
        "3",
        "4"
    ]
],
"formula_array": null,
"height": 32.0,
"last_cell": "$B$2",
"left": 0.0,
"name": null,
"number_format": "General",
"row": 1,
"row_height": 16.0,
"shape": [
    2,
    2
],
"size": 4,
"top": 0.0,
"value": [
    [
        1.0,
        2.0
    ],
    [
        3.0,
        4.0
    ]
],
"width": 130.0
}
```

In the command prompt where your server is running, press **Ctrl-C** to shut it down again.

The xlwings REST API is a thin wrapper around the [Python API](#) which makes it very easy if you have worked previously with xlwings. It also means that the REST API does require the Excel application to be up and running which makes it a great choice if the data in your Excel workbook is constantly changing as the REST API will always deliver the current state of the workbook without the need of saving it first.

---

**Note:** Currently, we only provide the GET methods to read the workbook. If you are also interested in the POST methods to edit the workbook, let us know via GitHub issues. Some other things will also need improvement, most notably exception handling.

---

## 29.2 Run the server

`xlwings restapi run` will run a Flask development server on <http://127.0.0.1:5000>. You can provide `--host` and `--port` as command line args and it also respects the Flask environment variables like `FLASK_ENV=development`.

If you want to have more control, you can run the server directly with Flask, see the [Flask docs](#) for more details:

```
set FLASK_APP=xlwings.rest.api
flask run
```

If you are on Mac, use `export FLASK_APP=xlwings.rest.api` instead of `set FLASK_APP=xlwings.rest.api`.

For production, you can use any WSGI HTTP Server like [gunicorn](#) (on Mac) or [waitress](#) (on Mac/Windows) to serve the API. For example, with gunicorn you would do: `gunicorn xlwings.rest.api:api`. Or with waitress (adjust the host accordingly if you want to make the api accessible from outside of localhost):

```
from xlwings.rest.api import api
from waitress import serve
serve(wsgiapp, host='127.0.0.1', port=5000)
```

## 29.3 Indexing

While the Python API offers Python's 0-based indexing (e.g. `xw.books[0]`) as well as Excel's 1-based indexing (e.g. `xw.books(1)`), the REST API only offers 0-based indexing, e.g. `/books/0`.

## 29.4 Range Options

The REST API accepts Range options as query parameters, see [xlwings.Range.options\(\)](#) e.g.

`/book/book1/sheets/0/range/A1?expand=table&transpose=true`

Remember that options only affect the value property.

## 29.5 Endpoint overview

End-point	Corresponds to	Short Description
<a href="#">/book</a>	<a href="#">Book</a>	Finds your workbook across all open instances of Excel and will open it if it can't find it
<a href="#">/books</a>	<a href="#">Books</a>	Books collection of the active Excel instance
<a href="#">/apps</a>	<a href="#">Apps</a>	This allows you to specify the Excel instance you want to work with

## 29.6 Endpoint details

### 29.6.1 /book

GET /book/<fullname\_or\_name>

Example response:

```
{
  "app": 1104,
  "fullname": "C:\\\\Users\\felix\\DEV\\xlwings\\scripts\\Book1.xlsx",
  "name": "Book1.xlsx",
  "names": [
    "Sheet1!myname1",
    "myname2"
  ],
  "selection": "Sheet2!$A$1",
  "sheets": [
    "Sheet1",
    "Sheet2"
  ]
}
```

GET /book/<fullname\_or\_name>/names

Example response:

```
{
  "names": [
    {
      "name": "Sheet1!myname1",
      "refers_to": "=Sheet1!$B$2:$C$3"
    },
    {
      "name": "myname2",
      "refers_to": "=Sheet1!$A$1"
    }
  ]
}
```

GET /book/<fullname\_or\_name>/names/<name>

Example response:

```
{
  "name": "myname2",
  "refers_to": "=Sheet1!$A$1"
}
```

GET /book/<fullname\_or\_name>/names/<name>/range

Example response:

```
{
  "address": "$A$1",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 1,
  "current_region": "$A$1:$B$2",
  "formula": "=1+1.1",
  "formula_array": "=1+1,1",
  "height": 14.25,
  "last_cell": "$A$1",
  "left": 0.0,
  "name": "myname2",
  "number_format": "General",
  "row": 1,
  "row_height": 14.3,
  "shape": [
    1,
    1
  ],
  "size": 1,
  "top": 0.0,
  "value": 2.1,
  "width": 51.0
}
```

GET /book/<fullname\_or\_name>/sheets

Example response:

```
{
  "sheets": [
    {
      "charts": [
        "Chart 1"
      ],
      "name": "Sheet1",
      "names": [
        "Sheet1!myname1"
      ],
      "pictures": [
        "Picture 3"
      ],
      "shapes": [
```

(continues on next page)

(continued from previous page)

```

        "Chart 1",
        "Picture 3"
    ],
    "used_range": "$A$1:$B$2"
},
{
    "charts": [],
    "name": "Sheet2",
    "names": [],
    "pictures": [],
    "shapes": [],
    "used_range": "$A$1"
}
]
}

```

GET /book/<fullname\_or\_name>/sheets/<sheet\_name\_or\_ix>

Example response:

```

{
  "charts": [
    "Chart 1"
  ],
  "name": "Sheet1",
  "names": [
    "Sheet1!myname1"
  ],
  "pictures": [
    "Picture 3"
  ],
  "shapes": [
    "Chart 1",
    "Picture 3"
  ],
  "used_range": "$A$1:$B$2"
}

```

GET /book/<fullname\_or\_name>/sheets/<sheet\_name\_or\_ix>/charts

Example response:

```

{
  "charts": [
    {
      "chart_type": "line",
      "height": 211.0,

```

(continues on next page)

(continued from previous page)

```

    "left": 0.0,
    "name": "Chart 1",
    "top": 0.0,
    "width": 355.0
  }
]
}

```

GET /book/<fullname\_or\_name>/sheets/<sheet\_name\_or\_ix>/charts/<chart\_name\_or\_ix>

Example response:

```

{
  "chart_type": "line",
  "height": 211.0,
  "left": 0.0,
  "name": "Chart 1",
  "top": 0.0,
  "width": 355.0
}

```

GET /book/<fullname\_or\_name>/sheets/<sheet\_name\_or\_ix>/names

Example response:

```

{
  "names": [
    {
      "name": "Sheet1!myname1",
      "refers_to": "=Sheet1!$B$2:$C$3"
    }
  ]
}

```

GET /book/<fullname\_or\_name>/sheets/<sheet\_name\_or\_ix>/names/<sheet\_scope\_name>

Example response:

```

{
  "name": "Sheet1!myname1",
  "refers_to": "=Sheet1!$B$2:$C$3"
}

```

GET /book/<fullname\_or\_name>/sheets/<sheet\_name\_or\_ix>/names/<sheet\_scope\_name>/range

Example response:

```
{
  "address": "$B$2:$C$3",
  "color": null,
  "column": 2,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    [
      "",
      ""
    ],
    [
      "",
      ""
    ]
  ],
  "formula_array": "",
  "height": 28.5,
  "last_cell": "$C$3",
  "left": 51.0,
  "name": "Sheet1!myname1",
  "number_format": "General",
  "row": 2,
  "row_height": 14.3,
  "shape": [
    2,
    2
  ],
  "size": 4,
  "top": 14.25,
  "value": [
    [
      null,
      null
    ],
    [
      null,
      null
    ]
  ],
  "width": 102.0
}
```

GET /book/<fullname\_or\_name>/sheets/<sheet\_name\_or\_ix>/pictures

Example response:

```
{
  "pictures": [
    {
      "height": 100.0,
      "left": 0.0,
      "name": "Picture 3",
      "top": 0.0,
      "width": 100.0
    }
  ]
}
```

GET /book/<fullname\_or\_name>/sheets/<sheet\_name\_or\_ix>/pictures/  
<picture\_name\_or\_ix>

Example response:

```
{
  "height": 100.0,
  "left": 0.0,
  "name": "Picture 3",
  "top": 0.0,
  "width": 100.0
}
```

GET /book/<fullname\_or\_name>/sheets/<sheet\_name\_or\_ix>/range

Example response:

```
{
  "address": "$A$1:$B$2",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    [
      "=1+1.1",
      "a string"
    ],
    [
      "43395.0064583333",
      ""
    ]
  ],
  "formula_array": null,
  "height": 28.5,
```

(continues on next page)

(continued from previous page)

```

"last_cell": "$B$2",
"left": 0.0,
"name": null,
"number_format": null,
"row": 1,
"row_height": 14.3,
"shape": [
  2,
  2
],
"size": 4,
"top": 0.0,
"value": [
  [
    2.1,
    "a string"
  ],
  [
    "Mon, 22 Oct 2018 00:09:18 GMT",
    null
  ]
],
"width": 102.0
}

```

GET /book/<fullname\_or\_name>/sheets/<sheet\_name\_or\_ix>/range/<address>

Example response:

```

{
  "address": "$A$1:$B$2",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    [
      "=1+1.1",
      "a string"
    ],
    [
      "43395.0064583333",
      ""
    ]
  ],
}

```

(continues on next page)

(continued from previous page)

```

"formula_array": null,
"height": 28.5,
"last_cell": "$B$2",
"left": 0.0,
"name": null,
"number_format": null,
"row": 1,
"row_height": 14.3,
"shape": [
    2,
    2
],
"size": 4,
"top": 0.0,
"value": [
    [
        2.1,
        "a string"
    ],
    [
        "Mon, 22 Oct 2018 00:09:18 GMT",
        null
    ]
],
"width": 102.0
}

```

GET /book/<fullname\_or\_name>/sheets/<sheet\_name\_or\_ix>/shapes

Example response:

```

{
  "shapes": [
    {
      "height": 211.0,
      "left": 0.0,
      "name": "Chart 1",
      "top": 0.0,
      "type": "chart",
      "width": 355.0
    },
    {
      "height": 100.0,
      "left": 0.0,
      "name": "Picture 3",
      "top": 0.0,

```

(continues on next page)

(continued from previous page)

```
    "type": "picture",
    "width": 100.0
  }
]
```

GET /book/<fullname\_or\_name>/sheets/<sheet\_name\_or\_ix>/shapes/<shape\_name\_or\_ix>

Example response:

```
{
  "height": 211.0,
  "left": 0.0,
  "name": "Chart 1",
  "top": 0.0,
  "type": "chart",
  "width": 355.0
}
```

### 29.6.2 /books

GET /books

Example response:

```
{
  "books": [
    {
      "app": 1104,
      "fullname": "Book1",
      "name": "Book1",
      "names": [],
      "selection": "Sheet2!$A$1",
      "sheets": [
        "Sheet1"
      ]
    },
    {
      "app": 1104,
      "fullname": "C:\\Users\\felix\\DEV\\xlwings\\scripts\\Book1.xlsx",
      "name": "Book1.xlsx",
      "names": [
        "Sheet1!myname1",
        "myname2"
      ],
      "selection": "Sheet2!$A$1",
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```

    "sheets": [
      "Sheet1",
      "Sheet2"
    ]
  },
  {
    "app": 1104,
    "fullname": "Book4",
    "name": "Book4",
    "names": [],
    "selection": "Sheet2!$A$1",
    "sheets": [
      "Sheet1"
    ]
  }
]
}

```

GET /books/<book\_name\_or\_ix>

Example response:

```

{
  "app": 1104,
  "fullname": "C:\\Users\\felix\\DEV\\xlwings\\scripts\\Book1.xlsx",
  "name": "Book1.xlsx",
  "names": [
    "Sheet1!myname1",
    "myname2"
  ],
  "selection": "Sheet2!$A$1",
  "sheets": [
    "Sheet1",
    "Sheet2"
  ]
}

```

GET /books/<book\_name\_or\_ix>/names

Example response:

```

{
  "names": [
    {
      "name": "Sheet1!myname1",
      "refers_to": "=Sheet1!$B$2:$C$3"
    },
  ],
}

```

(continues on next page)

(continued from previous page)

```
{
  "name": "myname2",
  "refers_to": "=Sheet1!$A$1"
}
]
```

GET /books/<book\_name\_or\_ix>/names/<name>

Example response:

```
{
  "name": "myname2",
  "refers_to": "=Sheet1!$A$1"
}
```

GET /books/<book\_name\_or\_ix>/names/<name>/range

Example response:

```
{
  "address": "$A$1",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 1,
  "current_region": "$A$1:$B$2",
  "formula": "=1+1.1",
  "formula_array": "=1+1,1",
  "height": 14.25,
  "last_cell": "$A$1",
  "left": 0.0,
  "name": "myname2",
  "number_format": "General",
  "row": 1,
  "row_height": 14.3,
  "shape": [
    1,
    1
  ],
  "size": 1,
  "top": 0.0,
  "value": 2.1,
  "width": 51.0
}
```

GET /books/<book\_name\_or\_ix>/sheets

Example response:

```
{
  "sheets": [
    {
      "charts": [
        "Chart 1"
      ],
      "name": "Sheet1",
      "names": [
        "Sheet1!myname1"
      ],
      "pictures": [
        "Picture 3"
      ],
      "shapes": [
        "Chart 1",
        "Picture 3"
      ],
      "used_range": "$A$1:$B$2"
    },
    {
      "charts": [],
      "name": "Sheet2",
      "names": [],
      "pictures": [],
      "shapes": [],
      "used_range": "$A$1"
    }
  ]
}
```

GET /books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>

Example response:

```
{
  "charts": [
    "Chart 1"
  ],
  "name": "Sheet1",
  "names": [
    "Sheet1!myname1"
  ],
  "pictures": [
    "Picture 3"
  ],
  "shapes": [
```

(continues on next page)

(continued from previous page)

```
"Chart 1",
"Picture 3"
],
"used_range": "$A$1:$B$2"
}
```

GET /books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/charts

Example response:

```
{
  "charts": [
    {
      "chart_type": "line",
      "height": 211.0,
      "left": 0.0,
      "name": "Chart 1",
      "top": 0.0,
      "width": 355.0
    }
  ]
}
```

GET /books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/charts/<chart\_name\_or\_ix>

Example response:

```
{
  "chart_type": "line",
  "height": 211.0,
  "left": 0.0,
  "name": "Chart 1",
  "top": 0.0,
  "width": 355.0
}
```

GET /books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/names

Example response:

```
{
  "names": [
    {
      "name": "Sheet1!myname1",
      "refers_to": "=Sheet1!$B$2:$C$3"
    }
  ]
}
```

GET /books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/names/<sheet\_scope\_name>

Example response:

```
{
  "name": "Sheet1!myname1",
  "refers_to": "=Sheet1!$B$2:$C$3"
}
```

GET /books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/names/<sheet\_scope\_name>/range

Example response:

```
{
  "address": "$B$2:$C$3",
  "color": null,
  "column": 2,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    [
      "",
      ""
    ],
    [
      "",
      ""
    ]
  ],
  "formula_array": "",
  "height": 28.5,
  "last_cell": "$C$3",
  "left": 51.0,
  "name": "Sheet1!myname1",
  "number_format": "General",
  "row": 2,
  "row_height": 14.3,
  "shape": [
    2,
    2
  ],
  "size": 4,
  "top": 14.25,
  "value": [
    [
      null,

```

(continues on next page)

(continued from previous page)

```
    null
  ],
  [
    null,
    null
  ]
],
"width": 102.0
}
```

GET /books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/pictures

Example response:

```
{
  "pictures": [
    {
      "height": 100.0,
      "left": 0.0,
      "name": "Picture 3",
      "top": 0.0,
      "width": 100.0
    }
  ]
}
```

GET /books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/pictures/  
<picture\_name\_or\_ix>

Example response:

```
{
  "height": 100.0,
  "left": 0.0,
  "name": "Picture 3",
  "top": 0.0,
  "width": 100.0
}
```

GET /books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/range

Example response:

```
{
  "address": "$A$1:$B$2",
  "color": null,
  "column": 1,
  "column_width": 8.47,
```

(continues on next page)

(continued from previous page)

```

"count": 4,
"current_region": "$A$1:$B$2",
"formula": [
  [
    "=1+1.1",
    "a string"
  ],
  [
    "43395.0064583333",
    ""
  ]
],
"formula_array": null,
"height": 28.5,
"last_cell": "$B$2",
"left": 0.0,
"name": null,
"number_format": null,
"row": 1,
"row_height": 14.3,
"shape": [
  2,
  2
],
"size": 4,
"top": 0.0,
"value": [
  [
    2.1,
    "a string"
  ],
  [
    "Mon, 22 Oct 2018 00:09:18 GMT",
    null
  ]
],
"width": 102.0
}

```

GET /books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/range/<address>

Example response:

```

{
  "address": "$A$1:$B$2",
  "color": null,

```

(continues on next page)

(continued from previous page)

```

"column": 1,
"column_width": 8.47,
"count": 4,
"current_region": "$A$1:$B$2",
"formula": [
  [
    "=1+1.1",
    "a string"
  ],
  [
    "43395.0064583333",
    ""
  ]
],
"formula_array": null,
"height": 28.5,
"last_cell": "$B$2",
"left": 0.0,
"name": null,
"number_format": null,
"row": 1,
"row_height": 14.3,
"shape": [
  2,
  2
],
"size": 4,
"top": 0.0,
"value": [
  [
    2.1,
    "a string"
  ],
  [
    "Mon, 22 Oct 2018 00:09:18 GMT",
    null
  ]
],
"width": 102.0
}

```

GET /books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/shapes

Example response:

```
{
```

(continues on next page)

(continued from previous page)

```

"shapes": [
  {
    "height": 211.0,
    "left": 0.0,
    "name": "Chart 1",
    "top": 0.0,
    "type": "chart",
    "width": 355.0
  },
  {
    "height": 100.0,
    "left": 0.0,
    "name": "Picture 3",
    "top": 0.0,
    "type": "picture",
    "width": 100.0
  }
]
}

```

GET /books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/shapes/<shape\_name\_or\_ix>

Example response:

```

{
  "height": 211.0,
  "left": 0.0,
  "name": "Chart 1",
  "top": 0.0,
  "type": "chart",
  "width": 355.0
}

```

### 29.6.3 /apps

GET /apps

Example response:

```

{
  "apps": [
    {
      "books": [
        "Book1",
        "C:\\Users\\felix\\DEV\\xlwings\\scripts\\Book1.xlsx",
        "Book4"
      ]
    }
  ]
}

```

(continues on next page)

(continued from previous page)

```
    ],
    "calculation": "automatic",
    "display_alerts": true,
    "pid": 1104,
    "screen_updating": true,
    "selection": "[Book1.xlsx]Sheet2!$A$1",
    "version": "16.0",
    "visible": true
  },
  {
    "books": [
      "Book2",
      "Book5"
    ],
    "calculation": "automatic",
    "display_alerts": true,
    "pid": 7920,
    "screen_updating": true,
    "selection": "[Book5]Sheet2!$A$1",
    "version": "16.0",
    "visible": true
  }
]
```

**GET /apps/<pid>****Example response:**

```
{
  "books": [
    "Book1",
    "C:\\Users\\felix\\DEV\\xlwings\\scripts\\Book1.xlsx",
    "Book4"
  ],
  "calculation": "automatic",
  "display_alerts": true,
  "pid": 1104,
  "screen_updating": true,
  "selection": "[Book1.xlsx]Sheet2!$A$1",
  "version": "16.0",
  "visible": true
}
```

**GET /apps/<pid>/books****Example response:**

```
{
  "books": [
    {
      "app": 1104,
      "fullname": "Book1",
      "name": "Book1",
      "names": [],
      "selection": "Sheet2!$A$1",
      "sheets": [
        "Sheet1"
      ]
    },
    {
      "app": 1104,
      "fullname": "C:\\Users\\felix\\DEV\\xlwings\\scripts\\Book1.xlsx",
      "name": "Book1.xlsx",
      "names": [
        "Sheet1!myname1",
        "myname2"
      ],
      "selection": "Sheet2!$A$1",
      "sheets": [
        "Sheet1",
        "Sheet2"
      ]
    },
    {
      "app": 1104,
      "fullname": "Book4",
      "name": "Book4",
      "names": [],
      "selection": "Sheet2!$A$1",
      "sheets": [
        "Sheet1"
      ]
    }
  ]
}
```

GET /apps/<pid>/books/<book\_name\_or\_ix>

Example response:

```
{
  "app": 1104,
  "fullname": "C:\\Users\\felix\\DEV\\xlwings\\scripts\\Book1.xlsx",
  "name": "Book1.xlsx",
```

(continues on next page)

(continued from previous page)

```
"names": [
  "Sheet1!myname1",
  "myname2"
],
"selection": "Sheet2!$A$1",
"sheets": [
  "Sheet1",
  "Sheet2"
]
}
```

GET /apps/<pid>/books/<book\_name\_or\_ix>/names

Example response:

```
{
  "names": [
    {
      "name": "Sheet1!myname1",
      "refers_to": "=Sheet1!$B$2:$C$3"
    },
    {
      "name": "myname2",
      "refers_to": "=Sheet1!$A$1"
    }
  ]
}
```

GET /apps/<pid>/books/<book\_name\_or\_ix>/names/<name>

Example response:

```
{
  "name": "myname2",
  "refers_to": "=Sheet1!$A$1"
}
```

GET /apps/<pid>/books/<book\_name\_or\_ix>/names/<name>/range

Example response:

```
{
  "address": "$A$1",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 1,
  "current_region": "$A$1:$B$2",

```

(continues on next page)

(continued from previous page)

```

"formula": "=1+1.1",
"formula_array": "=1+1,1",
"height": 14.25,
"last_cell": "$A$1",
"left": 0.0,
"name": "myname2",
"number_format": "General",
"row": 1,
"row_height": 14.3,
"shape": [
    1,
    1
],
"size": 1,
"top": 0.0,
"value": 2.1,
"width": 51.0
}

```

GET /apps/<pid>/books/<book\_name\_or\_ix>/sheets

Example response:

```

{
  "sheets": [
    {
      "charts": [
        "Chart 1"
      ],
      "name": "Sheet1",
      "names": [
        "Sheet1!myname1"
      ],
      "pictures": [
        "Picture 3"
      ],
      "shapes": [
        "Chart 1",
        "Picture 3"
      ],
      "used_range": "$A$1:$B$2"
    },
    {
      "charts": [],
      "name": "Sheet2",
      "names": [],

```

(continues on next page)

(continued from previous page)

```
    "pictures": [],
    "shapes": [],
    "used_range": "$A$1"
  }
]
```

**GET** /apps/<pid>/books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>

**Example response:**

```
{
  "charts": [
    "Chart 1"
  ],
  "name": "Sheet1",
  "names": [
    "Sheet1!myname1"
  ],
  "pictures": [
    "Picture 3"
  ],
  "shapes": [
    "Chart 1",
    "Picture 3"
  ],
  "used_range": "$A$1:$B$2"
}
```

**GET** /apps/<pid>/books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/charts

**Example response:**

```
{
  "charts": [
    {
      "chart_type": "line",
      "height": 211.0,
      "left": 0.0,
      "name": "Chart 1",
      "top": 0.0,
      "width": 355.0
    }
  ]
}
```

**GET** /apps/<pid>/books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/charts/  
<chart\_name\_or\_ix>

Example response:

```
{
  "chart_type": "line",
  "height": 211.0,
  "left": 0.0,
  "name": "Chart 1",
  "top": 0.0,
  "width": 355.0
}
```

GET /apps/<pid>/books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/names

Example response:

```
{
  "names": [
    {
      "name": "Sheet1!myname1",
      "refers_to": "=Sheet1!$B$2:$C$3"
    }
  ]
}
```

GET /apps/<pid>/books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/names/  
<sheet\_scope\_name>

Example response:

```
{
  "name": "Sheet1!myname1",
  "refers_to": "=Sheet1!$B$2:$C$3"
}
```

GET /apps/<pid>/books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/names/  
<sheet\_scope\_name>/range

Example response:

```
{
  "address": "$B$2:$C$3",
  "color": null,
  "column": 2,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    [
      "",

```

(continues on next page)

(continued from previous page)

```

        ""
    ],
    [
        "",
        ""
    ]
],
"formula_array": "",
"height": 28.5,
"last_cell": "$C$3",
"left": 51.0,
"name": "Sheet1!myname1",
"number_format": "General",
"row": 2,
"row_height": 14.3,
"shape": [
    2,
    2
],
"size": 4,
"top": 14.25,
"value": [
    [
        null,
        null
    ],
    [
        null,
        null
    ]
],
"width": 102.0
}

```

GET /apps/<pid>/books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/pictures

Example response:

```

{
  "pictures": [
    {
      "height": 100.0,
      "left": 0.0,
      "name": "Picture 3",
      "top": 0.0,
      "width": 100.0
    }
  ]
}

```

(continues on next page)

(continued from previous page)

```

    }
  ]
}

```

GET /apps/<pid>/books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/pictures/  
<picture\_name\_or\_ix>

Example response:

```

{
  "height": 100.0,
  "left": 0.0,
  "name": "Picture 3",
  "top": 0.0,
  "width": 100.0
}

```

GET /apps/<pid>/books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/range

Example response:

```

{
  "address": "$A$1:$B$2",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    [
      "=1+1.1",
      "a string"
    ],
    [
      "43395.0064583333",
      ""
    ]
  ],
  "formula_array": null,
  "height": 28.5,
  "last_cell": "$B$2",
  "left": 0.0,
  "name": null,
  "number_format": null,
  "row": 1,
  "row_height": 14.3,
  "shape": [

```

(continues on next page)

(continued from previous page)

```

    2,
    2
  ],
  "size": 4,
  "top": 0.0,
  "value": [
    [
      2.1,
      "a string"
    ],
    [
      "Mon, 22 Oct 2018 00:09:18 GMT",
      null
    ]
  ],
  "width": 102.0
}

```

GET /apps/<pid>/books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/range/<address>

Example response:

```

{
  "address": "$A$1:$B$2",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    [
      "=1+1.1",
      "a string"
    ],
    [
      "43395.0064583333",
      ""
    ]
  ],
  "formula_array": null,
  "height": 28.5,
  "last_cell": "$B$2",
  "left": 0.0,
  "name": null,
  "number_format": null,
  "row": 1,

```

(continues on next page)

(continued from previous page)

```

"row_height": 14.3,
"shape": [
  2,
  2
],
"size": 4,
"top": 0.0,
"value": [
  [
    2.1,
    "a string"
  ],
  [
    "Mon, 22 Oct 2018 00:09:18 GMT",
    null
  ]
],
"width": 102.0
}

```

GET /apps/<pid>/books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/shapes

Example response:

```

{
  "shapes": [
    {
      "height": 211.0,
      "left": 0.0,
      "name": "Chart 1",
      "top": 0.0,
      "type": "chart",
      "width": 355.0
    },
    {
      "height": 100.0,
      "left": 0.0,
      "name": "Picture 3",
      "top": 0.0,
      "type": "picture",
      "width": 100.0
    }
  ]
}

```

GET /apps/<pid>/books/<book\_name\_or\_ix>/sheets/<sheet\_name\_or\_ix>/shapes/  
<shape\_name\_or\_ix>

Example response:

```
{
  "height": 211.0,
  "left": 0.0,
  "name": "Chart 1",
  "top": 0.0,
  "type": "chart",
  "width": 355.0
}
```

## A

activate() (*xlwings.App* method), 134  
 activate() (*xlwings.Book* method), 140  
 activate() (*xlwings.Shape* method), 161  
 activate() (*xlwings.Sheet* method), 145  
 active (*xlwings.main.Apps* property), 132  
 active (*xlwings.main.Books* property), 138  
 active (*xlwings.main.Sheets* property), 144  
 add() (*xlwings.main.Apps* method), 132  
 add() (*xlwings.main.Books* method), 138  
 add() (*xlwings.main.Charts* method), 163  
 add() (*xlwings.main.Names* method), 168  
 add() (*xlwings.main.Pictures* method), 165  
 add() (*xlwings.main.Sheets* method), 145  
 add() (*xlwings.main.Tables* method), 170  
 add\_hyperlink() (*xlwings.Range* method), 150  
 address (*xlwings.Range* property), 150  
 api (*xlwings.App* property), 134  
 api (*xlwings.Book* property), 140  
 api (*xlwings.Chart* property), 164  
 api (*xlwings.main.Characters* property), 174  
 api (*xlwings.main.Charts* property), 163  
 api (*xlwings.main.Font* property), 173  
 api (*xlwings.main.Names* property), 168  
 api (*xlwings.main.Note* property), 169  
 api (*xlwings.main.PageSetup* property), 144  
 api (*xlwings.main.Pictures* property), 166  
 api (*xlwings.main.Shapes* property), 161  
 api (*xlwings.main.Table* property), 171  
 api (*xlwings.Name* property), 169  
 api (*xlwings.Picture* property), 167  
 api (*xlwings.Range* property), 150  
 api (*xlwings.Shape* property), 161  
 api (*xlwings.Sheet* property), 145  
 App (class in *xlwings*), 133  
 app (*xlwings.Book* property), 140  
 Apps (class in *xlwings.main*), 132

autofit() (*xlwings.Range* method), 150  
 autofit() (*xlwings.RangeColumns* method), 160  
 autofit() (*xlwings.RangeRows* method), 160  
 autofit() (*xlwings.Sheet* method), 145

## B

bold (*xlwings.main.Font* property), 173  
 Book (class in *xlwings*), 139  
 book (*xlwings.Sheet* property), 146  
 Books (class in *xlwings.main*), 138  
 books (*xlwings.App* property), 134

## C

calculate() (*xlwings.App* method), 134  
 calculation (*xlwings.App* property), 134  
 caller() (*xlwings.Book* class method), 140  
 cells (*xlwings.Sheet* property), 146  
 Characters (class in *xlwings.main*), 174  
 Chart (class in *xlwings*), 164  
 chart\_type (*xlwings.Chart* property), 164  
 Charts (class in *xlwings.main*), 163  
 charts (*xlwings.Sheet* property), 146  
 clear() (*xlwings.Range* method), 150  
 clear() (*xlwings.Sheet* method), 146  
 clear\_contents() (*xlwings.Range* method), 150  
 clear\_contents() (*xlwings.Sheet* method), 146  
 close() (*xlwings.Book* method), 140  
 color (*xlwings.main.Font* property), 173  
 color (*xlwings.Range* property), 150  
 column (*xlwings.Range* property), 151  
 column\_width (*xlwings.Range* property), 151  
 columns (*xlwings.Range* property), 151  
 copy() (*xlwings.Range* method), 151  
 copy() (*xlwings.Sheet* method), 146  
 copy\_picture() (*xlwings.Range* method), 151  
 count (*xlwings.main.Apps* property), 132  
 count (*xlwings.main.Charts* property), 163  
 count (*xlwings.main.Names* property), 168

count (*xlwings.main.Pictures* property), 167  
 count (*xlwings.main.Shapes* property), 161  
 count (*xlwings.Range* property), 152  
 count (*xlwings.RangeColumns* property), 160  
 count (*xlwings.RangeRows* property), 160  
 current\_region (*xlwings.Range* property), 152  
 cut\_copy\_mode (*xlwings.App* property), 134

## D

data\_body\_range (*xlwings.main.Table* property), 171  
 delete() (*xlwings.Chart* method), 164  
 delete() (*xlwings.main.Note* method), 169  
 delete() (*xlwings.Name* method), 169  
 delete() (*xlwings.Picture* method), 167  
 delete() (*xlwings.Range* method), 152  
 delete() (*xlwings.Shape* method), 161  
 delete() (*xlwings.Sheet* method), 147  
 display\_alerts (*xlwings.App* property), 134  
 display\_name (*xlwings.main.Table* property), 171

## E

enable\_events (*xlwings.App* property), 134  
 end() (*xlwings.Range* method), 152  
 expand() (*xlwings.Range* method), 152

## F

Font (class in *xlwings.main*), 173  
 font (*xlwings.main.Characters* property), 175  
 formula (*xlwings.Range* property), 153  
 formula2 (*xlwings.Range* property), 153  
 formula\_array (*xlwings.Range* property), 153  
 fullname (*xlwings.Book* property), 140

## G

get\_address() (*xlwings.Range* method), 153

## H

has\_array (*xlwings.Range* property), 154  
 header\_row\_range (*xlwings.main.Table* property), 171  
 height (*xlwings.Chart* property), 164  
 height (*xlwings.Picture* property), 167  
 height (*xlwings.Range* property), 154  
 height (*xlwings.Shape* property), 161  
 hwnd (*xlwings.App* property), 134  
 hyperlink (*xlwings.Range* property), 154

## I

Image (class in *xlwings.pro.reports*), 178  
 index (*xlwings.Sheet* property), 147  
 insert() (*xlwings.Range* method), 154  
 insert\_row\_range (*xlwings.main.Table* property), 171  
 interactive (*xlwings.App* property), 135  
 italic (*xlwings.main.Font* property), 174

## K

keys() (*xlwings.main.Apps* method), 133  
 kill() (*xlwings.App* method), 135

## L

last\_cell (*xlwings.Range* property), 155  
 left (*xlwings.Chart* property), 164  
 left (*xlwings.Picture* property), 167  
 left (*xlwings.Range* property), 155  
 left (*xlwings.Shape* property), 161  
 load() (in module *xlwings*), 131  
 lock\_aspect\_ratio (*xlwings.Picture* property), 167

## M

macro() (*xlwings.App* method), 135  
 macro() (*xlwings.Book* method), 141  
 Markdown (class in *xlwings.pro*), 175  
 Markdown (class in *xlwings.pro.reports*), 178  
 MarkdownStyle (class in *xlwings.pro*), 176  
 MarkdownStyle (class in *xlwings.pro.reports*), 179  
 merge() (*xlwings.Range* method), 155  
 merge\_area (*xlwings.Range* property), 155  
 merge\_cells (*xlwings.Range* property), 155  
 module  
     *xlwings*, 131  
     *xlwings.pro.reports*, 178

## N

Name (class in *xlwings*), 169  
 name (*xlwings.Book* property), 141  
 name (*xlwings.Chart* property), 164  
 name (*xlwings.main.Font* property), 174  
 name (*xlwings.main.Table* property), 171  
 name (*xlwings.Name* property), 169  
 name (*xlwings.Picture* property), 167  
 name (*xlwings.Range* property), 155  
 name (*xlwings.Shape* property), 162  
 name (*xlwings.Sheet* property), 147

Names (*class in xlwings.main*), 168  
 names (*xlwings.Book property*), 141  
 names (*xlwings.Sheet property*), 147  
 Note (*class in xlwings.main*), 169  
 note (*xlwings.Range property*), 155  
 number\_format (*xlwings.Range property*), 155

## O

offset() (*xlwings.Range method*), 156  
 open() (*xlwings.main.Books method*), 138  
 options() (*xlwings.Range method*), 156

## P

page\_setup (*xlwings.Sheet property*), 147  
 PageSetup (*class in xlwings.main*), 144  
 parent (*xlwings.Chart property*), 164  
 parent (*xlwings.main.Table property*), 171  
 parent (*xlwings.Picture property*), 167  
 parent (*xlwings.Shape property*), 162  
 paste() (*xlwings.Range method*), 157  
 Picture (*class in xlwings*), 167  
 Pictures (*class in xlwings.main*), 165  
 pictures (*xlwings.Sheet property*), 147  
 pid (*xlwings.App property*), 136  
 print\_area (*xlwings.main.PageSetup property*), 144  
 properties() (*xlwings.App method*), 136

## Q

quit() (*xlwings.App method*), 136

## R

Range (*class in xlwings*), 149  
 range (*xlwings.main.Table property*), 171  
 range() (*xlwings.App method*), 136  
 range() (*xlwings.Sheet method*), 147  
 RangeColumns (*class in xlwings*), 160  
 RangeRows (*class in xlwings*), 159  
 raw\_value (*xlwings.Range property*), 157  
 refers\_to (*xlwings.Name property*), 169  
 refers\_to\_range (*xlwings.Name property*), 169  
 render\_template() (*in module xlwings.pro.reports*), 179  
 render\_template() (*xlwings.App method*), 136  
 render\_template() (*xlwings.Book method*), 141  
 render\_template() (*xlwings.Sheet method*), 147  
 resize() (*xlwings.main.Table method*), 171  
 resize() (*xlwings.Range method*), 157

row (*xlwings.Range property*), 157  
 row\_height (*xlwings.Range property*), 157  
 rows (*xlwings.Range property*), 158

## S

save() (*xlwings.Book method*), 142  
 scale\_height() (*xlwings.Shape method*), 162  
 scale\_width() (*xlwings.Shape method*), 162  
 screen\_updating (*xlwings.App property*), 137  
 select() (*xlwings.Range method*), 158  
 select() (*xlwings.Sheet method*), 148  
 selection (*xlwings.App property*), 137  
 selection (*xlwings.Book property*), 142  
 set\_mock\_caller() (*xlwings.Book method*), 142  
 set\_source\_data() (*xlwings.Chart method*), 165  
 Shape (*class in xlwings*), 161  
 shape (*xlwings.Range property*), 158  
 Shapes (*class in xlwings.main*), 161  
 shapes (*xlwings.Sheet property*), 148  
 Sheet (*class in xlwings*), 145  
 sheet (*xlwings.Range property*), 158  
 Sheets (*class in xlwings.main*), 144  
 sheets (*xlwings.Book property*), 143  
 show\_autofilter (*xlwings.main.Table property*), 172  
 show\_headers (*xlwings.main.Table property*), 172  
 show\_table\_style\_column\_stripes (*xlwings.main.Table property*), 172  
 show\_table\_style\_first\_column (*xlwings.main.Table property*), 172  
 show\_table\_style\_last\_column (*xlwings.main.Table property*), 172  
 show\_table\_style\_row\_stripes (*xlwings.main.Table property*), 172  
 show\_totals (*xlwings.main.Table property*), 172  
 size (*xlwings.main.Font property*), 174  
 size (*xlwings.Range property*), 158  
 startup\_path (*xlwings.App property*), 137  
 status\_bar (*xlwings.App property*), 137

## T

Table (*class in xlwings.main*), 171  
 table (*xlwings.Range property*), 158  
 table\_style (*xlwings.main.Table property*), 172  
 Tables (*class in xlwings.main*), 170  
 tables (*xlwings.Sheet property*), 148  
 text (*xlwings.main.Characters property*), 175  
 text (*xlwings.main.Note property*), 169

`text` (*xlwings.Shape* property), 162  
`to_pdf()` (*xlwings.Book* method), 143  
`to_pdf()` (*xlwings.Sheet* method), 148  
`to_png()` (*xlwings.Chart* method), 165  
`to_png()` (*xlwings.Range* method), 158  
`top` (*xlwings.Chart* property), 165  
`top` (*xlwings.Picture* property), 167  
`top` (*xlwings.Range* property), 158  
`top` (*xlwings.Shape* property), 162  
`totals_row_range` (*xlwings.main.Table* property),  
172  
`type` (*xlwings.Shape* property), 162

## U

`unmerge()` (*xlwings.Range* method), 159  
`update()` (*xlwings.main.Table* method), 172  
`update()` (*xlwings.Picture* method), 168  
`used_range` (*xlwings.Sheet* property), 149

## V

`value` (*xlwings.Range* property), 159  
`version` (*xlwings.App* property), 137  
`view()` (in module *xlwings*), 131  
`visible` (*xlwings.App* property), 138  
`visible` (*xlwings.Sheet* property), 149

## W

`width` (*xlwings.Chart* property), 165  
`width` (*xlwings.Picture* property), 168  
`width` (*xlwings.Range* property), 159  
`width` (*xlwings.Shape* property), 162  
`wrap_text` (*xlwings.Range* property), 159

## X

`xlwings`  
    module, 131  
`xlwings.arg()` (in module *xlwings*), 177  
`xlwings.func()` (in module *xlwings*), 176  
`xlwings.pro.reports`  
    module, 178  
`xlwings.ret()` (in module *xlwings*), 177  
`xlwings.sub()` (in module *xlwings*), 177