xlwings - Make Excel Fly!

Release dev

Zoomer Analytics LLC

Apr 19, 2021
# Getting Started

## 1 Video course

## 2 Installation
- 2.1 Prerequisites
- 2.2 Installation
- 2.3 Add-in
- 2.4 Dependencies
- 2.5 How to activate xlwings PRO
- 2.6 Optional Dependencies
- 2.7 Update
- 2.8 Uninstall

## 3 Quickstart
- 3.1 1. Interacting with Excel from a Jupyter notebook
- 3.2 2. Scripting: Automate/interact with Excel from Python
- 3.3 3. Macros: Call Python from Excel
- 3.4 4. UDFs: User Defined Functions (Windows only)

## 4 Connect to a Book
- 4.1 Python to Excel
- 4.2 Excel to Python (RunPython)
- 4.3 User Defined Functions (UDFs)

## 5 Syntax Overview
- 5.1 Active Objects
- 5.2 Full qualification
- 5.3 Range indexing/slicing
- 5.4 Range Shortcuts
- 5.5 Object Hierarchy

## 6 Data Structures Tutorial
- 6.1 Single Cells
- 6.2 Lists
6.3 Range expanding .................................................. 19
6.4 NumPy arrays .................................................... 19
6.5 Pandas DataFrames .............................................. 19
6.6 Pandas Series .................................................... 20
6.7 Chunking: Read/Write big DataFrames etc. ............... 20

7 Add-in & Settings .................................................. 23
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

8 RunPython ........................................................ 29
8.1 xlwings add-in .................................................... 29
8.2 Call Python with “RunPython” ............................... 29
8.3 Function Arguments and Return Values .................... 30

9 User Defined Functions (UDFs) ............................... 31
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 ......................................................... 36
9.9 The “caller” argument ......................................... 36
9.10 The “vba” keyword ............................................ 37
9.11 Macros .......................................................... 37
9.12 Call UDFs from VBA ........................................... 37
9.13 Asynchronous UDFs .......................................... 38

10 Matplotlib & Plotly Charts ................................. 39
10.1 Matplotlib ......................................................... 39
10.2 Plotly static charts ............................................. 42

11 Jupyter Notebooks: Interact with Excel .................. 45
11.1 The view function .............................................. 45
11.2 The load function .............................................. 45

12 Command Line Client (CLI) ................................. 47

13 xlwings Reports ................................................. 49
13.1 Quickstart ...................................................... 49
13.2 Frames ......................................................... 52
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>xlwings with other Office Apps</td>
<td>107</td>
</tr>
<tr>
<td>24.1</td>
<td>How To</td>
<td>107</td>
</tr>
<tr>
<td>24.2</td>
<td>Config</td>
<td>108</td>
</tr>
<tr>
<td>25</td>
<td>xlwings with R and Julia</td>
<td>109</td>
</tr>
<tr>
<td>25.1</td>
<td>R</td>
<td>109</td>
</tr>
<tr>
<td>25.2</td>
<td>Julia</td>
<td>110</td>
</tr>
<tr>
<td>26</td>
<td>Python API</td>
<td>113</td>
</tr>
<tr>
<td>26.1</td>
<td>Top-level functions</td>
<td>113</td>
</tr>
<tr>
<td>26.2</td>
<td>Object model</td>
<td>114</td>
</tr>
<tr>
<td>26.3</td>
<td>UDF decorators</td>
<td>152</td>
</tr>
<tr>
<td>26.4</td>
<td>Reports</td>
<td>154</td>
</tr>
<tr>
<td>27</td>
<td>REST API</td>
<td>157</td>
</tr>
<tr>
<td>27.1</td>
<td>Quickstart</td>
<td>157</td>
</tr>
<tr>
<td>27.2</td>
<td>Run the server</td>
<td>159</td>
</tr>
<tr>
<td>27.3</td>
<td>Indexing</td>
<td>159</td>
</tr>
<tr>
<td>27.4</td>
<td>Range Options</td>
<td>159</td>
</tr>
<tr>
<td>27.5</td>
<td>Endpoint overview</td>
<td>159</td>
</tr>
<tr>
<td>27.6</td>
<td>Endpoint details</td>
<td>160</td>
</tr>
<tr>
<td>Index</td>
<td></td>
<td>187</td>
</tr>
</tbody>
</table>
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 :)
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 add-in 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, appscript

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` 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)

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.
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:

```python
>>> 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()`:

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

Instantiate a sheet object:

```python
>>> sht = wb.sheets['Sheet1']
```
Reading/writing values to/from ranges is as easy as:

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

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

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

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

```python
>>> import pandas as pd

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

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

```python
>>> import matplotlib.pyplot as plt

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

### 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`:

```vba
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()`:

```python
# 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:

```python
import xlwings as xw

@xw.func
def hello(name):
    return 'Hello {0}'.format(name)
```

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

```python
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)*.
When reading/writing data to the active sheet, you don’t need a book object:

```python
>>> 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:

```python
>>> app = xw.App()  # or something like xw.apps[10559] for existing apps, get
→the available PIDs via xw.apps.keys()
>>> app.books['Book1']
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsaved book</td>
<td><code>xw.Book('Book1')</code></td>
<td><code>xw.books['Book1']</code></td>
</tr>
<tr>
<td>Book by (full)name</td>
<td><code>xw.Book(r'C:/path/to/file.xlsx')</code></td>
<td><code>xw.books.open(r'C:/path/to/file.xlsx')</code></td>
</tr>
</tbody>
</table>

**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.
The xlwings object model is very similar to the one used by VBA.
All code samples below depend on the following import:

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

### 5.1 Active Objects

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

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

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

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

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

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

(continues on next page)
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 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>
```

5.4 Range Shortcuts

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

```
>>> sht = xw.Book().sheets['Sheet1']
>>> sht['A1']
<Range [Book1]Sheet1!$A$1>
>>> sht['A1:B5']
>>> sht[0, 1]
<Range [Book1]Sheet1!$B$1>
>>> sht[:, :10]
<Range [Book1]Sheet1!$A$1:$J$10>
```
5.5 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:

```python
>>> rng = xw.apps[10559].books[0].sheets[0].range('A1')
>>> rng.sheet.book.app
<Excel App 10559>
```
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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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”):

```python
>>> 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:

```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:

```python
>>> sheet = xw.Book().worksheets[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:

```python
>>> import numpy as np

>>> sheet = xw.Book().worksheets[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

```python
>>> sheet = xw.Book().worksheets[0]

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

>>> df
```
>>> 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
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 >= 2007 on Windows and >= 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.

![References - VBAProject](image)

### 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!
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.

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.
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`. 
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.

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

This calls the following code in hello.py:

```python
# hello.py
import numpy as np
import xlwings as xw

def world():
```

(continues on next page)
You can then attach `HelloWorld` to a button or run it directly in the VBA Editor by hitting F5.

**Note:** Place `xlw.Book.caller()` within the function that is being called from Excel and not outside as a 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

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:

```python
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:

![Image of Excel spreadsheet with formula 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:

```python
@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:

![Screenshot of Excel array formula](image)

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:

```python
@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:

```python
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:

```python
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:

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

For further details see the Converters and Options documentation.
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:

```python
import numpy as np

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

As shown in the example, the `=dynamic_array(B2,C2)` formula is entered in cell B4, but instead of being coerced to a single number, the results appear as a table in cells B5:C9, reflecting the dimensions specified. Note that 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.

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:

```python
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:

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

```python
@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:

```python
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:

```vba
Sub MySub()
    Dim arr() As Variant
```

(continues on next page)
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:

```python
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.
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:

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

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

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

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

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:

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

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

or:
xlwings - Make Excel Fly!, Release dev

10.1. Matplotlib 41
10.1.4 Getting a Matplotlib figure

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

- via PyPlot interface:

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

or:

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

- via object oriented interface:

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

- via Pandas:

```python
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])
al = df.plot(kind='bar')
fig = al.get_figure()
```

10.2 Plotly static charts

This feature requires xlwings PRO.

10.2.1 Prerequisites

In addition to plotly you will need orca. The easiest way to get it is via conda:

```
$ conda install -c plotly plotly-orca psutil requests
```

For alternative ways of installation, see: 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:

```python
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`.

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`.

New in version 0.22.0.
Chapter 11. Jupyter Notebooks: Interact with Excel
xlwings comes with a command line client. On Windows, type the commands into a Command Prompt, on Mac, type them into a Terminal. To get an overview of all commands, simply type `xlwings` and hit Enter:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addin</td>
<td>Run &quot;xlwings addin install&quot; to install the Excel add-in (will be copied to the XLSTART folder). Instead of &quot;install&quot; you can also use &quot;update&quot;, &quot;remove&quot; or &quot;status&quot;. Note that this command may take a while. Use the &quot;--unprotected&quot; flag to install the add-in without password protection. You can install your custom add-in by providing the name or path via the --file flag, e.g. &quot;xlwings add-in install --file custom.xlam&quot; (New in 0.6.0, the unprotected flag was added in 0.20.4)</td>
</tr>
<tr>
<td>quickstart</td>
<td>Run &quot;xlwings quickstart myproject&quot; to create a folder called &quot;myproject&quot; in the current directory with an Excel file and a Python file, ready to be used. Use the &quot;--standalone&quot; flag to embed all VBA code in the Excel file and make it work without the xlwings add-in.</td>
</tr>
<tr>
<td>runpython</td>
<td>macOS only: run &quot;xlwings runpython install&quot; 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)</td>
</tr>
<tr>
<td>restapi</td>
<td>Use &quot;xlwings restapi run&quot; to run the xlwings REST API via Flask dev server. Accepts &quot;--host&quot; and &quot;--port&quot; as optional arguments.</td>
</tr>
<tr>
<td>license</td>
<td>xlwings PRO: Use &quot;xlwings license update -k KEY&quot; where &quot;KEY&quot; is your personal (trial) license key. This will update ~/.xlwings/xlwings.conf with the LICENSE_KEY</td>
</tr>
</tbody>
</table>

(continues on next page)
entry. If you have a paid license, you can run
"xlwings license deploy" to create a deploy key. This
is not available for trial keys.

config
Run "xlwings config create" to create the user config
file (~/.xlwings/xlwings.conf) 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 "--force" flag which will overwrite your
current configuration.
(New in 0.19.5)

code
Run "xlwings code embed" to embed all Python modules
of the current dir in your active Excel file. Use the
"--file" flag to only import a single file by
providing its path. To run embedded code, you need an
xlwings PRO license.
(New in 0.20.2)
This feature requires xlwings PRO.

13.1 Quickstart

xlwings Reports is part of xlwings PRO and a solution for template-based Excel and PDF reporting. It allows business users without Python knowledge to create & maintain Excel templates without having to go back to a Python developer for every change: xlwings Reports separates the Python code (that gets and prepares all the data) from the Excel template (that defines which data goes where and how it should be formatted). See also the xlwings Reports homepage. You can render one sheet at the time via `mysheet.render_template` or use the higher-level convenience function `xw.create_report` which first copies the template workbook and then loops through all sheets.

13.1.1 Render Sheets

Let’s first look at how to render a single sheet. This is a workbook stored as `Book1.xlsx`:

Running the following code:

```python
import xlwings as xw
wb = xw.Book('Book1.xlsx')
sheet = wb.sheets['template'].copy(name='report')
sheet.render_template(title='A Demo!', table=[[1, 2], [3, 4]])
wb.to_pdf()  # requires xlwings >=0.21.1
```

Leaves you with this:

See also the API reference.
## Chapter 13. xlwings Reports

### Table 1: Title Table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>{{ title }}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 2: Data Table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Demo!</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

---

*xlwings - Make Excel Fly!, Release dev*
New in version 0.22.0.

13.1.2 Render Workbooks

If your template is a full workbook, you can use the `create_report` function. Start by creating the following Python script `my_template.py`:

```python
from xlwings.pro.reports import create_report
import pandas as pd

df = pd.DataFrame(data=[[1,2],[3,4]])
w = create_report('my_template.xlsx', 'my_report.xlsx', title='MyTitle',
                  df=df)
w.to_pdf()  # requires xlwings >=0.21.1
```

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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{{ title }}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><strong>My DataFrame</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>{{ df }}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now run the Python script:

```
python my_template.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:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MyTitle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><strong>My DataFrame</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

The last line (`wb.to_pdf()`) will print the workbook as PDF, for more details on the options, see `Book.to_pdf()`.

Apart from Strings and Pandas DataFrames, you can also use numbers, lists, simple dicts, NumPy arrays, Matplotlib figures and PIL Image objects that have a filename.
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.

See also the API reference.

### 13.2 Frames

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 `<frame>` into row 1 of your Excel template wherever you want a new dynamic column to start. Row 1 will be removed automatically when creating the report. 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.

You can define and format table-like objects by formatting exactly

- one header and
- one data row

as shown in the screenshot:

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;frame&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;frame&gt;</td>
</tr>
<tr>
<td>2</td>
<td>Table 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Table 3</td>
</tr>
<tr>
<td>3</td>
<td>{{df1}}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>{{df2}}</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Table 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Table 4</td>
</tr>
<tr>
<td>7</td>
<td>{{df2}}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>{{df1}}</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

However, also make sure to check out how to use Excel Tables below, as they make the formatting easier.

Running the following code:

```python
from xlwings.pro.reports import create_report
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, df2=df2)
```

(continues on next page)
create_report('my_template.xlsx',
'my_report.xlsx',
**data)

will generate this report:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Table 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Table 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Table 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

13.3 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. On top of that, they are the easiest way to make the source of a chart dynamic. 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:

Running the following script:

```python
from xlwings.pro.reports import create_report
import pandas as pd

nrows, ncols = 3, 3
df = pd.DataFrame(data=nrows * [ncols * ['test']], columns=['col ' + str(i) for i in range(ncols)])
create_report('template.xlsx', 'output.xlsx', df=df.set_index('col 0'))
```

Will produce the following report:

**Note:**

- If you would like to exclude the DataFrame index, make sure to set the index to the first column e.g.: df.set_index('column_name').
### Chapter 13. xlwings Reports

#### Excel Sheet 1

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;frame&gt;</td>
</tr>
<tr>
<td>2</td>
<td>Title</td>
</tr>
<tr>
<td>4</td>
<td>{{ df }}</td>
</tr>
<tr>
<td>7</td>
<td>Some static text.</td>
</tr>
</tbody>
</table>

#### Excel Sheet 2

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Title</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>col 0</td>
<td>col 1</td>
</tr>
<tr>
<td>4</td>
<td>test</td>
<td>test</td>
</tr>
<tr>
<td>5</td>
<td>test</td>
<td>test</td>
</tr>
<tr>
<td>6</td>
<td>test</td>
<td>test</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Some static text.</td>
<td></td>
</tr>
</tbody>
</table>
• At the moment, you can only assign pandas DataFrames to tables.
• For Excel table support, you need at least version 0.21.0 and the index behavior was changed in 0.21.3

13.4 Excel Charts

Note: To use charts with a dynamic source, you’ll need at least xlwings version 0.22.1

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

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

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.

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'])

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

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 might want to 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.

### 13.5 Shape Text

New in version 0.21.4.

You can also use Shapes like Text Boxes or Rectangles with template text:

```
from xlwings.pro.reports import create_report

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

This code turns this template:

into this report:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Column 1</td>
<td>Q1</td>
<td>Q2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>North</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>South</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Bar chart showing comparison between North and South]
## Chapter 13. xlwings Reports

The image shows a section of an Excel spreadsheet with a chart and some placeholder text. The chart is a simple bar chart with two columns, labeled "Column2" and "Dummy." The placeholder text "{{ chart_data }}" suggests that this is a template for a report, where actual data will be inserted later.
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Column1</strong></td>
<td><strong>Q1</strong></td>
<td><strong>Q2</strong></td>
<td><strong>Q3</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>North</td>
<td>1000</td>
<td>4000</td>
<td>7000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>South</td>
<td>2000</td>
<td>5000</td>
<td>8000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>West</td>
<td>3000</td>
<td>6000</td>
<td>9000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Chart:**

- **X-axis:** North, South, West
- **Y-axis:** Values from 0 to 10000
- **Legend:**
  - Q1
  - Q2
  - Q3

---

13.5. Shape Text
While this works for simple text, you will lose the formatting if you have any. To prevent that, use a Markdown object, see below.

### 13.6 Markdown

New in version 0.23.0.

You can format text in cells or shapes via Markdown syntax:

```python
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.
"""

# The first sheet requires a shape as shown on the screenshot
sheet = xw.Book("MyTemplate.xlsx").sheets[0]
sheet.render_template(mypageholder=Markdown(mytext, style))
```

The temperature today is {{ temperature }} degrees celsius.

The temperature today is 12.3 degrees celsius.
This will render this template with the placeholder in a cell and a shape:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>{{ myplaceholder }}</td>
<td>{{ myplaceholder }}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Like this (this uses the default formatting):

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Title</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Text <strong>bold</strong> and <em>italic</em></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• A first bullet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• A second bullet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Another Title</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>This paragraph has a line break. Another line.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Title</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Text <strong>bold</strong> and <em>italic</em></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• A first bullet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• A second bullet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Another Title</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>This paragraph has a line break. Another line.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For more on Markdown, especially how to change the styling, see 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!

```python
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]

# 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:

![Excel Sheet with Markdown Formatting](image)

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:

```python
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.

(continues on next page)```
sheet = xw.Book("Book1.xlsx").sheets[0]

# Styling
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!

# 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:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Title</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Text <strong>bold</strong> and <em>italic</em></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>❤️ A first bullet</td>
<td>❤️ A second bullet</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><strong>Another Title</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>This paragraph has a line break.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Another line.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

```python
>>> 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`
15.1 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"
```

15.2 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:

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

### 15.3 Embedded Code

This feature requires xlwings PRO.

xlwings PRO allows you to store your Python code directly in Excel so you don’t have to distribute separate Python files.

On a command line, run the following command which will import all Python files from the current directory and paste them into sheets with the same name of the currently active workbook:

```
$ xlwings code embed
```

Then, use the VBA function `RunPython ("import mymodule;mymodule.myfunction()")` as usual.

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

```python
import xlwings as xw
import utils

def main():
    wb = xw.Book.caller()
    wb.sheets[0]["A1"].value = utils.get_random_number()
    wb.sheets[0]["A2"].value = utils.get_random_number()
```

![Excel Sheet with Embedded Code Example](image_url)
You can call this 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.

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

### 15.4 One-Click Zero-Config Installer

This feature requires xlwings PRO.

With xlwings PRO you get access to a private GitHub repository that will build your custom installer in the cloud — no local installation required. Using a custom installer to deploy the Python runtime has the following advantages:
• Zero Python knowledge required from end users
• Zero configuration required by end users
• No admin rights required
• Works for both UDFs and RunPython
• Works for external distribution
• Easy to deploy updates

15.4.1 End User Instructions

• Installing

Give the end user your Excel workbook and the installer. The user only has to double-click the installer and confirm a few prompts — no configuration is required.

• Updating

If you use the embedded code feature (see: Embedded Code), you can deploy updates by simply giving the user a new Excel file. Only when you change a dependency, you will need to create a new installer.

• Uninstalling

The application can be uninstalled again via Windows Settings > Apps & Features.

15.4.2 Build the Installer

Before you can build the installer, the project needs to be configured correctly, see below.

In the GitHub repo, go to x releases > Draft/Create a new release. Add a version like 1.0.0 to Tag version, then hit Publish release.

Wait a few minutes and refresh the page: the installer will appear under the release from where you can download it. You can follow the progress under the Actions tab.

15.4.3 Configuration

Excel file

You can add your Excel file to the repository if you like but it’s not a requirement. Configure the Excel file as follows:

• Add the standalone xlwings VBA module, e.g. via xlwings quickstart project --standalone

• Make sure that in the VBA editor (Alt-F11) under Tools > References xlwings is unchecked

• Rename the _xlwings.conf sheet into xlwings.conf
• In the xlwings.conf sheet, as Interpreter, set the following value: %LOCALAPPDATA%\project while replacing project with the name of your project

• If you like, you can hide the xlwings.conf sheet

Source code

Source code can either be embedded in the Excel file (see Embedded Code) or added to the src directory. The first option requires xlwings-pro in requirements.txt, the second option will also work with xlwings.

Dependencies

Add your dependencies to requirements.txt. For example:

```plaintext
xlwings==0.18.0
numpy==1.18.2
```

Code signing (optional)

Using a code sign certificate will show a verified publisher in the installation prompt. Without it, it will show an unverified publisher.

• Store your code sign certificate as sign_cert_file in the root of this repository (make sure your repo is private).

• Go to Settings > Secrets and add the password as code_sign_password.

Project details

Update the following under .github/main.yml:

```yaml
PROJECT:
APP_PUBLISHER:
```

Python version

Set your Python version under .github/main.yml:

```yaml
python-version: '3.7'
arithmetic: 'x64'
```

15.5 Deployment Key

This feature requires xlwings PRO.

If you have an xlwings PRO developer license, you can generate a deployment key. A deployment key allows you to send an xlwings PRO tool to an end user without them requiring a paid license. A deployment key is also perpetual, i.e. doesn’t expire like a developer license.

In return, a deployment key only works with the version of xlwings that was used to generate the deployment key. A developer can generate new deployment keys for new versions of xlwings as long as they have an active xlwings PRO subscription.
Note: You need a paid developer license to generate a deployment key. A trial license won’t work.

To create a deployment key, run the following command:

```
xlwings license deploy
```

Then paste the generated key into the xlwings config as LICENSE_KEY. For deployment purposes, usually the best place to do that is on a sheet called xlwings.conf, but you can also use an xlwings.conf file in either the same folder or in the .xlwings folder within the user’s home folder. To use an environment variable, use XLWINGS_LICENSE_KEY. See also User Settings.
16.1 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

16.2 Issue: Couldn’t find the local location of your OneDrive

Solution:

On either the xlwings.conf sheet or on the xlwings.conf file under your home folder (for location see User Config: Ribbon/Config File), add the following setting:

"ONEDRIVE_WIN", "C:\path\to\OneDrive"

Note: Don’t use quotes on the xlwings.conf sheet and if you are on macOS, use ONEDRIVE_MAC instead.
The purpose of xlwings PRO is to finance 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 license key and at least xlwings v0.19.0. Everything under the xlwings.pro subpackage is distributed under a commercial license.

### 17.1 PRO Features

- *Table.update()*: An easy way to keep an Excel table in sync with a pandas DataFrame
- *Embedded Code*: Store your Python source code directly in Excel for easy deployment.
- *xlwings Reports*: A template based reporting mechanism, allows business users to change the layout of the report without having to change Python code.
- *Plotly static charts*: Support for Plotly static charts.
- *One-Click Zero-Config Installer*: Guarantees that the end user does not need to know anything about Python.
17.2 More Infos

- Pricing: https://www.xlwings.org/pricing
- Trial license key: https://www.xlwings.org/trial
CHAPTER 18

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:

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

Syntax:

<table>
<thead>
<tr>
<th></th>
<th><code>xw.Range</code></th>
<th>UDFs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>reading</strong></td>
<td><code>xw.Range.options(convert=None, **kwargs).value</code></td>
<td><code>@arg('x', convert=None, **kwargs)</code></td>
</tr>
<tr>
<td><strong>writing</strong></td>
<td><code>xw.Range.options(convert=None, **kwargs).value = myvalue</code></td>
<td><code>@ret(convert=None, **kwargs)</code></td>
</tr>
</tbody>
</table>

**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:
18.1 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:

```python
>>> 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:

```python
>>> 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:

```python
@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:
    ```python
    >>> import datetime as dt
    >>> sht.range('A1').options(dates=dt.date).value
    ˓→
    '2017-02-20'
    ```

  - 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:

    ```python
    >>> 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:

    ```python
    @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:

```python
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:

```python
# 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
```

### 18.2 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.
18.2.1 Dictionary converter

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

```python
>>> 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`.

18.2.2 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:**

```python
>>> 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.]])
```

18.2.3 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**:

```python
>>> 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
```

### 18.2.4 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:**

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

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

(continues on next page)
The same sample for UDF (starting in Range('A13') on screenshot) looks like this:

```python
@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
```

### 18.2.5 xw.Range and ‘raw’ converters

Technically speaking, these are “no-converters”.

- If you need access to the `xlwings.Range` object directly, you can do:
  ```python
  @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:
  ```python
  >>> sht.range('A1:B2').value
  [[1.0, 'text'], [datetime.datetime(2016, 2, 1, 0, 0), None]]
  ```

### 18.3 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:
  ```
  >>> 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))
  ```
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:

```python
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:

```python
from xlwings.conversion import Converter, PandasDataFrameConverter

class DataFrameDropna(Converter):
    base = PandasDataFrameConverter

    @staticmethod
    def read_value(builtin_df, options):
        dropna = options.get('dropna', False)
        # set default to False
        if dropna:
            converted_df = builtin_df.dropna()
        else:
            converted_df = builtin_df
        # This will arrive in Python when using the DataFrameDropna converter
        → for reading
        return converted_df

    @staticmethod
```

(continues on next page)
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.:

```python
@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.
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:

![Error Message](image)

**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).

19.1 RunPython

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

```python
# 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:

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

19.2 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):

```python
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.
19.2. UDF debug server
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 add-in with the respective code.

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

### 20.1 In-Excel SQL

The xlwings add-in 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.
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>id</td>
<td>first_name</td>
<td>last_name</td>
<td>age</td>
<td>id</td>
<td>email</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Mariam</td>
<td>Alt</td>
<td>12</td>
<td>1</td>
<td>Mariam@Alt</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>Shenita</td>
<td>Truelove</td>
<td>55</td>
<td>2</td>
<td>Shenita@Truelove</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>Evelyn</td>
<td>Braddy</td>
<td>30</td>
<td>3</td>
<td>Evelyn@Braddy</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>Shery</td>
<td>Sam</td>
<td>35</td>
<td>5</td>
<td>Rogello@Mote</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>Rogello</td>
<td>Mote</td>
<td>88</td>
<td>6</td>
<td>Solomon@Okamura</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>Solomon</td>
<td>Okamura</td>
<td>33</td>
<td>7</td>
<td>Latashia@Alire</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>Jessica</td>
<td>Buelow</td>
<td>10</td>
<td>8</td>
<td>Roselee@Tarwater</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>Latashia</td>
<td>Alire</td>
<td>19</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>Roselee</td>
<td>Tarwater</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>Kiera</td>
<td>Saulsbury</td>
<td>55</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
SELECT a.id, a.first_name, a.last_name, b.email FROM a INNER JOIN b ON a.id = b.id
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>id</td>
<td>first_name</td>
<td>last_name</td>
<td>email</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>Mariam</td>
<td>Alt</td>
<td>Mariam@Alt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>Shenita</td>
<td>Truelove</td>
<td>Shenita@Truelove</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>Evelyn</td>
<td>Braddy</td>
<td>Evelyn@Braddy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>5</td>
<td>Rogello</td>
<td>Mote</td>
<td>Rogello@Mote</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>6</td>
<td>Solomon</td>
<td>Okamura</td>
<td>Solomon@Okamura</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>8</td>
<td>Latashia</td>
<td>Alire</td>
<td>Latashia@Alire</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>9</td>
<td>Roselee</td>
<td>Tarwater</td>
<td>Roselee@Tarwater</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Chapter 20. Extensions
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 standardxlsx files rather than xlsm files. This tutorial assumes you’re familiar with how xlwings and its configuration works.

## 21.1 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 with 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.

### 21.2 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.

### 21.3 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.
21.3. Importing UDFs
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.

### 21.4 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.

### 21.5 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.

### 21.6 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.
21.7 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:

```plaintext
Conda Path: %USERPROFILE%\anaconda3
Conda Env: base
Interpreter_Mac: $HOME/opt/anaconda3/bin/python
```

- **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`. 
New in version 0.13.0.

22.1 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:

```python
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)\n```

(continues on next page)
t.daemon = True
t.start()

    # THIS DOESN'T WORK - passing xlwings objects to threads will fail!
    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:

```python
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()

    q.put(cell)

q.join()

22.2 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

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,
CHAPTER 23

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:

```python
>>> 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 it’s way into the library (cross-platform and with a Pythonic syntax).

23.1 Example: Workaround to use VBA’s `Range.WrapText`

```python
# 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:** New in v0.12.0 and still in a somewhat early stage that involves a bit of manual work. Currently, this functionality is only available on Windows for UDFs. The RunPython functionality is currently not supported.

### 24.1 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:

```python
>>> 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.

24.2 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.
While xlwings is a pure Python package, there are cross-language packages that allow for a relatively straightforward use from/with other languages. This means, however, that you’ll always need to have Python with xlwings installed in addition to R or Julia. We recommend the Anaconda distribution, see also Installation.

25.1 R

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

Setup:

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

25.1.1 Simple functions with R

Original R function that we want to access from Excel (saved in r_file.R):
myfunction <- function(x, y){
  return(x * y)
}

Python wrapper code:

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

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

After importing this function (see: User Defined Functions (UDFs)), it will be available as UDF from Excel.

### 25.1.2 Array functions with R

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

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

Python wrapper code:

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

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

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

After importing this function (see: User Defined Functions (UDFs)), it will be available as UDF from Excel.

### 25.2 Julia

Setup:
• Install Julia and Python

• Run `Pkg.add("PyCall")` from an interactive Julia interpreter

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

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

julia> xw.Book()
PyObject <Book [Workbook1]>

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

julia> xw.Range("A1")[:value]
"Hello World"
```
Chapter 25. xlwings with R and Julia
26.1 Top-level functions

\texttt{xlwings.view(obj, sheet=None, table=True)}

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.

\textbf{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.

\textbf{Parameters}

- \texttt{obj (any type with built-in converter)} – the object to display, e.g. numbers, strings, lists, numpy arrays, pandas dataframes
- \texttt{sheet (Sheet, default None)} – Sheet object. If none provided, the first sheet of a new workbook is used.
- \texttt{table (bool, default True)} – If your object is a pandas DataFrame, by default it is formatted as an Excel Table

\textbf{Examples}

\begin{verbatim}
>>> import xlwings as xw
>>> import pandas as pd
>>> import numpy as np
\end{verbatim}

(continues on next page)
>>> 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.

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

 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

**Examples**

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

See also: view

Changed in version 0.23.1.

## 26.2 Object model

### 26.2.1 Apps

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

A collection of all app objects:

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

**active**

Returns the active app.
New in version 0.9.0.

**add()**


**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.

### 26.2.2 App

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

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

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

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

```python
>>> 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 front-most application and hand over focus from Python to Excel.

New in version 0.9.0.

**api**

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

New in version 0.9.0.

**books**

A collection of all Book objects that are currently open.

New in version 0.9.0.

**calculate** *

Calculates all open books.

New in version 0.3.6.

**calculation**

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

**Examples**

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

Changed in version 0.9.0.

**display_alerts**

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

New in version 0.9.0.

**hwnd**

Returns the Window handle *(Windows-only)*.

New in version 0.9.0.

**kill** *

Forces the Excel app to quit by killing its process.

New in version 0.9.0.
**macro** (*name*)

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

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

**Examples**

This VBA function:

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

can be accessed like this:

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

See also: `Book.macro()`

New in version 0.9.0.

**pid**

Returns the PID of the app.

New in version 0.9.0.

**quit()**

Quits the application without saving any workbooks.

New in version 0.3.3.

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

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

New in version 0.9.0.

**screen_updating**

Turn screen updating off to speed up your script. You won’t be able to see what the script is doing, but it will run faster. Remember to set the screen_updating property back to True when your script ends.

New in version 0.3.3.

**selection**

Returns the selected cells as Range.

New in version 0.9.0.
**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.

**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.

**version**
Returns the Excel version number object.

**Examples**

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

Changed in version 0.9.0.

**visible**
Gets or sets the visibility of Excel to `True` or `False`.
New in version 0.3.3.

### 26.2.3 Books

**class** `xlwings.main.Books` *(impl)*
A collection of all book objects:

```python
>>> import xlwings as xw
>>> xw.books  # active app
>>> xw.apps[10559].books  # specific app, get the PIDs via xw.apps.keys()
```

New in version 0.9.0.

**active**
Returns the active Book.

**add()**

**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.

### 26.2.4 Book

```python
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:

```python
>>> 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:

```python
>>> app = xw.App()  # or something like xw.apps[10559] for existing apps, get the PIDs via xw.apps.keys()
>>> app.books['Book1']
```

<table>
<thead>
<tr>
<th></th>
<th>xw.Book</th>
<th>xw.books</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsaved book</td>
<td><code>xw.Book('Book1')</code></td>
<td><code>xw.books['Book1']</code></td>
</tr>
<tr>
<td>Book by (full)name</td>
<td><code>xw.Book(r'C:/path/to/file.xlsx')</code></td>
<td><code>xw.books.open(r'C:/path/to/file.xlsx')</code></td>
</tr>
</tbody>
</table>

**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_onlyRecommended** *(bool, default False)* – Set to True to mute the read-only recommended message

• **origin** *(int)* – For text files only. Specifies where it originated. Use XlPlatform 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.

**api**

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

New in version 0.9.0.

**app**

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

New in version 0.9.0.

**classmethod caller** *

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

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

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

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

Examples

This VBA function:

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

can be accessed like this:

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

See also: App.macro()

New in version 0.7.1.

name
Returns the name of the book as str.

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

Changed in version 0.9.0.
save(path=None)
Saves the Workbook. If a path is being provided, this works like SaveAs() in Excel. If no path is specified and if the file hasn’t been saved previously, it’s being saved in the current working directory with the current filename. Existing files are overwritten without prompting.

Parameters path (str or path-like object, default None) – Full path to the workbook

Example

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

New in version 0.3.1.

selection
Returns the selected cells as Range.
New in version 0.9.0.

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

Examples

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

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

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

New in version 0.3.1.

sheets
Returns a sheets collection that represents all the sheets in the book.
New in version 0.9.0.

```to_pdf(path=None, include=None, exclude=None)
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.

**Examples**

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

See also `xlwings.Sheet.to_pdf()`

New in version 0.21.1.

### 26.2.5 Sheets

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

A collection of all sheet objects:

```python
>>> 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.

**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.

### 26.2.6 Sheet

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

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

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

Changed in version 0.9.0.

**activate** ()

Activates the Sheet and returns it.

**api**

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

New in version 0.9.0.

**autofit** *(axis=None)*

Autofits the width of either columns, rows or both on a whole Sheet.

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

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

**Examples**

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

New in version 0.2.3.

**book**

Returns the Book of the specified Sheet. Read-only.
cells
Returns a Range object that represents all the cells on the Sheet (not just the cells that are currently in use).
New in version 0.9.0.

charts
See Charts
New in version 0.9.0.

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

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

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

```python
# 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.

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

name  
Gets or sets the name of the Sheet.

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

New in version 0.9.0.

pictures  
See Pictures

New in version 0.9.0.

range (cell1, cell2=None)  
Returns a Range object from the active sheet of the active book, see Range().

New in version 0.9.0.

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.

Returns sheet

Return type  xlwings Sheet

Examples

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

See also xlwings.pro.reports.create_report()

select ()  
Selects the Sheet. Select only works on the active book.

New in version 0.9.0.
shapes
See Shapes
New in version 0.9.0.

tables
See Tables
New in version 0.21.0.

to_pdf (path=None)
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.

Examples

```python
>>> 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.

used_range
Used Range of Sheet.

Returns

Return type xw.Range
New in version 0.13.0.

visible
Gets or sets the visibility of the Sheet (bool).
New in version 0.21.1.

26.2.7 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:**

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

**Specific Sheet:**

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

**add_hyperlink** (*address, text_to_display=*, None, screen_tip=*= None*)

Adds a hyperlink to the specified Range (single Cell)

**Parameters**

- **address** (*str*) – The address of the hyperlink.

- **text_to_display** (*str, default None*) – The text to be displayed for the hyperlink. Defaults to the hyperlink address.

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

New in version 0.3.0.

**address**

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

New in version 0.9.0.

**api**

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

New in version 0.9.0.

**autofit()**

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

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

- To autofit only the height of the rows use `xw.Range('A1:B2').rows.autofit()`
clear ()
Clears the content and the formatting of a Range.

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

color
Gets and sets the background color of the specified Range.
To set the color, either use an RGB tuple (0, 0, 0) or a color constant. To remove the background, set the color to None, see Examples.

    Returns   RGB
    Return type tuple

Examples

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

New in version 0.3.0.

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

    Returns
    Return type Integer

New in version 0.3.5.

column_width
Gets or sets the width, in characters, of a Range. One unit of column width is equal to the width of one character in the Normal style. For proportional fonts, the width of the character 0 (zero) is used.
If all columns in the Range have the same width, returns the width. If columns in the Range have different widths, returns None.

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

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

    Returns
    Return type float
New in version 0.4.0.

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

count
Returns the number of cells.

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

Returns

Return type  Range object

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

derive (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

```python
>>> 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**

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

New in version 0.9.0.

**formula**

Gets or sets the formula for the given Range.

**formula2**

Gets or sets the formula2 for the given Range.

**formula_array**

Gets or sets an array formula for the given Range.

New in version 0.7.1.

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

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

**Parameters**

- **row_absolute** (**bool**, default True) – Set to True to return the row part of the reference as an absolute reference.

- **column_absolute** (**bool**, default True) – Set to True to return the column part of the reference as an absolute reference.

- **include_sheetname** (**bool**, default False) – Set to True to include the Sheet name in the address. Ignored if external=True.

- **external** (**bool**, default False) – Set to True to return an external reference with workbook and worksheet name.
Returns

Return type  str

Examples

```python
>>> 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.

has_array

Are we part of an Array formula?

height

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

Returns

Return type  float

New in version 0.4.0.

hyperlink

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

Examples

```python
>>> 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

last_cell

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

Returns

Return type Range

Example

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

New in version 0.3.5.

left

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

Returns

Return type float

New in version 0.6.0.

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.

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.

merge_cells

Returns True if the Range contains merged cells, otherwise False

name

Sets or gets the name of a Range.

New in version 0.4.0.
number_format

Gets and sets the number_format of a Range.

Examples

```python
>>> 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

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

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

Parameters

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

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

Returns  Range object

Return type  Range
New in version 0.3.0.

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

Returns

Return type  Integer
New in version 0.3.5.

**row_height**

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

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

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

**Returns**

**Return type** float

New in version 0.4.0.

**rows**

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

New in version 0.9.0.

**select ()**

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

New in version 0.9.0.

**shape**

Tuple of Range dimensions.

New in version 0.3.0.

**sheet**

Returns the Sheet object to which the Range belongs.

New in version 0.9.0.

**size**

Number of elements in the Range.

New in version 0.3.0.

**table**

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

New in version 0.21.0.

**top**

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

**Returns**

**Return type** float

New in version 0.6.0.

**unmerge ()**

Separates a merged area into individual cells.
value
Gets and sets the values for the given Range. See 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()`

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

**Returns**

**Return type** float

New in version 0.4.0.

### 26.2.8 RangeRows

class `xlwings.RangeRows(rng)`

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

**Example**

```python
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('A2:C2')

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

autofit()
    Autofits the height of the rows.

count
    Returns the number of rows.

    New in version 0.9.0.

### 26.2.9 RangeColumns

class `xlwings.RangeColumns(rng)`

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

```python
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('B1:B4')
for c in rng.columns:
    print(c.address)
```

`autofit()`

Autofits the width of the columns.

`count`

Returns the number of columns.

New in version 0.9.0.

26.2.10 Shapes

class xlwings.main.Shapes(inp1)

A collection of all shape objects on the specified sheet:

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

New in version 0.9.0.

`api`

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

`count`

Returns the number of objects in the collection.

26.2.11 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.

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.

height

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

New in version 0.5.0.

left

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

New in version 0.5.0.

name

Returns or sets the name of the shape.

New in version 0.5.0.

parent

Returns the parent of the shape.

New in version 0.9.0.

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.

text
Returns or sets the text of a shape.

New in version 0.21.4.

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

New in version 0.5.0.

type
Returns the type of the shape.

New in version 0.9.0.

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

New in version 0.5.0.

26.2.12 Charts

class xlwings.main.Charts (impl)
A collection of all chart objects on the specified sheet:

```python
>>> 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

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

**api**

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

**count**

Returns the number of objects in the collection.

### 26.2.13 Chart

**class** `xlwings.Chart` *(name_or_index=None, impl=None)*

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

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

**api**

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

New in version 0.9.0.

**chart_type**

Returns and sets the chart type of the chart. 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_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.

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

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

name
  Returns or sets the name of the chart.

parent
  Returns the parent of the chart.
  New in version 0.9.0.

set_source_data(source)
  Sets the source data range for the chart.

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

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

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

26.2.14 Pictures

class xlwings.main.Pictures(impl)
A collection of all picture objects on the specified sheet:

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

New in version 0.9.0.

add(image, link_to_file=False, save_with_document=True, left=0, top=0, width=None,
height=None, name=None, update=False, scale=1)
  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 0)* – Left position in points.
- **top** *(float, default 0)* – Top position in points.
- **width** *(float, default None)* – Width in points. If PIL/Pillow is installed, it defaults to the width of the picture. Otherwise it defaults to 100 points.
- **height** *(float, default None)* – Height in points. If PIL/Pillow is installed, it defaults to the height of the picture. Otherwise it defaults to 100 points.
- **name** *(str, default None)* – Excel picture name. Defaults to Excel standard name if not provided, e.g. ‘Picture 1’.
- **update** *(bool, default False)* – Replace an existing picture with the same name. Requires name to be set.

Returns

Return type **Picture**

Examples

1. Picture

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

2. Matplotlib

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

api

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

count

Returns the number of objects in the collection.
26.2.15 Picture

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

Changed in version 0.9.0.

api

- Returns the native object (pywin32 or appscript obj) of the engine being used.
- New in version 0.9.0.

delete()

- Deletes the picture.
- New in version 0.5.0.

height

- Returns or sets the number of points that represent the height of the picture.
- New in version 0.5.0.

left

- Returns or sets the number of points that represent the horizontal position of the picture.
- New in version 0.5.0.

name

- Returns or sets the name of the picture.
- New in version 0.5.0.

parent

- Returns the parent of the picture.
- New in version 0.9.0.

top

- Returns or sets the number of points that represent the vertical position of the picture.
- New in version 0.5.0.

update(image)

- Replaces an existing picture with a new one, taking over the attributes of the existing picture.
- Parameters: `image` - Either a filepath or a Matplotlib figure object.
- New in version 0.5.0.

width

- Returns or sets the number of points that represent the width of the picture.
New in version 0.5.0.

## 26.2.16 Names

class xlwings.main.Names(impl)

A collection of all name objects in the workbook:

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

New in version 0.9.0.

add(name, refers_to)

Defines a new name for a range of cells.

**Parameters**

- **name** *(str)* – Specifies the text to use as the name. Names cannot include spaces and cannot be formatted as cell references.

- **refers_to** *(str)* – Describes what the name refers to, in English, using A1-style notation.

**Returns**

**Return type** Name

New in version 0.9.0.

api

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

New in version 0.9.0.

count

Returns the number of objects in the collection.

## 26.2.17 Name

class xlwings.Name(impl)

The name object is a member of the names collection:

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

New in version 0.9.0.

api

Returns the native object (pywin32 or appscript obj) of the engine being used.
New in version 0.9.0.

**delete()**
Deletes the name.
New in version 0.9.0.

**name**
Returns or sets the name of the name object.
New in version 0.9.0.

**refers_to**
Returns or sets the formula that the name is defined to refer to, in A1-style notation, beginning with an equal sign.
New in version 0.9.0.

**refers_to_range**
Returns the Range object referred to by a Name object.
New in version 0.9.0.

### 26.2.18 Tables

**class** `xlwings.main.Tables` *(impl)*
A collection of all table objects on the specified sheet:

```python
>>> 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**

```python
>>> 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>>
```

### 26.2.19 Table

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

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

```python
>>> 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.

**api**

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

**data_body_range**

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

**display_name**

Returns or sets the display name for the specified Table object

**header_row_range**

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

---

26.2. Object model

147
**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

**name**
Returns or sets the name of the Table.

**parent**
Returns the parent of the table.

**range**
Returns an xlwings range object of the table.

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

**show_headers**
Show or hide the header (read/write)

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

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

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

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

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

**table_style**
Gets or sets the table style. See Tables.add for possible values.

**totals_row_range**
Returns an xlwings range object representing the Total row

**update**(data)
This method requires xlwings PRO

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

Changed in version 0.21.3.

**Parameters**

**data** *(pandas DataFrame)* — Currently restricted to pandas DataFrames. If you want to hide the index, set the first column as the index, e.g. `df.set_index('column_name')`.

**Returns**

**Return type** Table
Examples

```python
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)
```

### 26.2.20 Font

```python
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.

**api**

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

New in version 0.23.0.

**bold**

Returns or sets the bold property (boolean).

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

New in version 0.23.0.

**color**

Returns or sets the color property (tuple).

```python
>>> sheet['A1'].font.color = (255, 0, 0) # RGB tuple
>>> sheet['A1'].font.color
(255, 0, 0)
```
New in version 0.23.0.

**italic**

Returns or sets the italic property (boolean).

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

New in version 0.23.0.

**name**

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

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

New in version 0.23.0.

**size**

Returns or sets the size (float).

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

New in version 0.23.0.

---

### 26.2.21 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 Apple-Script.

New in version 0.23.0.

**api**

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

New in version 0.23.0.

**font**

Returns or sets the text property of a characters object.
New in version 0.23.0.

**text**

Returns or sets the text property of a characters object.

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

New in version 0.23.0.

### 26.2.22 Markdown

**class** `xlwings.pro.Markdown`

```python
text, 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
```

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**

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

```
26.2.23 MarkdownStyle

```python
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:

```text
>>> 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:

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

New in version 0.23.0.
```

26.3 UDF decorators

```python
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.
```

```python
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:

```python
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) Supress the index and header of a returned DataFrame:

```python
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:

```python
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.

## 26.4 Reports

`xlwings.pro.reports.create_report(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, PIL Image objects that have a filename and Matplotlib 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'
- `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** `wb`

**Return type** `xlwings Book`

**Examples**

In `my_template.xlsx`, put the following Jinja variables in two cells: `{{ title }}` and `{{ df }}`

```python
>>> from xlwings.pro.reports import create_report
>>> import pandas as pd
>>> df = pd.DataFrame(data=[[1,2],[3,4]])
>>> wb = create_report('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)
>>> wb = create_report('my_template.xlsx', 'my_report.xlsx', **data)

If you need to handle external links or a password, use it like so:

```python
>>> wb = create_report('my_template.xlsx', 'my_report.xlsx',
    book_settings={'update_links': True, 'password': 'mypassword'},
    **data)
```

You can control the Excel instance by passing in an xlwings App instance. For example, to run the report in a separate and hidden instance of Excel, do the following:

```python
>>> import xlwings as xw
>>> from xlwings.pro.reports import create_report
>>> app = xw.App(visible=False)  # Separate and hidden Excel instance
>>> wb = create_report('my_template.xlsx', 'my_report.xlsx', app=app,
    **data)
>>> app.quit()  # Close the wb and quit the Excel instance
```
CHAPTER 27

REST API

New in version 0.13.0.

27.1 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):

```
{
    "address": "$A$1:$B$2",
    "color": null,
    "column": 1,
    "column_width": 10.0,
    "count": 4,
    "current_region": "$A$1:$B$2",
    "formula": [

(continues on next page)
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.
27.2 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)
```

27.3 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`.

27.4 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.

27.5 Endpoint overview

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Corresponds to</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/book</td>
<td>Book</td>
<td>Finds your workbook across all open instances of Excel and will open it if it can’t find it</td>
</tr>
<tr>
<td>/books</td>
<td>Books</td>
<td>Books collection of the active Excel instance</td>
</tr>
<tr>
<td>/apps</td>
<td>Apps</td>
<td>This allows you to specify the Excel instance you want to work with</td>
</tr>
</tbody>
</table>
27.6 Endpoint details

27.6.1 /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:
GET /book/<fullname_or_name>/sheets

Example response:

```json
{
  "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)
```
GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>

Example response:

```json
{
  "charts": [
    "Chart 1"
  ],
  "name": "Sheet1",
  "names": [
    "Sheet1!mynamel"
  ],
  "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:

```json
{
  "charts": [
    {
      "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>/charts/<chart_name_or_ix>

Example response:

```json
{
  "chart_type": "line",
  "height": 211.0,
  "left": 0.0,
  "name": "Chart 1",
  "top": 0.0,
  "width": 355.0
}
```
"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!mynamel",
      "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!mynamel",
  "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": "C3",
"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:

```json
{
    "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": [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:

```json
{
    "address": "$A$1:$B$2",

```
GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>/shapes

Example response:

```json
{
    "shapes": [
        {
            "height": 211.0,
            "left": 0.0,
            "name": "Chart 1",
            "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/<shape_name_or_ix>

Example response:

```
{
  "height": 211.0,
  "left": 0.0,
  "name": "Chart 1",
  "top": 0.0,
  "type": "chart",
  "width": 355.0
}
```

27.6.2 /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": []
    }
  ]
}
```
GET /books/<book_name_or_ix>

Example response:

```json
{
  "app": 1104,
  "fullname": "C:\Users\felix\DEV\xlwings\scripts\Book1.xlsx",
  "name": "Book1.xlsx",
  "names": ["Sheet1!mynamel", "myname2"],
  "selection": "Sheet2!$A$1",
  "sheets": ["Sheet1", "Sheet2"],
}
```

GET /books/<book_name_or_ix>/names

Example response:

```json
{
  "names": [
    {"name": "Sheet1!mynamel",
      "refers_to": "=Sheet1!$B$2:$C$3"
    },
    ...
  ]
}
```
GET /books/<book_name_or_ix>/names/<name>

Example response:

```json
{
  "name": "myname2",
  "refers_to": "=Sheet1!$A$1"
}
```

GET /books/<book_name_or_ix>/names/<name>/range

Example response:

```json
{
  "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:

```json
{
  "sheets": [
```

27.6. Endpoint details
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": [ 
      "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 1"
   ],
   "name": "Sheet2",
   "names": [ 
   ],
   "pictures": [ 
   ],
   "shapes": [ 
      "Chart 1",
      "Picture 3"
   ],
   "used_range": "$A$1"
}
```
GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/charts/<chart_name_or_ix>

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>/names

Example response:

```
{
    "names": [ 
        {
            "name": "Sheet1!mynamel",
            "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!mynamel",
    "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:

```

Example response:

```
{
    "pictures": [
        {
            "height": 100.0,
            "left": 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$2",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$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]
}
```

(continues on next page)
GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/range/<address>

Example response:

```json
{
  "address": "$A1:$B2",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A1:$B2",
  "formula": [
    ["=1+1.1",
     "a string"],
    ["43395.0064583333",
    ""
    ],
  ],
  "formula_array": null,
  "height": 28.5,
  "last_cell": "$B2",
  "left": 0.0,
  "name": null,
  "number_format": null,
  "row": 1,
  "row_height": 14.3,
  "shape": [2, 2],
  "size": 4,
  "top": 0.0,
  "value": [
    
  ]
}
```

(continues on next page)
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:

{
    "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
}
### 27.6.3 /apps

**GET /apps**

Example response:

```json
{
  "apps": [
    {
      "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$1",
      "version": "16.0",
      "visible": true
    },
    {
      "books": [
        "Book2",
        "Book5"
      ],
      "calculation": "automatic",
      "display_alerts": true,
      "pid": 7920,
      "screen_updating": true,
      "selection": "[Book5]Sheet2!$A$1",
      "version": "16.0",
      "visible": true
    }
  ]
}
```

**GET /apps/<pid>**

Example response:

```json
{
  "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"
}
```
(continues on next page)
GET `/apps/<pid>/books`  

Example response:

```json
{
  "books": [
    {
      "app": 1104,
      "fullname": "Book1",
      "name": "Book1",
      "names": [],
      "selection": "Sheet2!$A$1",
      "sheets": [
        "Sheet1"
      ]
    },
    {
      "app": 1104,
      "fullname": "C:\Users\felix\DEV\xlwings\scripts\Book1.xlsx",
      "name": "Book1.xlsx",
      "names": [
        "Sheet1!mynamel",
        "myname2"
      ],
      "selection": "Sheet2!$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:

```json
{}
```
GET /apps/<pid>/books/<book_name_or_ix>/names

Example response:


GET /apps/<pid>/books/<book_name_or_ix>/names/<name>

Example response:


GET /apps/<pid>/books/<book_name_or_ix>/names/<name>/range

Example response:

(continues on next page)
GET /apps/<pid>/books/<book_name_or_ix>/sheets

Example response:

```json
{
    "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 /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>

Example response:

```json
{
    "charts": [
        "Chart 1"
    ],
    "name": "Sheet1",
    "names": [
        "Sheet1!mynamel"
    ],
    "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:

```json
{
    "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:

```json
{
    "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:

```json
```
GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/names/<sheet_scope_name>

Example response:

```json
{
    "name": "Sheet1!mynamel",
    "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:

```json
{
    "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!mynamel",
    "number_format": "General",
    "row": 2,
    "row_height": 14.3,
    "shape": [2, 2,
        4,
    ],
    "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
        }
    ]
}

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",
    "height": 100.0,
    "left": 0.0,
    "name": "Picture 3",
    "top": 0.0,
    "width": 100.0
}

(continues on next page)
GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/range/<address>

Example response:

```json
{
  "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",
   ""]
}
```
GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/shapes

Example response:

```json
{
    "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,
            "width": 102.0
        }
    ]
}
```
GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/shapes/<shape_name_or_ix>

Example response:

```json
{
    "height": 211.0,
    "left": 0.0,
    "name": "Chart 1",
    "top": 0.0,
    "type": "chart",
    "width": 355.0
}
```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>activate()</code> (xlwings.App method), 116</td>
<td><code>bold</code> (xlwings.main.Font attribute), 149</td>
<td><code>calculate()</code> (xlwings.App method), 116</td>
</tr>
<tr>
<td><code>activate()</code> (xlwings.Book method), 120</td>
<td><code>Book</code> (class in xlwings), 119</td>
<td><code>calculation</code> (xlwings.App attribute), 116</td>
</tr>
<tr>
<td><code>activate()</code> (xlwings.Shape method), 139</td>
<td><code>callers</code> (xlwings.Book attribute), 120</td>
<td><code>clear()</code> (xlwings.Range attribute), 125</td>
</tr>
<tr>
<td><code>activate()</code> (xlwings.Sheet method), 124</td>
<td><code>characters</code> (xlwings.Sheet attribute), 125</td>
<td><code>clear_contents()</code> (xlwings.Range method), 129</td>
</tr>
<tr>
<td><code>active</code> (xlwings.main.Apps attribute), 114</td>
<td><code>Charts</code> (class in xlwings), 140</td>
<td><code>clear_contents()</code> (xlwings.Sheets method), 125</td>
</tr>
<tr>
<td><code>active</code> (xlwings.main.Books attribute), 118</td>
<td><code>Chars</code> (class in xlwings), 141</td>
<td><code>color</code> (xlwings.Range attribute), 129</td>
</tr>
<tr>
<td><code>active</code> (xlwings.main.Sheets attribute), 123</td>
<td><code>Chart</code> (class in xlwings), 141</td>
<td><code>close()</code> (xlwings.Book method), 121</td>
</tr>
<tr>
<td><code>add()</code> (xlwings.main.Apps method), 115</td>
<td><code>Chart_type</code> (xlwings.Chart attribute), 141</td>
<td><code>color</code> (xlwings.main.Font attribute), 149</td>
</tr>
<tr>
<td><code>add()</code> (xlwings.main.Books method), 118</td>
<td><code>Charts</code> (class in xlwings), 140</td>
<td><code>column</code> (xlwings.Range attribute), 129</td>
</tr>
<tr>
<td><code>add()</code> (xlwings.main.Charts method), 140</td>
<td><code>charts</code> (xlwings.Sheet attribute), 125</td>
<td><code>column_width</code> (xlwings.Range attribute), 129</td>
</tr>
<tr>
<td><code>add()</code> (xlwings.main.Shapes method), 138</td>
<td><code>clear()</code> (xlwings.RangeColumns method), 138</td>
<td><code>columns</code> (xlwings.Range attribute), 130</td>
</tr>
<tr>
<td><code>add()</code> (xlwings.main.Pictures method), 142</td>
<td><code>clear()</code> (xlwings.RangeRows method), 137</td>
<td><code>copy()</code> (xlwings.Range method), 130</td>
</tr>
<tr>
<td><code>add()</code> (xlwings.main.Sheets method), 123</td>
<td><code>clear_contents()</code> (xlwings.Range method), 129</td>
<td></td>
</tr>
</tbody>
</table>
copy() (xlwings.Sheet method), 125
count (xlwings.main.Apps attribute), 115
count (xlwings.main.Charts attribute), 141
count (xlwings.main.Names attribute), 145
count (xlwings.main.Shapes attribute), 138
count (xlwings.Range attribute), 130
count (xlwings.RangeColumns attribute), 138
count (xlwings.RangeRows attribute), 137
create_report() (in module xlwings.pro.reports), 154
current_region (xlwings.Range attribute), 130
data_body_range (xlwings.main.Table attribute), 147
delete() (xlwings.Chart method), 142
delete() (xlwings.Name method), 146
delete() (xlwings.Picture method), 144
delete() (xlwings.Shape method), 139
delete() (xlwings.Range method), 130
delete() (xlwings.Sheet method), 125
display_alerts (xlwings.App attribute), 116
display_name (xlwings.main.Table attribute), 147
end() (xlwings.Range method), 130
expand() (xlwings.Range method), 131
Font (class in xlwings.main), 149
font (xlwings.main.Characters attribute), 150
formula (xlwings.Range attribute), 131
formula2 (xlwings.Range attribute), 131
formula_array (xlwings.Range attribute), 131
fullname (xlwings.Book attribute), 121
get_address() (xlwings.Range method), 131
has_array (xlwings.Range attribute), 132
header_row_range (xlwings.main.Table attribute), 147
height (xlwings.Chart attribute), 142
height (xlwings.Picture attribute), 144
height (xlwings.Range attribute), 132
height (xlwings.Shape attribute), 139
hwnd (xlwings.App attribute), 116
hyperlink (xlwings.Range attribute), 132
index (xlwings.Sheet attribute), 126
insert() (xlwings.Range method), 132
insert_row_range (xlwings.main.Table attribute), 147
italic (xlwings.main.Font attribute), 150
keys() (xlwings.main.Apps method), 115
kill() (xlwings.App method), 116
last_cell (xlwings.Range attribute), 133
left (xlwings.Chart attribute), 142
left (xlwings.Picture attribute), 144
left (xlwings.Range attribute), 133
left (xlwings.Shape attribute), 139
load() (in module xlwings), 114
macro() (xlwings.App method), 116
macro() (xlwings.Book method), 121
Markdown (class in xlwings.pro), 151
MarkdownStyle (class in xlwings.pro), 152
merge() (xlwings.Range method), 133
merge_area (xlwings.Range attribute), 133
merge_cells (xlwings.Range attribute), 133
Name (class in xlwings), 145
name (xlwings.Book attribute), 121
name (xlwings.Chart attribute), 142
name (xlwings.main.Font attribute), 150
name (xlwings.main.Table attribute), 148
name (xlwings.Name attribute), 146
name (xlwings.Picture attribute), 144
name (xlwings.Range attribute), 133
name (xlwings.Shape attribute), 139
name (xlwings.Sheet attribute), 126
Names (class in xlwings.main), 145
names (xlwings.Book attribute), 121
names (xlwings.Sheet attribute), 126
number_format (xlwings.Range attribute), 133
| O | offset() (xlwings.Range method), 134 |
|   | open() (xlwings.main.Books method), 118 |
|   | options() (xlwings.Range method), 134 |
| P | parent (xlwings.Chart attribute), 142 |
|   | parent (xlwings.main.Table attribute), 148 |
|   | parent (xlwings.Picture attribute), 144 |
|   | paste() (xlwings.Range method), 135 |
|   | Picture (class in xlwings), 144 |
|   | Pictures (class in xlwings.main), 142 |
|   | pictures (xlwings.Sheet attribute), 126 |
|   | pid (xlwings.App attribute), 117 |
| Q | quit() (xlwings.App method), 117 |
| R | Range (class in xlwings), 127 |
|   | range (xlwings.main.Table attribute), 148 |
|   | range() (xlwings.App method), 117 |
|   | range() (xlwings.Sheet method), 126 |
|   | RangeColumns (class in xlwings), 137 |
|   | RangeRows (class in xlwings), 137 |
|   | raw_value (xlwings.Range attribute), 135 |
|   | refers_to (xlwings.Name attribute), 146 |
|   | refers_to_range (xlwings.Name attribute), 146 |
|   | render_template() (xlwings.Sheet method), 126 |
|   | resize() (xlwings.Range method), 135 |
|   | row (xlwings.Range attribute), 135 |
|   | rows (xlwings.Range attribute), 136 |
| S | save() (xlwings.Book method), 121 |
|   | scale_height() (xlwings.Shape method), 139 |
|   | scale_width() (xlwings.Shape method), 139 |
|   | screen_updating (xlwings.App attribute), 117 |
|   | select() (xlwings.Range method), 136 |
|   | select() (xlwings.Sheet method), 126 |
|   | selection (xlwings.App attribute), 117 |
|   | selection (xlwings.Book attribute), 122 |
|   | set_mock_caller() (xlwings.Book method), 122 |
|   | set_source_data() (xlwings.Chart method), 142 |
|   | Shape (class in xlwings), 138 |
|   | shape (xlwings.Range attribute), 136 |
|   | Shapes (class in xlwings.main), 138 |
|   | shapes (xlwings.Sheet attribute), 126 |
|   | Sheet (class in xlwings), 124 |
|   | sheet (xlwings.Range attribute), 136 |
|   | Sheets (class in xlwings.main), 123 |
|   | sheets (xlwings.Book attribute), 122 |
|   | show Autofilter (xlwings.main.Table attribute), 148 |
|   | show_headers (xlwings.main.Table attribute), 148 |
|   | show_table_style_column_stripes (xlwings.main.Table attribute), 148 |
|   | show_table_style_first_column (xlwings.main.Table attribute), 148 |
|   | show_table_style_last_column (xlwings.main.Table attribute), 148 |
|   | show_table_style_row_stripes (xlwings.main.Table attribute), 148 |
|   | show_totals (xlwings.main.Table attribute), 148 |
|   | size (xlwings.main.Font attribute), 150 |
|   | size (xlwings.Range attribute), 136 |
|   | startup_path (xlwings.App attribute), 117 |
|   | status_bar (xlwings.App attribute), 118 |
| T | Table (class in xlwings.main), 147 |
|   | table (xlwings.Range attribute), 136 |
|   | table_style (xlwings.main.Table attribute), 148 |
|   | Tables (class in xlwings.main), 146 |
|   | tables (xlwings.Sheet attribute), 127 |
|   | text (xlwings.main.Characters attribute), 151 |
|   | text (xlwings.Shape attribute), 140 |
|   | to_pdf() (xlwings.Book method), 122 |
|   | to_pdf() (xlwings.Sheet method), 127 |
|   | top (xlwings.Chart attribute), 142 |
|   | top (xlwings.Picture attribute), 144 |
|   | top (xlwings.Range attribute), 136 |
|   | top (xlwings.Shape attribute), 140 |
|   | totals_row_range (xlwings.main.Table attribute), 148 |
|   | type (xlwings.Shape attribute), 140 |
| U | unmerge() (xlwings.Range method), 136 |
update() (xlwings.main.Table method), 148
update() (xlwings.Picture method), 144
used_range (xlwings.Sheet attribute), 127

V
value (xlwings.Range attribute), 136
version (xlwings.App attribute), 118
view() (in module xlwings), 113
visible (xlwings.App attribute), 118
visible (xlwings.Sheet attribute), 127

W
width (xlwings.Chart attribute), 142
width (xlwings.Picture attribute), 144
width (xlwings.Range attribute), 137
width (xlwings.Shape attribute), 140

X
xlwings (module), 113
xlwings.arg() (in module xlwings), 152
xlwings.func() (in module xlwings), 152
xlwings.pro.reports (module), 154
xlwings.ret() (in module xlwings), 153
xlwings.sub() (in module xlwings), 152