xlwings - Make Excel Fly!

Release dev

Zoomer Analytics LLC

Dec 27, 2022

GETTING STARTED

31 What's New   159

VIDEO COURSE

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

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

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

INSTALLATION

2.1 Prerequisites

- xlwings (Open Source) requires an **installation of Excel** and therefore only works on **Windows** and **macOS**. Note that macOS currently does not support UDFs.

- **xlwings PRO offers additional features:**

    - *File Reader* (new in v0.28.0): Runs additionally on Linux and doesn't require an installation of Excel.

    - *xlwings Server* (new in v0.26.0). Runs additionally on Linux and doesn't require a local installation of Python. Works with Desktop Excel on Windows and macOS as well as with Excel on the web and Google Sheets.

- xlwings requires at least Python 3.7.

Here are previous versions of xlwings that support older versions of Python:

- Python 3.6: 0.25.3

- 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 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
```

2.3 Add-in

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

```
xlwings addin install
```

To automate 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 run `xlwings add install` again after upgrading the xlwings package.

Note: When you are on macOS and are using the VBA standalone module instead of the add-in, you need to run $ `xlwings runpython install` once.

2.4 Dependencies

For automating Excel, you'll need the following dependencies:

- **Windows**: `pywin32`
- **Mac**: `psutil`, `appscript`

The dependencies are automatically installed via `conda` or `pip`. If you would like to install xlwings without dependencies, you can run `pip install xlwings --no-deps`.

2.5 How to activate xlwings PRO

See *xlwings PRO*.

2.6 Optional Dependencies

- NumPy

- pandas

- Matplotlib

- Pillow

- Jinja2 (for xlwings.pro.reports)

- requests (for permissioning)

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

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

2.7 Update

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

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

or:

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

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

```
xlwings addin install
```

2.8 Uninstall

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

```
xlwings addin remove
```

Then

```
pip uninstall xlwings
```

or:

```
conda remove xlwings
```

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

QUICKSTART

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

3.1 1. Interacting with Excel from a Jupyter notebook

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

3.2 2. Scripting: Automate/interact with Excel from Python

Establish a connection to a workbook:

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

If you have the same file open in two instances of Excel, you need to fully qualify it and include the app instance. You will find your app instance key (the PID) via `xw.apps.keys()`:

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

Instantiate a sheet object:

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

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

```
>>> sheet['A1'].value = 'Foo 1'
>>> sheet['A1'].value
'Foo 1'
```

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

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

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

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

Matplotlib figures can be shown as pictures in Excel:

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

3.3 3. Macros: Call Python from Excel

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

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

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

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

Note: Per default, RunPython expects `hello.py` in the same directory as the Excel file with the same name, **but you can change both of these things**: if your Python file is an 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]['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 f'Hello {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)*.

CONNECT TO A BOOK

4.1 Python to Excel

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

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

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

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

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

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

4.2 Excel to Python (RunPython)

To reference the calling book when using RunPython in VBA, use `xw.Book.caller()`, see *Call Python with "RunPython"*. Check out the section about *Debugging* to see how you can call a script from both sides, Python and Excel, without the need to constantly change between `xw.Book.caller()` and one of the methods explained above.

4.3 User Defined Functions (UDFs)

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

SYNTAX OVERVIEW

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

All code samples below depend on the following import:

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

5.1 Active Objects

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

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

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

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

```
import xlwings as xw
sheet1 = xw.Book("MyBook.xlsx").sheets[0]

sheet1.range("A1")
sheet1.range("A1:C3")
sheet1.range((1,1))
sheet1.range((1,1), (3,3))
sheet1.range("NamedRange")

# Or using index/slice notation
sheet1["A1"]
sheet1["A1:C3"]
sheet1[0, 0]
```

```
sheet1[0:4, 0:4]
sheet1["NamedRange"]
```

5.2 Full qualification

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

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

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

5.3 App context manager

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

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

5.4 Range indexing/slicing

Range objects support indexing and slicing, a few examples:

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

5.5 Range Shortcuts

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

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

5.6 Object Hierarchy

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

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

DATA STRUCTURES TUTORIAL

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

All code samples below depend on the following import:

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

6.1 Single Cells

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

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

6.2 Lists

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

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

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

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

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

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

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

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

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

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

6.3 Range expanding

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

```
>>> sheet = xw.Book().sheets[0]
>>> sheet['A1'].value = [[1,2], [3,4]]
>>> range1 = sheet['A1'].expand('table')  # or just .expand()
>>> range2 = sheet['A1'].options(expand='table')
>>> range1.value
[[1.0, 2.0], [3.0, 4.0]]
>>> range2.value
[[1.0, 2.0], [3.0, 4.0]]
>>> sheet['A3'].value = [5, 6]
>>> range1.value
[[1.0, 2.0], [3.0, 4.0]]
>>> range2.value
[[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]]
```

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

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

6.4 NumPy arrays

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

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

6.5 Pandas DataFrames

```
>>> sheet = xw.Book().sheets[0]
>>> df = pd.DataFrame([[1.1, 2.2], [3.3, None]], columns=['one', 'two'])
>>> df
   one  two
0  1.1  2.2
1  3.3  NaN
>>> sheet['A1'].value = df
>>> sheet['A1:C3'].options(pd.DataFrame).value
   one  two
0  1.1  2.2
1  3.3  NaN
# options: work for reading and writing
>>> sheet['A5'].options(index=False).value = df
>>> sheet['A9'].options(index=False, header=False).value = df
```

6.6 Pandas Series

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

```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
```

ADD-IN & SETTINGS

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

Note: The ribbon of the add-in is compatible with Excel >= 2007 on Windows and >= 2016 on macOS. On macOS, all UDF related functionality is not available.

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

7.1 Run main

New in version 0.16.0.

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

7.2 Installation

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

```
xlwings addin install
```

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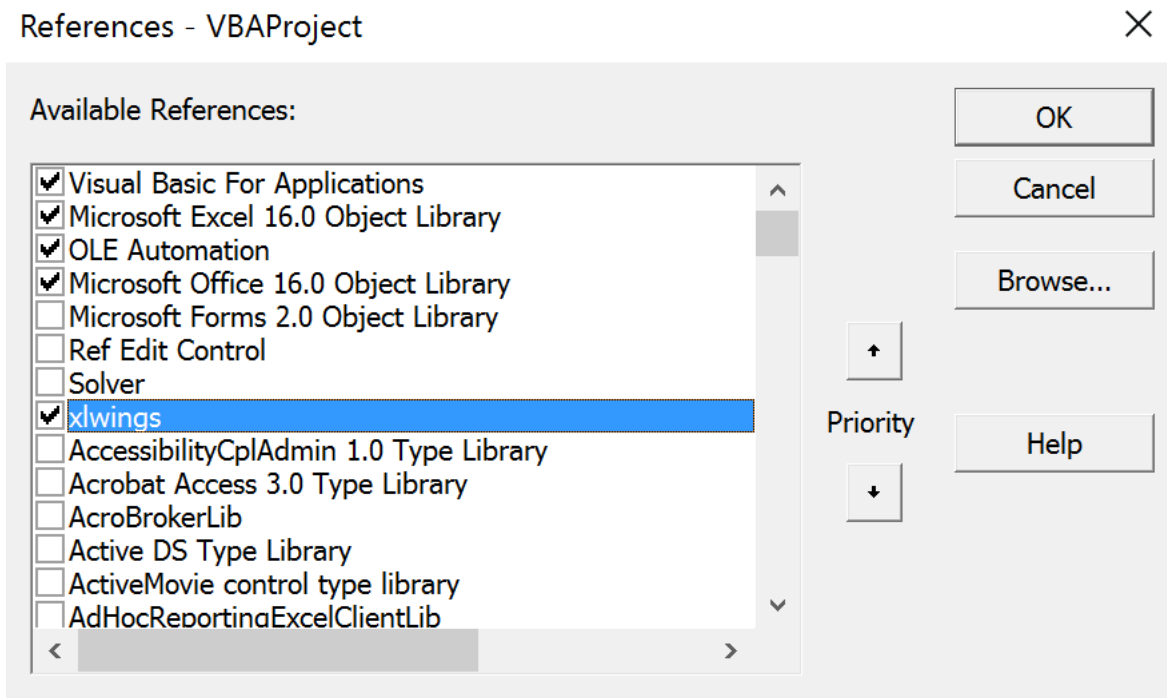
7.3 User Settings

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

- `Interpreter`: This is the path to the Python interpreter. This works also with virtual or conda envs on Mac. If you use conda envs on Windows, then leave this empty and use `Conda Path` and `Conda Env` below instead. Examples: `"C:\Python39\pythonw.exe"` or `"/usr/local/bin/python3.9"`. Note that in the settings, this is stored as `Interpreter_Win` or `Interpreter_Mac`, respectively, see below!

- `PYTHONPATH`: If the source file of your code is not found, add the path to its directory here.

- `Conda Path`: If you are on Windows and use Anaconda or Miniconda, then type here the path to your installation, e.g. `C:\Users\Username\Miniconda3` or `%USERPROFILE%\Anaconda`. NOTE that you need at least conda 4.6! You also need to set `Conda Env`, see next point.

- `Conda Env`: If you are on Windows and use Anaconda or Miniconda, type here the name of your conda env, e.g. `base` for the base installation or `myenv` for a conda env with the name `myenv`.

- `UDF Modules`: Names of Python modules (without .py extension) from which the UDFs are being imported. Separate multiple modules by ";". Example: `UDF_MODULES = "common_udfs;myproject"` The default imports a file in the same directory as the Excel spreadsheet with the same name but ending in `.py`.

- `Debug UDFs`: Check this box if you want to run the xlwings COM server manually for debugging, see *Debugging*.

- `RunPython:  Use UDF Server`: Uses the same COM Server for RunPython as for UDFs. This will be faster, as the interpreter doesn't shut down after each call.

- `Restart UDF Server`: This restarts the UDF Server/Python interpreter.

- `Show Console`: Check the box in the ribbon or set the config to `TRUE` if you want the command prompt to pop up. This currently only works on Windows.

- `ADD_WORKBOOK_TO_PYTHONPATH`: Uncheck this box to not automatically add the directory of your workbook to the PYTHONPATH. This can be helpful if you experience issues with OneDrive/SharePoint: uncheck this box and provide the path where your source file is manually via the PYTHONPATH setting.

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 Making use of Environment Variables

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

- On Windows, you can use all environment variables like so: `%USERPROFILE%\Anaconda`.

- On macOS, the following special variables are supported: `$HOME, $APPLICATIONS, $DOCUMENTS, $DESKTOP`.

7.5 User Config: Ribbon/Config File

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

- Windows: `.xlwings\xlwings.conf` in your home folder, that is usually `C:\Users\<username>`

- macOS: `~/Library/Containers/com.microsoft.Excel/Data/xlwings.conf`

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

```
"INTERPRETER_WIN","C:\path\to\python.exe"
"INTERPRETER_MAC","/path/to/python"
"PYTHONPATH",""
"ADD_WORKBOOK_TO_PYTHONPATH",""
"CONDA PATH",""
"CONDA ENV",""
"UDF MODULES",""
"DEBUG UDFS",""
"USE UDF SERVER",""
"SHOW CONSOLE",""
"ONEDRIVE_CONSUMER_WIN",""
"ONEDRIVE_CONSUMER_WIN",""
"ONEDRIVE_COMMERCIAL_WIN",""
"ONEDRIVE_COMMERCIAL_MAC",""
"SHAREPOINT_WIN",""
"SHAREPOINT_MAC",""
```

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

7.6 Workbook Directory Config: Config file

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

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

7.7 Workbook Config: xlwings.conf Sheet

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

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

7.8 Alternative: Standalone VBA module

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

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

RUNPYTHON

8.1 xlwings add-in

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

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

```
$ xlwings quickstart myproject
```

8.2 Call Python with "RunPython"

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

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

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

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


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

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

Note: Place `xw.Book.caller()` within the function that is being called from Excel and not outside as global variable. Otherwise it prevents Excel from shutting down properly upon exiting and leaves you with

---

a zombie process when you use `Use UDF Server = True`.

8.3 Function Arguments and Return Values

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

USER DEFINED FUNCTIONS (UDFS)

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

Note:

- UDFs are currently only available on Windows.

- For details of how to control the behaviour of the arguments and return values, have a look at *Converters and Options*.

- For a comprehensive overview of the available decorators and their options, check out the corresponding API docs: *UDF decorators*.

9.1 One-time Excel preparations

1) Enable `Trust access to the VBA project object model` under `File > Options > Trust Center > Trust Center Settings > Macro Settings`. You only need to do this once. Also, this is only required for importing the functions, i.e. end users won't need to bother about this.

2) Install the add-in via command prompt: `xlwings addin install` (see *Add-in & Settings*).

9.2 Workbook preparation

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

9.3 A simple UDF

The default addin settings expect a Python source file in the way it is created by `quickstart`:

- in the same directory as the Excel file

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

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

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

```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:

- 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:

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):
```

(continues on next page)

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

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

9.7 Dynamic Array Formulas

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

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

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

```python
import numpy as np

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

Note:

- Expanding array formulas will overwrite cells without prompting

- Pre v0.15.0 doesn't allow to have volatile functions as arguments, e.g. you cannot use functions like `=TODAY()` as arguments. Starting with v0.15.0, you can use volatile functions as input, but the UDF will be called more than 1x.

- Dynamic Arrays have been refactored with v0.15.0 to be proper legacy arrays: To edit a dynamic array with xlwings >= v0.15.0, you need to hit `Ctrl-Shift-Enter` while in the top left cell. Note that you don't have to do that when you enter the formula for the first time.

B4			×	✓	*fx*	=dynamic_array(B2,C2)	

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

B4			×	✓	*fx*	=dynamic_array(B2,C2)	

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

9.8 Docstrings

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

```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:
    # =get_caller_address()
    return caller.address
```

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

9.10 The "vba" keyword

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

```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
Dim i As Long, j As Long

    arr = my_imported_function(...)

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

End Sub
```

9.13 Asynchronous UDFs

Note: This is an experimental feature

New in version v0.14.0.

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

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

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

```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.

MATPLOTLIB & PLOTLY CHARTS

10.1 Matplotlib

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

10.1.1 Getting started

The easiest sample boils down to:

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

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

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

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

10.1.2 Full integration with Excel

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

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

```python
@xw.func
def myplot(n, caller):
    fig = plt.figure()
```
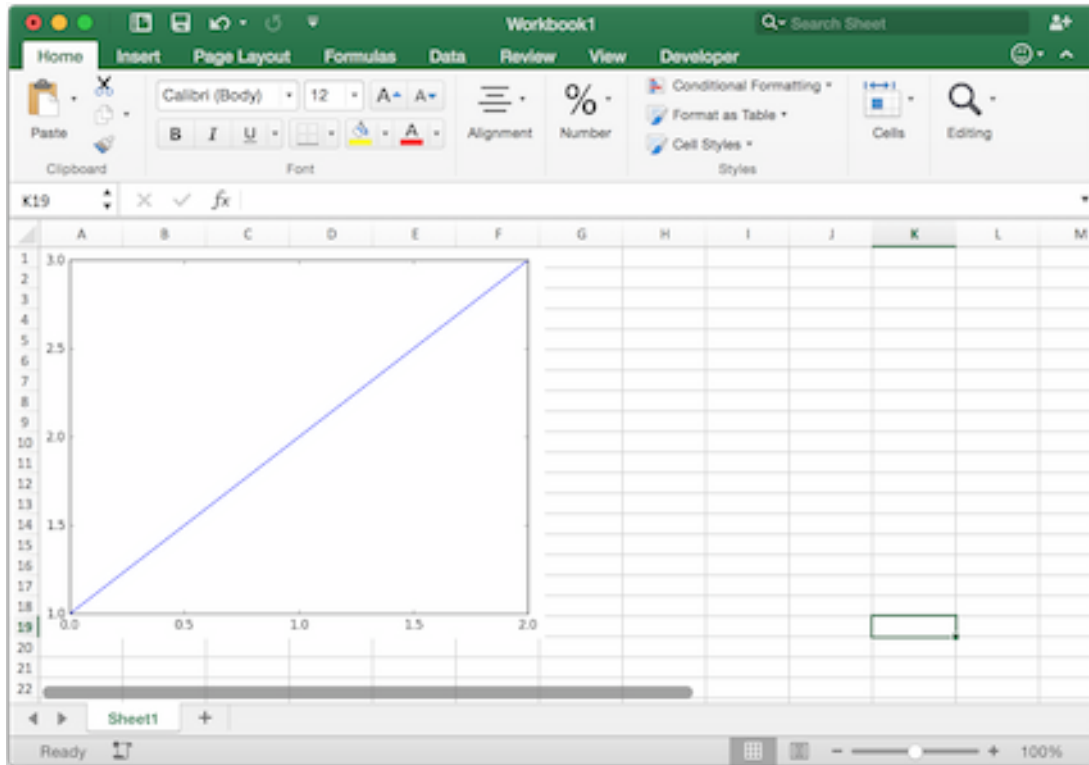
```
        plt.plot(range(int(n)))
        caller.sheet.pictures.add(fig, name='MyPlot', update=True)
        return 'Plotted with n={}'.format(n)
```

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:

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

or:

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

10.1.4 Getting a Matplotlib figure

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

- via PyPlot interface:

```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))
ax = fig.add_subplot(111)
ax.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'])
ax = df.plot(kind='bar')
fig = ax.get_figure()
```

Note: When working with Google Sheets, you can use a maximum of 1 million pixels per picture. Total pixels is a function of figure size and dpi: (width in inches * dpi) * (height in inches * dpi). For example, `fig = plt.figure(figsize=(6, 4))` with 200 dpi (default dpi when using `pictures.add()`) will result in (6 * 200) * (4 * 200) = 960,000 px. To change the dpi, provide `export_options`: `pictures.add(fig, export_options={"bbox_inches": "tight", "dpi": 300})`. Existing figure size can be checked via `fig.get_size_inches()`. pandas also accepts `figsize` like so: `ax = df.plot(figsize=(3, 3))`. Note that `"bbox_inches": "tight"` crops the image and therefore will reduce the number of pixels in a non-deterministic way. `export_options` will be passed to `figure.figsave()` when using Matplotlib and to `figure.write_image()` when using Plotly.

10.2 Plotly static charts

10.2.1 Prerequisites

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

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

or conda:

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

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

10.2.2 How to use

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

```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)
```
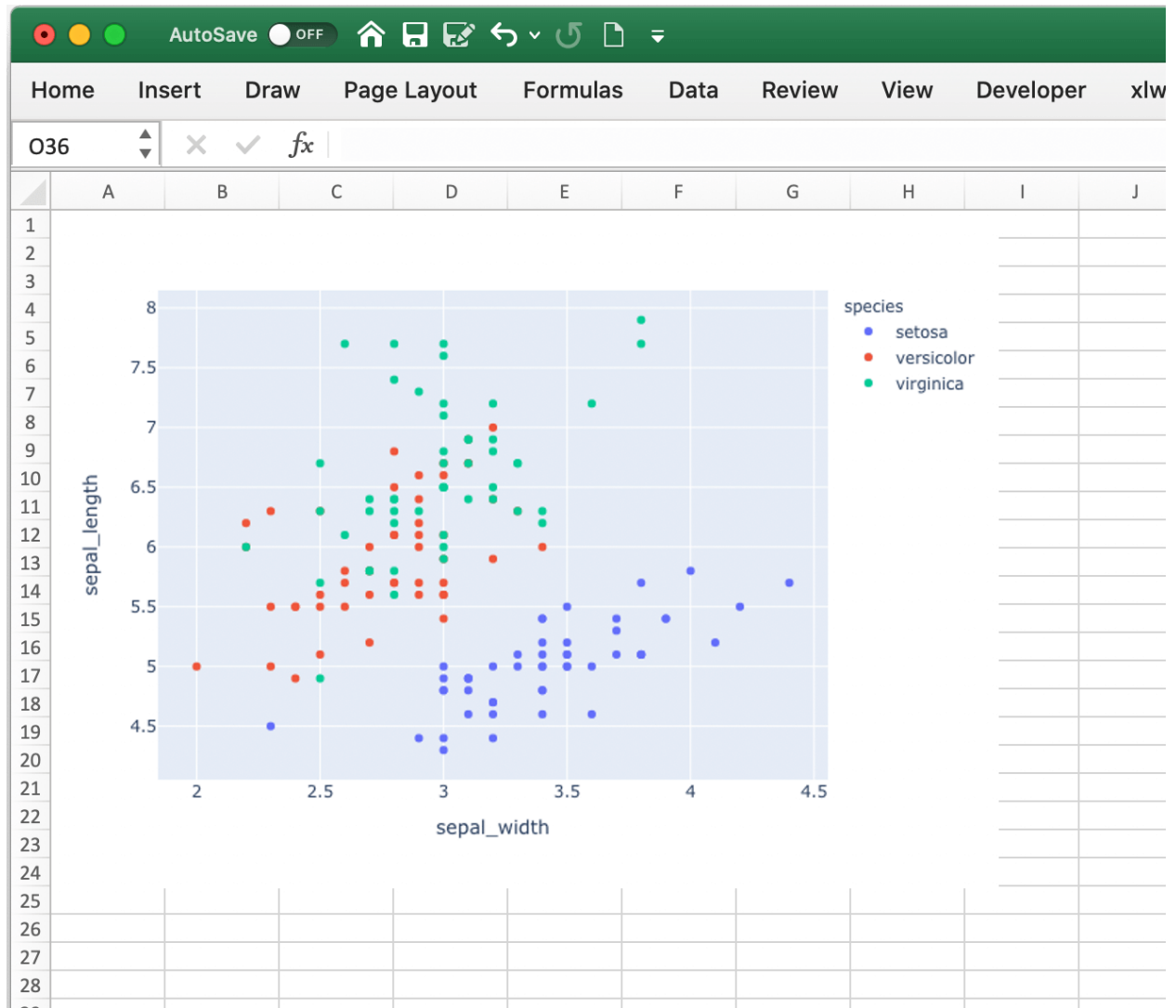
JUPYTER NOTEBOOKS: INTERACT WITH EXCEL

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.

COMMAND LINE CLIENT (CLI)

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

addin	Run "xlwings addin install" to install the Excel add-in (will be copied to the user's XLSTART folder). Instead of "install" you can also use "update", "remove" or "status". Note that this command may take a while. You can install your custom add-in by providing the name or path via the --file/-f flag, e.g. "xlwings add-in install -f custom.xlam or copy all Excel files in a directory to the XLSTART folder by providing the path via the --dir flag." To install the add-in for every user globally, use the --glob/-g flag and run this command from an Elevated Command Prompt. (New in 0.6.0, the --dir flag was added in 0.24.8 and the --glob flag in 0.28.4)
quickstart	Run "xlwings quickstart myproject" to create a folder called "myproject" in the current directory with an Excel file and a Python file, ready to be used. Use the "--standalone" flag to embed all VBA code in the Excel file and make it work without the xlwings add-in. Use "--fastapi" for creating a project that uses a remote Python interpreter. Use "--addin --ribbon" to create a template for a custom ribbon addin. Leave away the "--ribbon" if you don't want a ribbon tab.
runpython	macOS only: run "xlwings runpython install" 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)
restapi	Use "xlwings restapi run" to run the xlwings REST API via Flask dev server. Accepts "--host" and "--port" as

	optional arguments.
license	xlwings PRO: Use "xlwings license update -k KEY" where "KEY" is your personal (trial) license key. This will update ~/.xlwings/xlwings.conf with the LICENSE_KEY 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 workbook's dir in your active Excel file. Use the "--file" flag to only import a single file by providing its path. Requires xlwings PRO. (Changed in 0.23.4)
permission	"xlwings permission cwd" prints a JSON string that can be used to permission the execution of all modules in the current working directory via GET request. "xlwings permission book" does the same for code that is embedded in the active workbook. (New in 0.23.4)
release	Run "xlwings release" to configure your active workbook to work with a one-click installer for easy deployment. Requires xlwings PRO. (New in 0.23.4)
copy	Run "xlwings copy os" to copy the xlwings Office Scripts module. Run "xlwings copy gs" to copy the xlwings Google Apps Script module. Run "xlwings copy vba" to copy the standalone xlwings VBA module. Run "xlwings copy vba --addin" to copy the xlwings VBA module for custom add-ins. (New in 0.26.0, 'vba' added in 0.28.7)
auth	Microsoft Azure AD: "xlwings auth azuread", see https://docs.xlwings.org/en/stable/server_authentication.html (New in 0.28.6)
vba	This functionality allows you to easily write VBA code in an external editor: run "xlwings vba edit" to update the VBA modules of the active workbook from their local exports everytime you hit save. If you run this the first time, the modules will be exported from Excel into your current working directory. To

```
overwrite the local version of the modules with those
from Excel, run "xlwings vba export". To overwrite the
VBA modules in Excel with their local versions, run
"xlwings vba import". The "--file/-f" flag allows you
to specify a file path instead of using the active
Workbook. Requires "Trust access to the VBA project
object model" enabled. NOTE: Whenever you change
something in the VBA editor (such as the layout of a
form or the properties of a module), you have to run
"xlwings vba export".
(New in 0.26.3, changed in 0.27.0)
```

DEPLOYMENT

13.1 Zip files

New in version 0.15.2.

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

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

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

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

13.2 RunFrozenPython

Changed in version 0.15.2.

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

Note:

- This does not work with UDFs.

- Currently only available on Windows, but support for Mac should be easy to add.

- You need at least 0.15.2 to support arguments whereas the syntax changed in 0.15.6

Use it as follows:

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

ONEDRIVE AND SHAREPOINT

Since v0.27.4, xlwings works with locally synced files on OneDrive, OneDrive for Business, and Share-Point. Some constellations will work out-of-the-box, while others require you to edit the configuration via the `xlwings.conf` file (see *User Config*) or the workbook's `xlwings.conf` sheet (see *Workbook Config*).

Note: This documentation is for OneDrive and SharePoint files that are synced to a local folder. This means that both, the Excel and Python file, need to show the green check mark in the File Explorer/Finder as status—a cloud icon will not work. If, in turn, you are looking for the documentation to run xlwings with Excel on the web, see *xlwings Server PRO*.

An easy workaround if you run into issues is to:

- Disable the `ADD_WORKBOOK_TO_PYTHONPATH` setting (either via the checkbox on the Ribbon or via the settings in the `xlwings.conf` sheet).

- Add the directory of your Python source file to the `PYTHONPATH`—again, either via Ribbon or `xlwings.conf` sheet.

If you are using the PRO version, you could instead also embed your code to get around these issues.

For a bit more flexibility, follow the solutions below.

14.1 OneDrive (Personal)

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

- **Windows** (Example):

ONEDRIVE_CONSUMER_WIN	%USERPROFILE%\OneDrive

- **macOS** (Example):

ONEDRIVE_CONSUMER_MAC	$HOME/OneDrive

14.2 OneDrive for Business

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

ONEDRIVE_COMMERCIAL_WIN	%USERPROFILE%\OneDrive - My Company LLC

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

ONEDRIVE_COMMERCIAL_MAC	$HOME/OneDrive - My Company LLC

14.3 SharePoint (Online and On-Premises)

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

- **Windows**:

SHAREPOINT_WIN	%USERPROFILE%\My Company LLC

- **macOS**:

SHAREPOINT_MAC	$HOME/My Company LLC

14.4 Implementation Details & Limitations

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

URLs for OneDrive and OneDrive for Business can be translated fairly straight forward to the local equivalent. You will need to know the root directory of the local drive though: on Windows, these are usually provided via environment variables for OneDrive. On macOS they don't exist, which is the reason why you need to provide the root directory for OneDrive. On Windows, the root directory for SharePoint can sometimes be derived from the env vars, too, but this is not guaranteed. On macOS, you'll need to provide it always anyway.

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

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

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

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

TROUBLESHOOTING

15.1 Issue: dll not found

Solution:

1) `xlwings32-<version>.dll` and `xlwings64-<version>.dll` are both in the same directory as your `python.exe`. If not, something went wrong with your installation. Reinstall it with `pip` or `conda`, see *Installation*.

2) Check your `Interpreter` in the add-in or config sheet. If it is empty, then you need to be able to open a windows command prompt and type `python` to start an interactive Python session. If you get the error `'python' is not recognized as an internal or external command, operable program or batch file.`, then you have two options: Either add the path of where your `python.exe` lives to your Windows path (see https://www.computerhope.com/issues/ch000549.htm) or set the full path to your interpreter in the add-in or your config sheet, e.g. `C:\Users\MyUser\anaconda\pythonw.exe`

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

Solution:

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

CONVERTERS AND OPTIONS

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

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

All code samples below depend on the following import:

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

Syntax:

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

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

```
myrange.options(pd.DataFrame, index=False, numbers=int).value
```

16.1 Default Converter

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

- single cells are read in as `floats` in case the Excel cell holds a number, as `unicode` in case it holds text, as `datetime` if it contains a date and as `None` in case it is empty.

- columns/rows are read in as lists, e.g. `[None, 1.0, 'a string']`

- 2d cell ranges are read in as list of lists, e.g. `[[None, 1.0, 'a string'], [None, 2.0, 'another string']]`

The following options can be set:

- **ndim**

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

  ```
  >>> import xlwings as xw
  >>> sheet = xw.Book().sheets[0]
  >>> sheet['A1'].value = [[1, 2], [3, 4]]
  >>> sheet['A1'].value
  1.0
  >>> sheet['A1'].options(ndim=1).value
  [1.0]
  >>> sheet['A1'].options(ndim=2).value
  [[1.0]]
  >>> sheet['A1:A2'].value
  [1.0 3.0]
  >>> sheet['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`:

  ```
  >>> sheet['A1'].value = 1
  >>> sheet['A1'].value
  1.0
  >>> sheet['A1'].options(numbers=int).value
  1
  ```

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

  Using this on UDFs looks like this:

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

Note: Excel always stores numbers internally as floats, which is the reason why the *int* converter rounds numbers first before turning them into integers. Otherwise it could happen that e.g. 5 might be returned as 4 in case it is represented as a floating point number that is slightly smaller than 5. Should you require Python's original *int* in your converter, use *raw int* instead.

- **dates**

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

  - Range:

    ```
    >>> import datetime as dt
    >>> sheet['A1'].options(dates=dt.date).value
    ```

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

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

  ```
  >>> my_date_handler = lambda year, month, day, **kwargs: "%04i-%02i-%02i" %
  (year, month, day)
  >>> sheet['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: `>>> sheet['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: `sheet['A1'].options(transpose=True).value = [1, 2, 3]`

  - UDFs:

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

- **expand**

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

```
>>> import xlwings as xw
>>> sheet = xw.Book().sheets[0]
>>> sheet['A1'].value = [[1,2], [3,4]]
>>> range1 = sheet['A1'].expand()
>>> range2 = sheet['A1'].options(expand='table')
>>> range1.value
[[1.0, 2.0], [3.0, 4.0]]
>>> range2.value
[[1.0, 2.0], [3.0, 4.0]]
>>> sheet['A3'].value = [5, 6]
>>> range1.value
[[1.0, 2.0], [3.0, 4.0]]
>>> range2.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
  ```

- **err_to_str** (new in v0.28.0)

  If `True`, will include cell errors such as `#N/A` as strings. By default, they will be converted to `None`.

- **formatter** (new in v0.28.1)

Note: You can't use formatters with Excel tables.

The `formatter` option accepts the name of a function. The function will be called after writing the values to Excel and allows you to easily style the range in a very flexible way. How it works is best shown with a little example:

```python
import pandas as pd
import xlwings as xw

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

def table(rng: xw.Range, df: pd.DataFrame):
    """This is the formatter function"""
    # Header
    rng[0, :].color = "#A9D08E"

    # Rows
    for ix, row in enumerate(rng.rows[1:]):
        if ix % 2 == 0:
            row.color = "#D0CECE"  # Even rows

    # Columns
    for ix, col in enumerate(df.columns):
        if "two" in col:
            rng[1:, ix].number_format = "0.0%"


df = pd.DataFrame(data={"one": [1, 2, 3, 4], "two": [5, 6, 7, 8]})
sheet["A1"].options(formatter=table, index=False).value = df
```

Running this code will format the DataFrame like this:

The formatter's signature is: `def myformatter(myrange, myvalues)` where `myrange` corresponds to the range where `myvalues` are written to. `myvalues` is simply what you assign to the `value` property in the last line of the example. Since we're using this with a DataFrame, it makes sense to name the argument accordingly and using type hints will help your editor with auto-completion. If you would use a nested list instead of a DataFrame, you would write something like this instead:

```
def table(rng: xw.Range, values: list[list]):  # Python >= 3.9
```

For Python <= 3.8, you'll need to capitalize `List` and import it like so: `from typing import List`.

16.2 Built-in Converters

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

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

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

16.2.1 Dictionary converter

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


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

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

16.2.2 Numpy array converter

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

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

Example:

```
>>> import numpy as np
>>> sheet = xw.Book().sheets[0]
>>> sheet['A1'].options(transpose=True).value = np.array([1, 2, 3])
>>> sheet['A1:A3'].options(np.array, ndim=2).value
array([[ 1.],
       [ 2.],
       [ 3.]])
```

16.2.3 Pandas Series converter

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

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

index: int or Boolean

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

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

header: Boolean

When reading, set it to `False` if Excel doesn't show either index or series names.

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

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

Example:

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

```
>>> sheet = xw.Book().sheets[0]
>>> s = sheet['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
```

16.2.4 Pandas DataFrame converter

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

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

index: int or Boolean

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

header: int or Boolean

> When reading, it expects the number of column headers shown in Excel.
>
> When writing, include or exclude the index and series names by setting it to `True` or `False`.

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

Example:

```
>>> sheet = xw.Book().sheets[0]
>>> df = sheet['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:
>>> sheet['A1'].value = df

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

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

```
@xw.func
@xw.arg('x', pd.DataFrame, header=2)
@xw.ret(index=False)
```

Chapter 16. Converters and Options

```
def myfunction(x):
    # x is a DataFrame, do something with it
    return x
```

16.2.5 xw.Range and 'raw' converters

Technically speaking, these are "no-converters".

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

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

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

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

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

>>> sheet['A1:B2'].options('raw').value  # or sheet['A1:B2'].raw_value
((1.0, 'text'), (pywintypes.datetime(2016, 2, 1, 0, 0, tzinfo=TimeZoneInfo(
→'GMT Standard Time', True)), None))
```

16.3 Custom Converter

Here are the steps to implement your own converter:

- Inherit from `xlwings.conversion.Converter`

- Implement both a `read_value` and `write_value` method as static- or classmethod:

    - In `read_value`, `value` is what the base converter returns: hence, if no `base` has been specified it arrives in the format of the default converter.

    - In `write_value`, `value` is the original object being written to Excel. It must be returned in the format that the base converter expects. Again, if no `base` has been specified, this is the default converter.

The `options` dictionary will contain all keyword arguments specified in the `options` method, e.g. when calling `myrange.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
    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:

```python
# Fire up a Workbook and create a sample DataFrame
sheet = xw.Book().sheets[0]
df = pd.DataFrame([[1.,10.],[2.,np.nan], [3., 30.]])
```

- Default converter for DataFrames:

```python
# Write
sheet['A1'].value = df

# Read
sheet['A1:C4'].options(pd.DataFrame).value
```

- DataFrameDropna converter:

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

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

- Register an alias (optional):

```
DataFrameDropna.register('df_dropna')

# Write
sheet['A12'].options('df_dropna', dropna=True).value = df

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

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

```
DataFrameDropna.register(pd.DataFrame)

# Write
sheet['A13'].options(dropna=True).value = df

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

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

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

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

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

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

DEBUGGING

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

- **RunPython**: When calling Python through `RunPython`, you can set a `mock_caller` to make it easy to switch back and forth between calling the function from Excel and Python.

- **UDFs**: For debugging User Defined Functions, xlwings offers a convenient debugging server

To begin with, Excel will show Python errors in a Message Box:
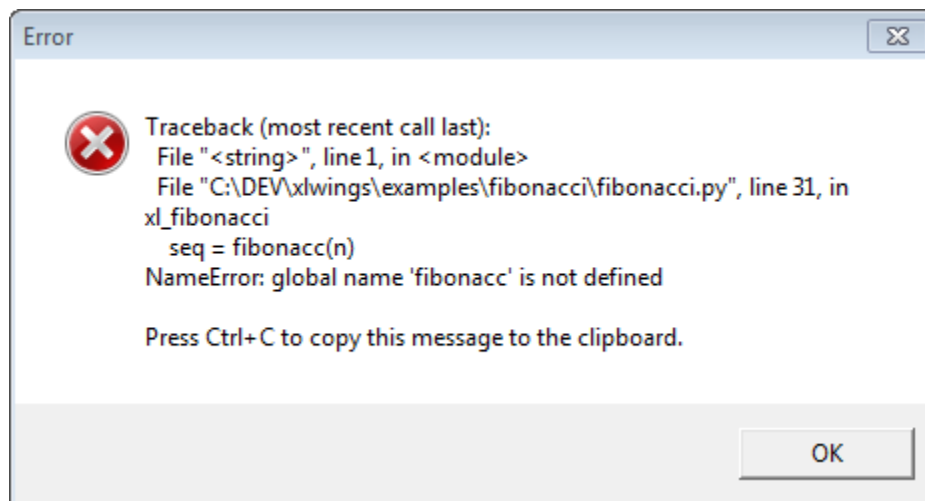
Note: On Mac, if the `import` of a module/package fails before `xlwings` is imported, the popup will not be shown and the StatusBar will not be reset. However, the error will still be logged in the log file (`/Users/<User>/Library/Containers/com.microsoft.Excel/Data/xlwings.log`).

17.1 RunPython

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

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

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

17.2 UDF debug server

Windows only: To debug UDFs, just check the `Debug UDFs` in the *Add-in & Settings*, at the top of the xlwings VBA module. Then add the following lines at the end of your Python source file and run it. Depending on which IDE you use, you might need to run the code in "debug" mode (e.g. in case you're using PyCharm or PyDev):

```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.

EIGHTEEN
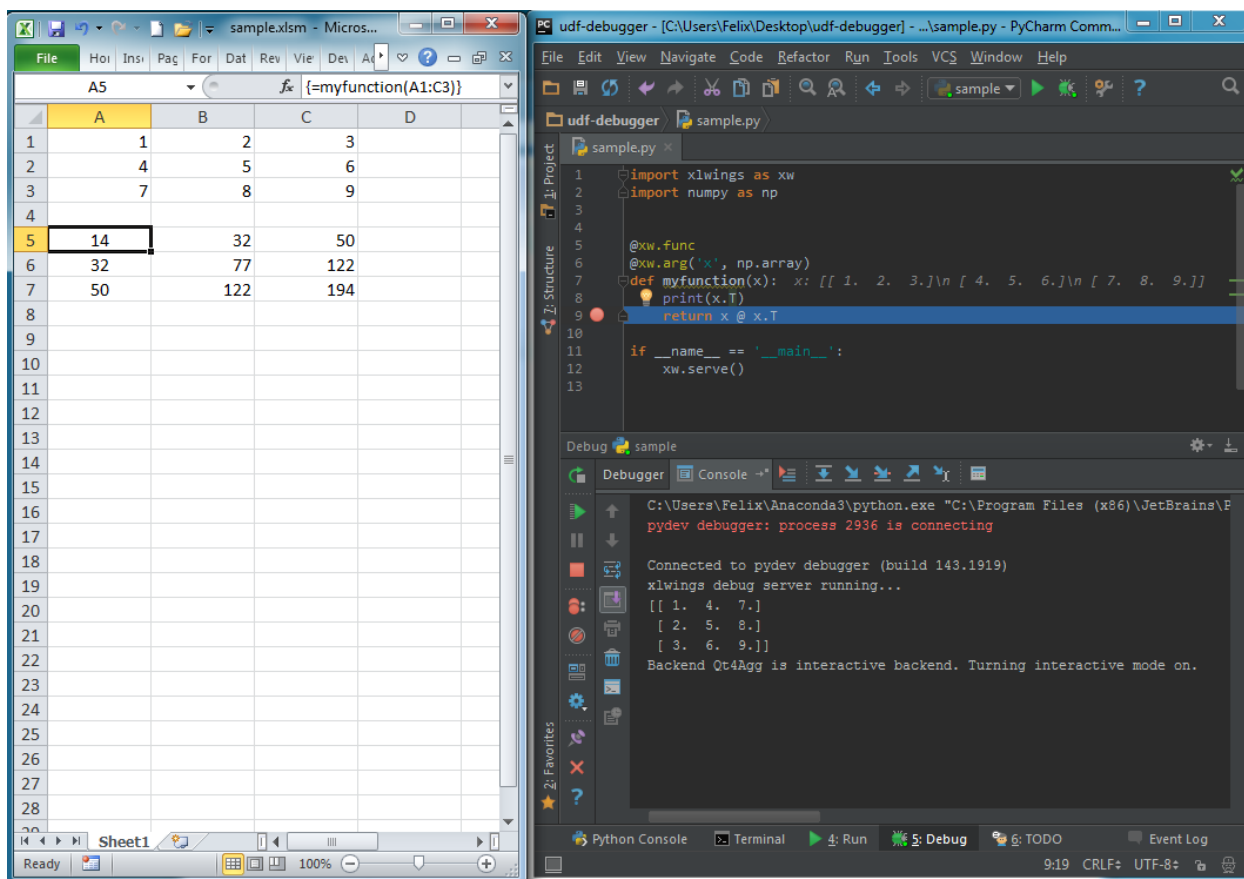
EXTENSIONS

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

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

18.1 In-Excel SQL

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

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

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

A16			⋮	×	✓	*fx*	=sql(A14,A1:D11,G1:H8)	

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

NINETEEN

CUSTOM ADD-INS

New in version 0.22.0.

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

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

19.1 Quickstart

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

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

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

- Double-click the Excel add-in to open it in Excel

- Add a new empty workbook (`Ctrl+N` on Windows or `Command+N` on macOS)

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

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

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

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

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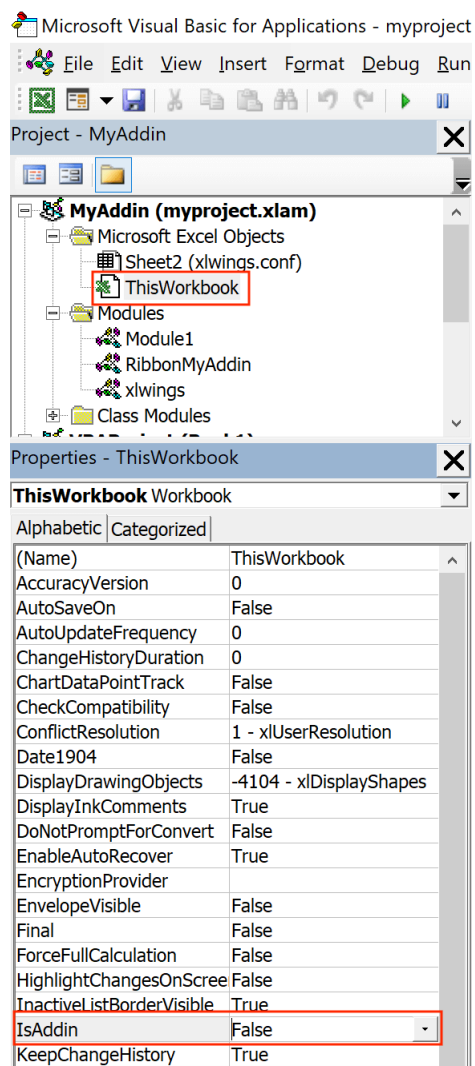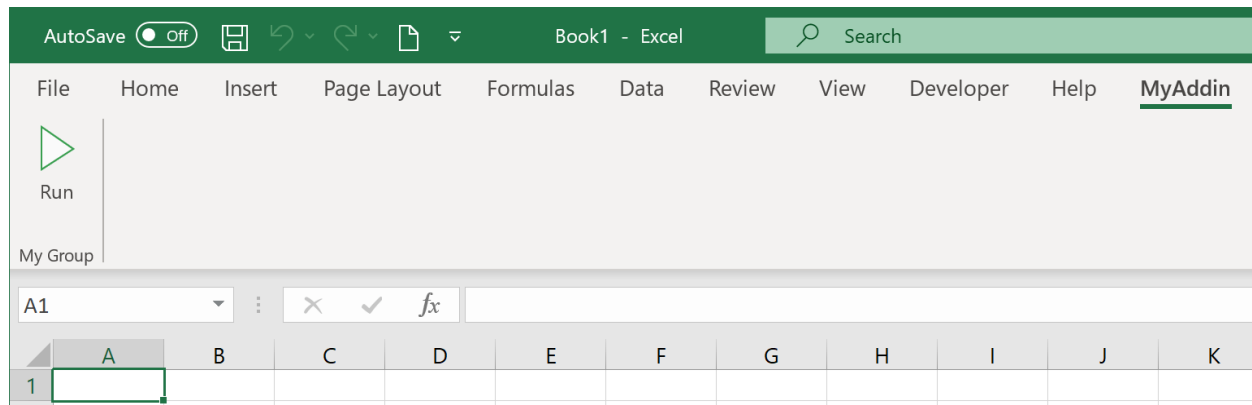
- Activate the sheet config by renaming it from `_myaddin.conf` to `myaddin.conf`

- Set your `Interpreter_Win/_Mac` or `Conda` settings (you may want to take them over from the xl-wings settings for now)

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

19.2 Changing the Ribbon menu

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

19.3 Importing UDFs

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

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

19.4 Configuration

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

19.5 Installation

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

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

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

19.6 Renaming your add-in

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

- In the `xlwings` VBA module, change `Public Const PROJECT_NAME As String = "myaddin"` to `Public Const PROJECT_NAME As String = "demo"`. You'll find this line at the top, right after the `Declare` statements.

- If you rely on the `myaddin.conf` sheet for your configuration, rename it to `demo.conf`

- Right-click the VBA project, select `MyAddin Properties...` and rename the `Project Name` from `MyAddin` to `Demo`.

- If you use the ribbon, you want to rename the `RibbonMyAddin` VBA module to `RibbonDemo`. To do this, select the module in the VBA editor, then rename it in the `Properties` window. If you don't see the Properties window, hit `F4`.

- Open the add-in in the Office RibbonX Editor (see above) and replace all occurrences of `MyAddin` with `Demo` in the XML code.

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

19.7 Deployment

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

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

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

```
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`.

THREADING AND MULTIPROCESSING

New in version 0.13.0.

20.1 Threading

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

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

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

num_threads = 4


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


q = Queue()

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

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

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

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()

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

q.join()
```

20.2 Multiprocessing

Note: Multiprocessing is only supported on Windows!

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

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

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


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

MISSING FEATURES

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

1) Most importantly, open an issue on GitHub. Adding functionality should be user driven, so only if you tell us about what you're missing, it's eventually going to find its way into the library. By the way, we also appreciate pull requests!

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

```
>>> sheet = xw.Book().sheets[0]
>>> sheet.api
# Windows (pywin32)
<win32com.gen_py.Microsoft Excel 16.0 Object Library._Worksheet instance at
↪0x2260624985352>
# macOS (appscript)
app(pid=2319).workbooks['Workbook1'].worksheets[1]
```

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).

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

```
# Windows
sheet['A1'].api.WrapText = True

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

TWENTYTWO

XLWINGS WITH OTHER OFFICE APPS

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

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

22.1 How To

1) As usual, write your Python function and import it into Excel (see *User Defined Functions (UDFs)*).

2) Press `Alt-F11` to get into the VBA editor, then right-click on the `xlwings_udfs` VBA module and select `Export File...`. Save the `xlwings_udfs.bas` file somewhere.

3) Switch into the other Office app, e.g. Microsoft Access and click again `Alt-F11` to get into the VBA editor. Right-click on the VBA Project and `Import File...`, then select the file that you exported in the previous step. Once imported, replace the app name in the first line to the one that you are using, i.e. `Microsoft Access` or `Microsoft Outlook` etc. so that the first line then reads: `#Const App = "Microsoft Access"`

4) Now import the standalone xlwings VBA module (`xlwings.bas`). You can find it in your xlwings installation folder. To know where that is, do:

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

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

22.2 Config

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

OVERVIEW PRO

xlwings PRO is source-available and dual-licensed under one of the following licenses:

- xlwings PRO License (commercial use requires a paid plan)

- PolyForm Noncommercial License 1.0.0 (non-commercial use is free)

23.1 PRO Features

- *Ultra Fast File Reader*: Similar to `pandas.read_excel()` but 5-25 times faster and you can leverage the convenient xlwings syntax. Works without an Excel installation and therefore on all platforms including Linux.

- *xlwings Server*: With xlwings Server, you don't need to install Python locally anymore. Instead, run it as a web app on a server. Works with Desktop Excel on Windows and macOS and with Google Sheet and Excel on the web. Runs on all platforms, including Linux, WSL and Docker.

- *Embedded code*: Store your Python source code directly in Excel for easy deployment.

- *xlwings Reports*: A template-based reporting framework, allowing business users to change the layout of the report without having to touch the Python code.

- *Markdown Formatting*: Support for Markdown formatting of text in cells and shapes like e.g., text boxes.

- *Permissioning*: Control which users can run which Python modules via xlwings.

Paid plans come with additional services like:

- *1-click Installer*: Easily build your own Python installer including all dependencies—your end users don't need to know anything about Python

- On-demand video course

- Direct Support

Check out the paid plans for more details!

23.2 License Key Activation

To use xlwings PRO, you need to install a license key on a Terminal/Command Prompt like so:

```
xlwings license update -k YOUR_LICENSE_KEY
```

Make sure to replace `LICENSE_KEY` with your personal key (see below). This will store the license key in your `xlwings.conf` file (see *User Config: Ribbon/Config File* for where this is on your system). Instead of running this command, you can also store the license key as an environment variable with the name `XLWINGS_LICENSE_KEY`.

License Key for Commercial Purpose:

- To try xlwings PRO for free in a commercial context, request a trial license key: https://www.xlwings.org/trial

- To use xlwings PRO in a commercial context beyond the trial, you need to enroll in a paid plan (they include additional services like support and the ability to create one-click installers): https://www.xlwings.org/pricing

xlwings PRO licenses are developer licenses, are verified offline (i.e., no telemetry/license server involved) and allow royalty-free deployments to unlimited internal end-users and servers for a hassle-free management. External end-users are included with the business plan. Deployments use deploy keys that don't expire but instead are bound to a specific version of xlwings.

License Key for non-commercial Purpose:

- To use xlwings PRO for free in a non-commercial context, use the following license key: `noncommercial` (Note that you need at least xlwings 0.26.0).

1-CLICK INSTALLER PRO

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

- Run an installer (one installer can power many different Excel workbooks)

- Use the Excel workbook as if it was a normal macro-enabled workbook

Advantages:

- **Zero-config**: The end user doesn't have to configure anything throughout the whole process.

- **No add-in required**: No installation of the xlwings add-in required.

- **Easy to update**: If you want to deploy an update of your Python code, it's often good enough to distribute a new version of your workbook.

- **No conflicts**: The installer doesn't touch any environment variables or registry keys and will therefore not conflict with any existing Python installations.

- **Deploy key**: The release command will add a deploy key as your LICENSE_KEY. A deploy key won't expire and end users won't need a paid subscription.

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

There is a video walkthrough at: https://www.youtube.com/watch?v=yw36VT_n1qg

24.1 Step 1: One-Click Installer

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

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

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

```
2 lines (1 sloc)   22 Bytes                          Raw   Blame          ✏   🗑
   1    xlwings[pro]==0.20.8
   2
```

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

---

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

On the next screen, click on `Draft a new release` (note, the very first time, you will see a green button called `Create a new release` instead):
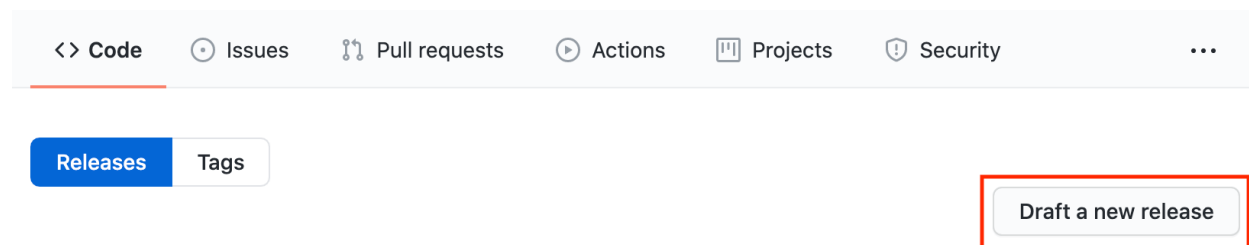
This will bring up the following screen, where you'll only have to fill in a `Tag version` (e.g., `1.0.0`), then click on the green button `Publish release`:

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

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

24.2 Step 2: Release Command (CLI)

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

- Make sure that you have enabled `Trust access to the VBA project object model` under `File > Options > Trust Center > Trust Center Settings > Macro Settings`. You only need to do this once and since this is a developer setting, your end users won't need to bother about this. This setting is needed so that xlwings can update the Excel file with the correct version of the VBA code.

- Run the installer from the previous step. This will not interfere with your existing Python installation as it won't touch your environment variables or registry. Instead, it will only write to the following folder: `%LOCALAPPDATA%\<installer-name>`.

- Make sure that your local version of xlwings corresponds to the version of xlwings in the `requirements.txt` from the installer. The easiest way to double-check this is to run `pip freeze` on a Command Prompt or Anaconda Prompt. If your local version of xlwings differs, install the same version as the installer uses via: `pip install xlwings==<version from installer>`.

To work with the release command, you should have your workbook in the `xlsm` format next to your Python code:

```
myworkbook.xlsm
mymodule_one.py
mypackage/
  mymodule_two.py
...
```

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

```
xlwings release
```

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

- `Name of your one-click installer?` *Type in the name of your one-click installer. If you want to use a different Python distribution (e.g., Anaconda), you can leave this empty (but you will need to update the xlwings.conf sheet with the Conda settings once the release command has been run).*

- `Embed your Python code?` [Y/n] *This will copy the Python code into the sheets of the Excel file.*

- `Hide the config sheet?` [Y/n] *This will hide the xlwings.conf sheet.*

- `Hide the sheets with the embedded Python code?` [Y/n] *If you embed your Python code, this will hide all sheets with a .py ending.*

- `Allow your tool to run without the xlwings add-in?` [Y/n] *This will remove the VBA reference to xlwings and copy in the xlwings VBA modules so that the end users don't need to have the xlwings add-in installed. Note that in this case, you will need to have your RunPython calls bound to a button as you can't use the Ribbon's Run main button anymore.*

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

24.3 Updating a Release

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

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

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

24.4 Embedded Code Explained

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

```
xlwings code embed
```

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

Note that you can have multiple Excel sheets and import them like normal Python files. Consider this example:
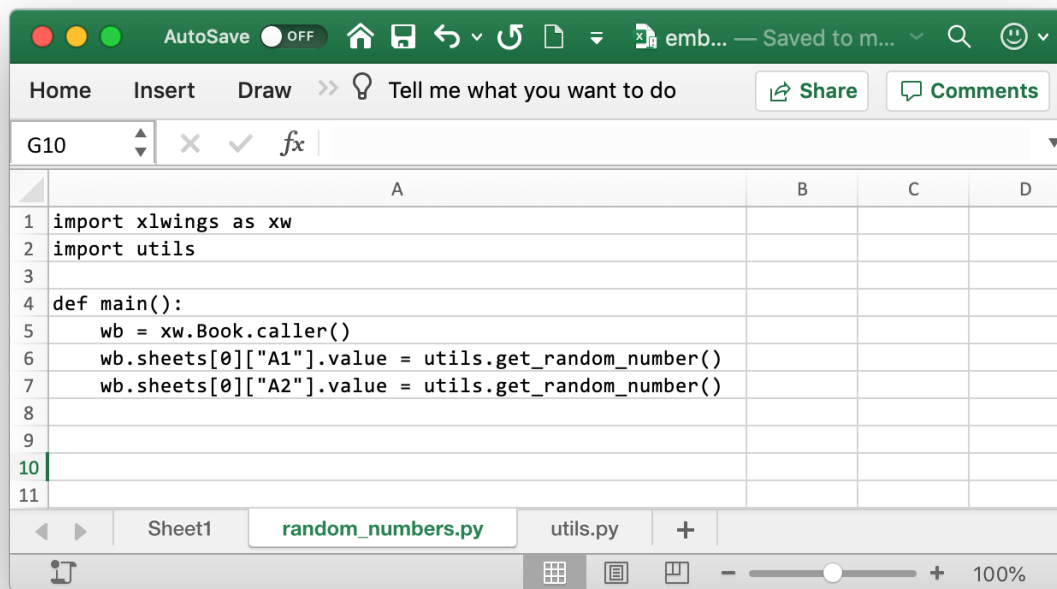
You can call the `main` function from VBA like so:

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

Note:

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

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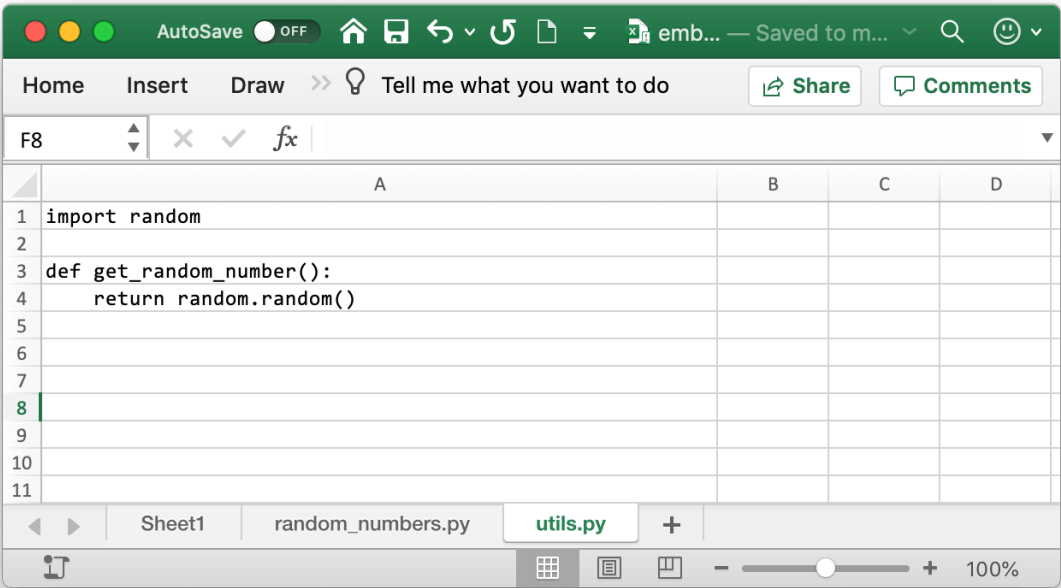
PERMISSIONING PRO

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

- **GET**: This is the simpler option as you only need to host a static JSON file that you can generate via the xlwings CLI. You can use any web server that is capable of serving static files (e.g., nginx) or use a free external service like GitHub pages. However, every permission change requires you to update the JSON file on the server.

- **POST**: This option relies on the web server to validate the incoming payload of the POST request. While this requires custom logic on your end, you are able to connect it with any internal system (such as a database or LDAP server) to dynamically decide whether a user should be able to run a specific Python module through xlwings.

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

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

25.1 Prerequisites

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

```
pip install requests
```

or via Conda:

```
conda install requests
```

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

25.2 Configuration

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

- To prevent end users from overwriting `xlwings.conf`, you'll need to make sure that the file is owned by the Administrator while giving end users read-only permissions.

- Add the following settings while replacing the `PERMISSION_CHECK_URL` and `PERMISSION_CHECK_METHOD` (POST or GET) with the appropriate value for your case. `PERMISSION_CHECK_AUTHORIZATION` is an optional setting that allows you to send a token with POST requests via the `Authorization` header:

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

25.3 GET request

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

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

```
cd myfolder
xlwings permission cwd
```

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

```
xlwings permission book
```

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

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

(continues on next page)

```
        "sha256": "355200bb9ae00fcec1d7b660e7dd95fb3dbf246a9db397a6daa2471458a8e6cb
↪",
        "machine_names": [
            "DESKTOP-QQ27RP3"
        ]
    }
  ]
}
```

All you need to do at this point is:

- Add additional machines names e.g., `"machine_names: [""DESKTOP-QQ27RP3",` `"DESKTOP-XY12AS2"]`. Alternatively, you can use the `"*"` wildcard if you want to allow the module to be used on all end user's computers. In case of the wildcard, it will still make sure that the file's content hasn't been changed by looking at its sha256 hash. xlwings uses `import socket;socket.gethostname()` as the machine name.

- Make this JSON file accessible via your web server and update the settings in the `xlwings.conf` file accordingly (see above).

25.4 POST request

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

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

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

- Return the HTTP status code 200 ("success") if the user is allowed to run the code of these modules

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

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

25.5 Implementation Details & Limitations

- Currently, `RunPython` and user-defined functions (UDFs) are supported. `RunFrozenPython` is not supported.

- Permissions checks are only done when the Python module is run via Excel/xlwings, it has no effect on Python code that is run from Python directly.

- `RunPython` won't allow you to run code that uses the `from x import y` syntax. Use `import x;x.y` instead.

- The answer of the permissioning server is cached for the duration of the Python session. For UDFs, this means until the functions are re-imported or the `Restart UDF Server` button is clicked or until Excel is restarted. The same is true if you run `RunPython` with the `Use UDF Server` option. By default, however, `RunPython` starts a new Python session every time, so it will contact the server whenever you call `RunPython`.

- Only top-level modules are checked, i.e. modules that are imported as UDFs or run via `RunPython` call. Any modules that are imported as dependencies of these modules are not checked.

- `RunPython` with external Python source files depends on logic in the VBA part of xlwings. UDFs and RunPython calls that use embedded code will only rely on Python to perform the permissioning check.

EXCEL FILE READER PRO

This feature requires at least v0.28.0.

xlwings PRO comes with an ultra fast Excel file reader. Compared with `pandas.read_excel()`, you should be able to see speedups anywhere between 5 to 25 times when reading a single sheet. The exact speed will depend on your content, file format, and Python version. The following Excel file formats are supported:

- `xlsx` / `xlsm` / `xlam`

- `xlsb`

- `xls`

Other advantages include:

- Support for named ranges

- Support for dynamic ranges via `myrange.expand()` or `myrange.options(expand="table")`, respectively.

- Support for converters so you can read in ranges not just as pandas DataFrames, but also as NumPy arrays, lists, scalar values, dictionaries, etc.

- You can read out cell errors like `#DIV/0!` or `#N/A` as strings instead of converting them all into `NaN`

Unlike the classic ("interactive") use of xlwings that requires Excel to be installed, reading a file doesn't depend on an installation of Excel and therefore works everywhere where Python runs. However, reading directly from a file requires the workbook to be saved before xlwings is able to pick up any changes.

26.1 Reading a specific range

To open a file in read mode, provide the `mode="r"` argument: `xw.Book("myfile.xlsx", mode="r")`. You usually want to use `Book` as a context manager so that the file is automatically closed and resources cleaned up once the code leaves the body of the `with` statement:

```python
import xlwings as xw

with xw.Book("myfile.xlsx", mode="r") as book:
    sheet1 = book.sheets[0]
    data = sheet1["A1:B2"].value
```

If you don't use the `with` statement, make sure to close the book manually via `book.close()`.

26.2 Reading an entire sheet

To read an entire sheet, use the `cells` property:

```python
with xw.Book("myfile.xlsx", mode="r") as book:
    sheet1 = book.sheets[0]
    data = sheet1.cells.value
```

26.3 Converters: DataFrames etc.

You can use the usual converters, for example to read in a range as a DataFrame:

```python
with xw.Book("myfile.xlsx", mode="r") as book:
    sheet1 = book.sheets[0]
    df = sheet1["A1:B2"].options("df").value
    # As usual, you can also provide more options
    df = sheet1["A1:B2"].options("df", index=False).value
```

For more details, see *Converters and Options*.

26.4 Named Ranges

Named ranges can be accessed like so:

```python
with xw.Book("myfile.xlsx", mode="r") as book:
    sheet1 = book.sheets[0]
    data = sheet1["myname"].value   # get values
    address = sheet1["myname"].address   # get address
```

Alternatively, you can also access them via the *Names* collection:

```python
with xw.Book("myfile.xlsx", mode="r") as book:
    for name in book.names:
        print(name.refers_to_range.value)
```

26.5 Dynamic Ranges

You can make use of the usual range expansion to read in a range of dynamic size:

```python
with xw.Book("myfile.xlsx", mode="r") as book:
    sheet1 = book.sheets[0]
    data = sheet1["A1"].expand().value
```

26.6 Cell errors

While xlwings reads in cell errors such as `#N/A` as `None` by default, you may want to read them in as strings if you're specifically looking for these by using the `err_to_str` option:

```python
with xw.Book("myfile.xlsx", mode="r") as book:
    sheet1 = book.sheets[0]
    data = sheet1["A1:B2"].option(err_to_str=True).value
```

26.7 Limitations

- The reader is currently only available via `pip install xlwings`. Installation via `conda` is not yet supported, but you can still use pip to install xlwings into a Conda environment!

- Date cells: Excel cells with a Date/Time are currently only converted to a `datetime` object in Python for `xlsx` file formats. For `xlsb` format, pandas has the same restriction though (it uses `pyxlsb` under the hood).

- Dynamic ranges: `myrange.expand()` is currently inefficient, so will slow down the reading considerably if the dynamic range is big.

- Named ranges: Named ranges with sheet scope are currently not shown with their proper name: E.g. `mybook.names[0].name` will show the name `mylocalname` instead of including the sheet name like so `Sheet1!mylocalname`. Along the same lines, the `names` property can only be accessed via `book` object, not via `sheet` object. Other defined names (formulas and constants) are currently not supported.

- Excel tables: Accessing data via table names isn't supported at the moment.

- Options: except for `err_to_str`, non-default options are currently inefficient and will slow down the read operation. This includes `dates`, `empty`, and `numbers`.

- Formulas: currently only the cell values are supported, but not the cell formulas.

- This is only a file reader, writing files is currently not supported.

XLWINGS REPORTS PRO

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

xlwings Reports supports all commonly required components:

- **Text**: Easily format your text via Markdown syntax.

- **Tables (dynamic)**: Write pandas DataFrames to Excel cells and Excel tables and format them dynamically based on the number of rows.

- **Charts**: Use your favorite charting engine: Excel charts, Matplotlib, or Plotly.

- **Images**: You can include both raster (e.g., png) or vector (e.g., svg) graphics, including dynamically generated ones, e.g., QR codes or plots.

- **Multi-column Layout**: Split your content up into e.g. a classic two column layout by using Frames.

- **Single Template**: Generate reports in various languages, for various funds etc. based on a single template.

- **PDF Report**: Generate PDF reports automatically and "print" the reports on PDFs in your corporate layout for pixel-perfect results including headers, footers, backgrounds and borderless graphics.

- **Easy Pre-processing**: Since everything is based on Python, you can connect with literally any data source and clean it with pandas or some other library.

- **Easy Post-processing**: Again, with Python you're just a few lines of code away from sending an email with the reports as attachment or uploading the reports to your web server, S3 bucket etc.

27.1 Quickstart

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

- mysheet.render_template(**data): replaces the placeholders in mysheet

- mybook.render_template(**data): replaces the placeholders in all sheets of mybook

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

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

```python
# report.py
from pathlib import Path

import pandas as pd
import xlwings as xw

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

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

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

Then create the following Excel file called mytemplate.xlsx:

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

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

```
python report.py
```

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

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

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

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

See also `render_templates` (API reference).

27.1.1 Render Books and Sheets

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

Running the following code:

```python
import xlwings as xw


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

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

Copies the template sheet first and then fills it in:

See also the *mysheet.render_template (API reference)* and *mybook.render_template (API reference)*.

New in version 0.22.0.

27.2 DataFrames

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

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

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

```python
import xlwings as xw
import pandas as pd

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


27.2.1 DataFrames Filters

noheader

Hide the column headers

Example:

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

header

Only return the header

Example:

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

sortasc

Sort in ascending order (indices are zero-based)

Example: sort by second, then by first column:

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

sortdesc

Sort in descending order (indices are zero-based)

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

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

columns

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

See also: `colslice`

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

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

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

mul, div, sum, sub

Apply an arithmetic operation (multiply, divide, sum, subtract) on a column (indices are zero-based)

Syntax:

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

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

Example: multiply the first column by 100:

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

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

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

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

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

maxrows

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

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

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

Syntax:

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

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

Examples:

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

aggsmall

Aggregate rows with values below a certain threshold (currently, only `sum` is supported as aggregation function)

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

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

Syntax:

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

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

Examples:

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

head

Only show the top n rows

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

Example:

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

tail

Only show the bottom n rows

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

Example:

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

rowslice

Slice the rows

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

Syntax:

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

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

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

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

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

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

colslice

Slice the columns

See also: `columns`

Syntax:

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

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

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

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

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

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

vmerge

Merge cells vertically for adjacent cells with the same value — can be used to represent hierarchies

---

Note: The `vmerge` filter does not work in Excel tables, as Excel tables don't support merged cells!

Note that the screenshot uses 4 *Frames* and the text is centered/vertically aligned in the template.

Syntax (arguments are optional):

```
{{ df | vmerge(col_index1, col_index2, ...) }}
```

Example (default): Hierarchical mode across all columns — this is helpful if the number of columns is dynamic. In hierarchical mode, cells are merged vertically in the first column (indices are zero-based) and cells in the next columns are merged only within the merged cells of the previous column:

```
{{ df | vmerge }}
```

Example: Hierarchical mode across the specified columns only:

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

Example: Independent mode: If you want to merge cells within columns independently of each other, use the filter multiple times. This sample merge cells vertically in the first two columns (indices are zero-based):

```
{{ df | vmerge(0) | vmerge(1) }}
```

formatter

Note: You can't use formatters with Excel tables.

The `formatter` filter accepts the name of a function. The function will be called after writing the values to Excel and allows you to easily style the range in a very flexible way:

```
{{ df | formatter("myformatter") }}
```

The formatter's signature is: `def myformatter(rng, df)` where `rng` corresponds to the range where the original DataFrame `df` is written to. Adding type hints (as shown in the example below) will help your editor with auto-completion.

Note: Within the reports framework, formatters need to be decorated with `xlwings.reports.formatter` (see example below)! This isn't necessary though when you use them as part of the standard xlwings API.

Let's run through the Quickstart example again, amended by a formatter.

Example:

```python
from pathlib import Path

import pandas as pd
import xlwings as xw
from xlwings.reports import formatter

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


@formatter
def table(rng: xw.Range, df: pd.DataFrame):
    """This is the formatter function"""
    # Header
    rng[0, :].color = "#A9D08E"

    # Rows
    for ix, row in enumerate(rng.rows[1:]):
        if ix % 2 == 0:
            row.color = "#D0CECE"  # Even rows

    # Columns
    for ix, col in enumerate(df.columns):
        if 'two' in col:
            rng[1:, ix].number_format = '0.0%'

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

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

27.3 Excel Tables

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

Running the following script:

	A	B			A	B
1	{{ df | formatter("table") }}			1	one	two
2				2	1	500.0%
3				3	2	600.0%
4				4	3	700.0%
5				5	4	800.0%
6				6		

	A	B	C
1	Title	**<frame>**	
2			
3	{{ df }}		
4			
5			
6	Some static text.		
7			

```
import pandas as pd

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

with xw.App(visible=True) as app:
    book = app.render_template('template.xlsx', 'output.xlsx', df=df)
```

Will produce the following report:

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

Note:

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

---

27.4 Excel Charts

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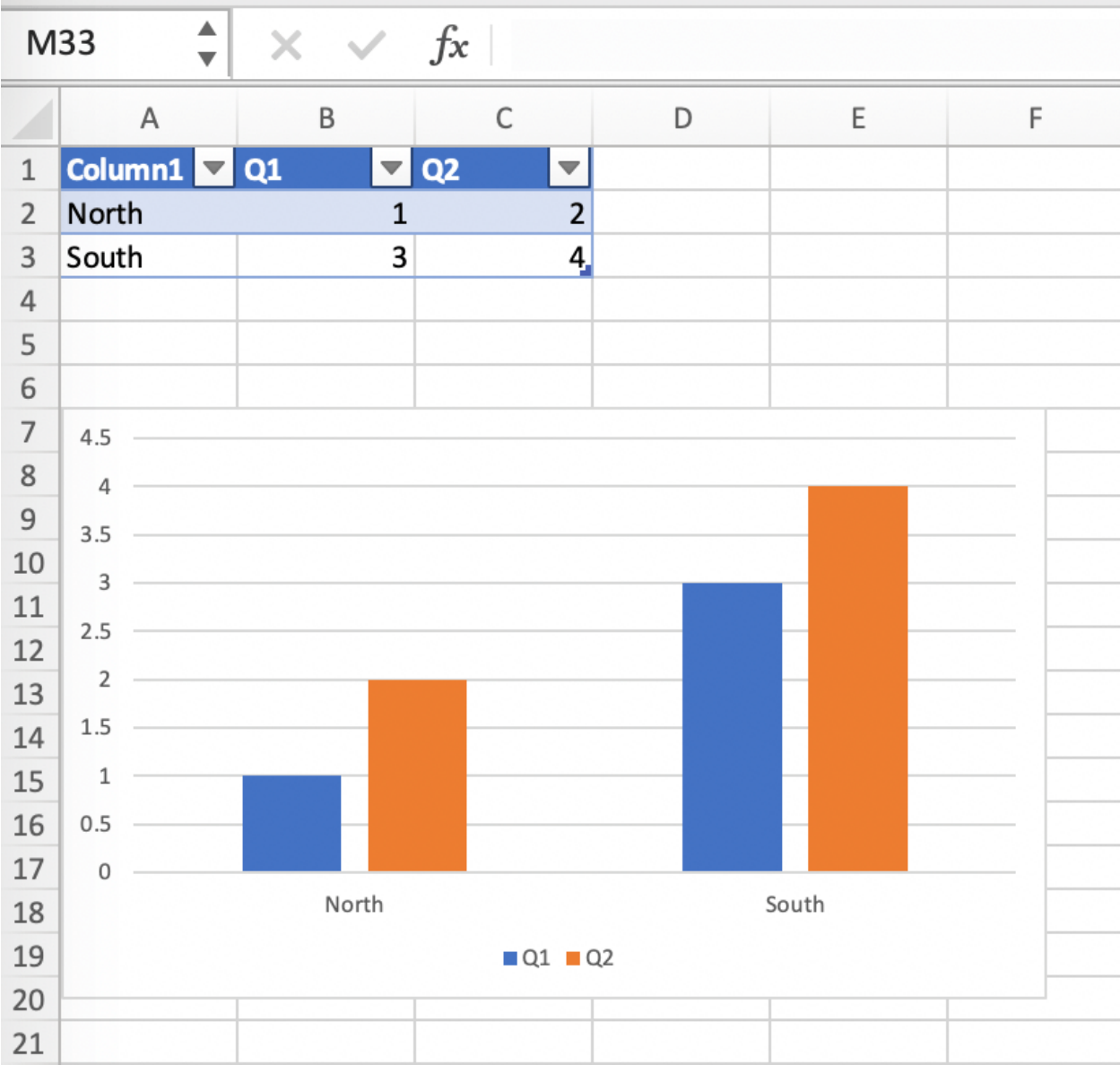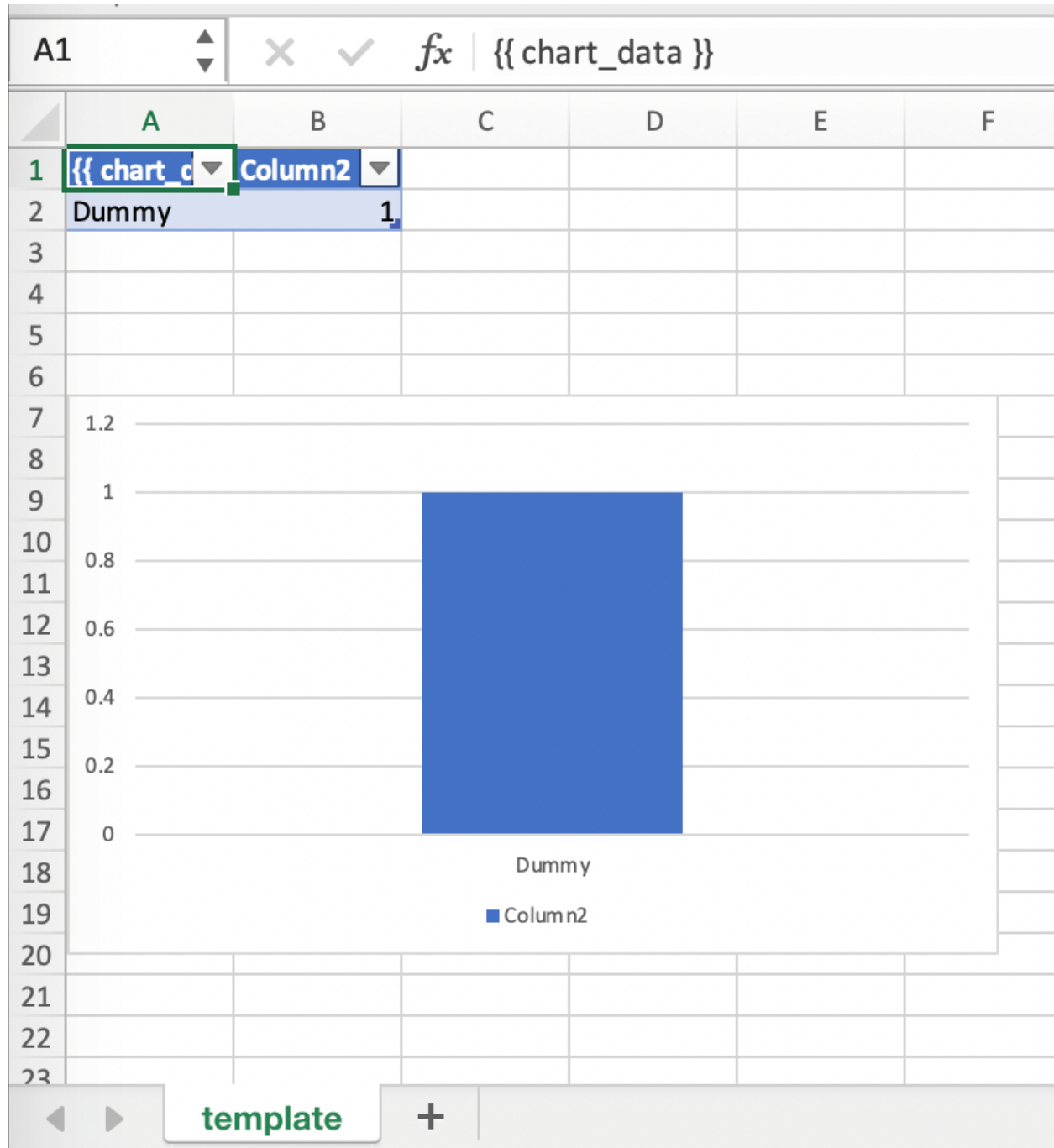
5. Assuming your file is called `mytemplate.xlsx` and your sheet `template` like on the previous screen-shot, you can run the following code:

```python
import xlwings as xw
import pandas as pd

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

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

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

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

27.5 Images

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

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

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

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

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

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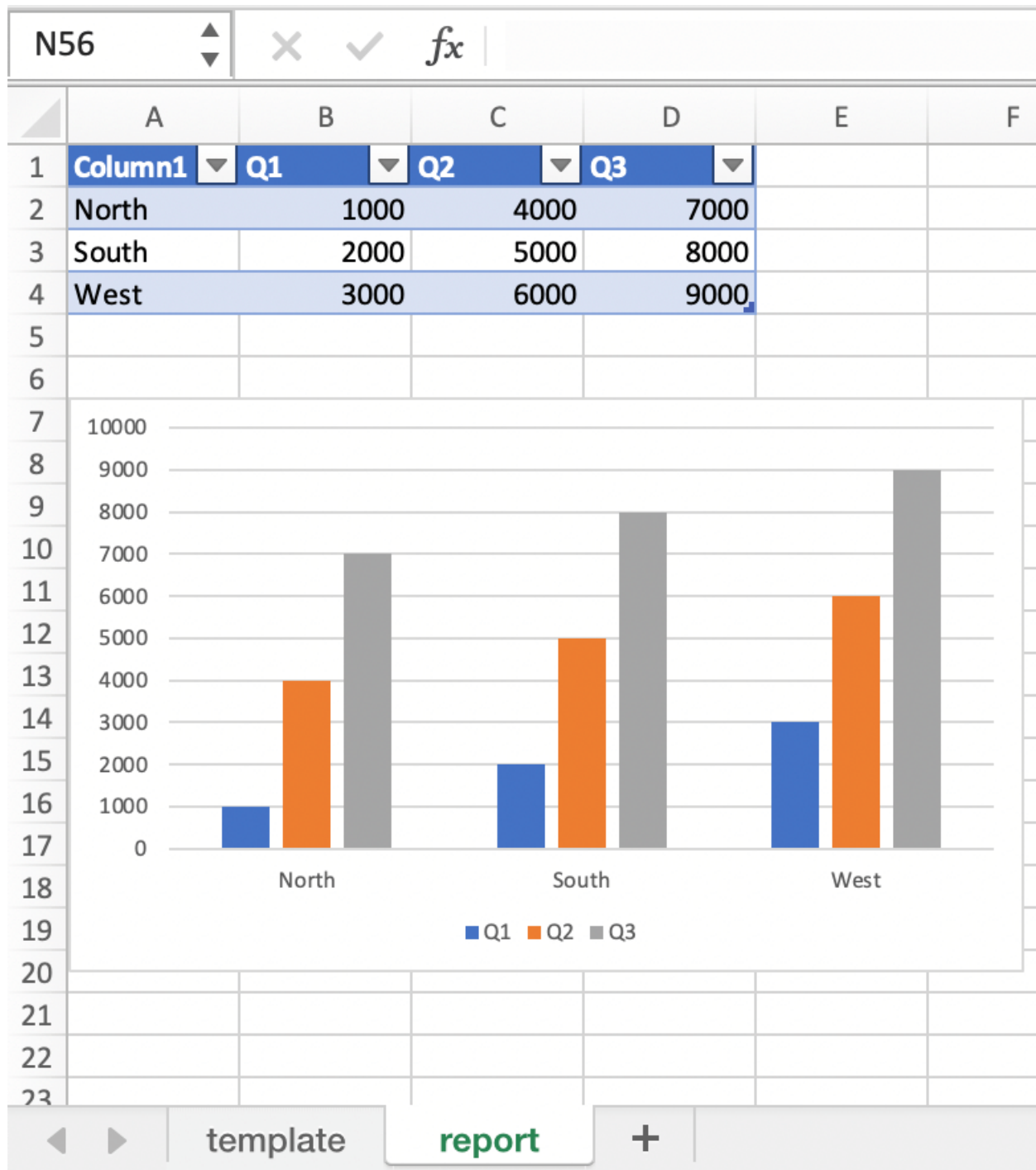
- **width**: Set the width in pixels (height will be scaled proportionally).

  Example:

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

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

  Example:

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

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

  Example:

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

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

  Example:

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

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

  See also: `left`

  Example:

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

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

  See also: `top`

  Example:

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

27.6 Matplotlib and Plotly Plots

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

27.6.1 Matplotlib

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

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

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

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

27.6.2 Plotly

Plotly works practically the same:

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

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

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

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

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

27.7 Text

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

27.7.1 Simple Text without Formatting

New in version 0.21.4.

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

```python
with xw.App(visible=True) as app:
    app.render_template('template.xlsx', 'output.xlsx', temperature=12.3)
```

This code turns this template:

into this report:

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

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

- **fontcolor**: Change the color of the whole (!) cell or shape. The primary purpose of this filter is to make white fonts visible in Excel. For most other colors, you can just change the color in Excel itself. Note that this filter changes the font of the whole cell or shape and only has an effect if there is just a

single placeholder—if you need to manipulate single words, use Markdown instead, see below. Black and white can be used as word, otherwise use a hex notation of your desired color.

Example:

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

27.7.2 Markdown Formatting

New in version 0.23.0.

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

```python
import xlwings as xw
from xlwings.reports import Markdown

mytext = """\
# Title

Text **bold** and *italic*

* A first bullet
* A second bullet

# {{ second_title }}

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

# The first sheet requires a shape as shown on the screenshot
sheet = xw.sheets.active
sheet.render_template(myplaceholder=Markdown(mytext),
                      second_title='Another Title')
```

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

Like this (this uses the default formatting):

For more details about Markdown, especially about how to change the styling, see *Markdown*.

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

	A	B	C	D	E	F
	Title Text **bold** and *italic* • A first bullet • A second bullet **Another Title** This paragraph has a line break.	**Title** Text **bold** and *italic* • A first bullet • A second bullet **Another Title** This paragraph has a line break.				
1	Another line.	Another line.				
2						

27.8 Date and Time

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

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

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

Example: The default formatting is `December 1, 2020`:

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

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

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

27.9 Number Format

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

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

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

27.10 Frames: Multi-column Layout

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

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

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

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

You can define and format DataFrames by formatting

- one header and

- one data row

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

Running the following code:

```python
import pandas as pd

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

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

with xw.App(visible=True) as app:
    book = app.render_template('my_template.xlsx',
                               'my_report.xlsx',
                               **data)
```

will generate this report:

27.11 PDF Layout

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

```python
import pandas as pd

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

with xw.App(visible=True) as app:
    book = app.render_template('template.xlsx',
                               'report.xlsx',
                               month_year = 'May 21',
                               summary_text = '...')
    book.to_pdf('report.pdf', layout='monthly_layout.pdf')
```

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

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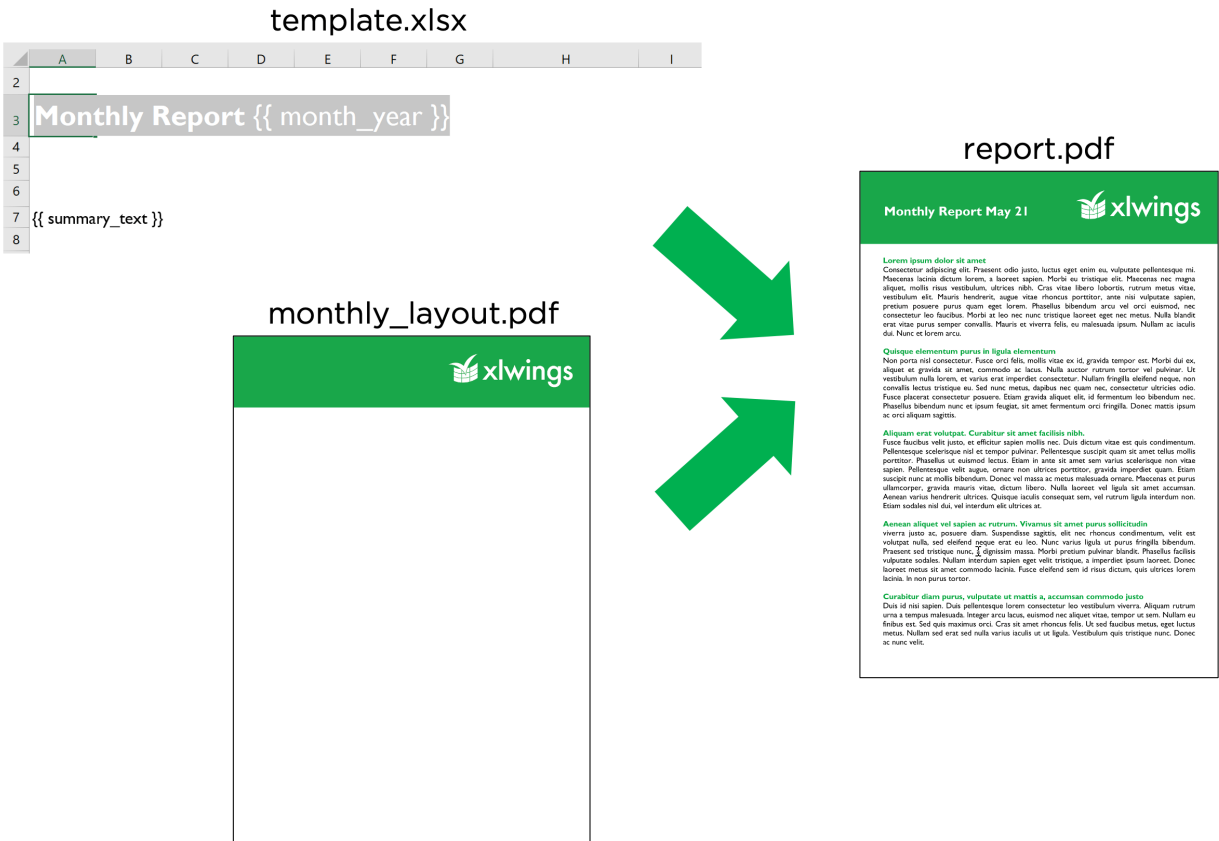
MARKDOWN FORMATTING PRO

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.reports 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
```

(continues on next page)

```
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:

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.reports import Markdown, MarkdownStyle

mytext = """\
# Title

Text **bold** and *italic*

* A first bullet
* A second bullet

# Another Title

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

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

# Styling
style = MarkdownStyle()
```

```python
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:

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 PRO*.

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

XLWINGS SERVER PRO

This feature requires at least v0.27.0.

Instead of installing Python on each end-user's machine, you can work with a server-based Python installation. It's essentially a web application, but uses your spreadsheet as the frontend instead of a web page in a browser. xlwings Server doesn't just work with the Desktop versions of Excel on Windows and macOS but additionally supports Google Sheets and Excel on the web for a full cloud experience. xlwings Server runs everywhere where Python runs, including Linux, Docker and WSL (Windows Subsystem for Linux). it can run on your local machine, as a (serverless) cloud service, or on an on-premise server.

Important: This feature currently only covers parts of the RunPython API (UDFs are not yet supported). See also *Limitations* and *Roadmap*.

29.1 Why is this useful?

Having to install a local installation of Python with the correct dependencies is the number one friction when using xlwings. Most excitingly though, xlwings Server adds support for the web-based spreadsheets: Google Sheets and Excel on the web.

To automate Office on the web, you have to use Office Scripts (i.e., TypeScript, a typed superset of JavaScript) and for Google Sheets, you have to use Apps Script (i.e., JavaScript). If you don't feel like learning JavaScript, xlwings allows you to write Python code instead. But even if you are comfortable with JavaScript, you are very limited in what you can do, as both Office Scripts and Apps Script are primarily designed to automate simple spreadsheet tasks such as inserting a new sheet or formatting cells rather than performing data-intensive tasks. They also make it very hard/impossible to use external JavaScript libraries and run in environments with minimal resources.

Note: From here on, when I refer to the **xlwings JavaScript module**, I mean either the xlwings Apps Script module if you use Google Sheets or the xlwings Office Scripts module if you use Excel on the web.

On the other hand, xlwings Server brings you these advantages:

- **Work with the whole Python ecosystem**: including pandas, machine learning libraries, database packages, web scraping, boto (for AWS S3), etc. This makes xlwings a great alternative for Power

Query, which isn't currently available for Excel on the web or Google Sheets.

- **Leverage your existing development workflow**: use your favorite IDE/editor (local or cloud-based) with full Git support, allowing you to easily track changes, collaborate and perform code reviews. You can also write unit tests using pytest.

- **Remain in control of your data and code**: except for the data you expose in Excel or Google Sheets, everything stays on your server. This can include database passwords and other sensitive info such as customer data. There's also no need to give the Python code to end-users: the whole business logic with your secret sauce is protected on your own infrastructure.

- **Choose the right machine for the job**: whether that means using a GPU, a ton of CPU cores, lots of memory, or a gigantic hard disc. As long as Python runs on it, you can go from serverless functions as offered by the big cloud vendors all the way to a self-managed Kubernetes cluster under your desk (see *Production Deployment*).

- **Headache-free deployment and maintenance**: there's only one location (usually a Linux server) where your Python code lives and you can automate the whole deployment process with continuous integration pipelines like GitHub actions etc.

- **Cross-platform**: xlwings Server works with Google Sheets, Excel on the web and the Desktop apps of Excel on Windows and macOS.

29.2 Prerequisites

Excel (via VBA)

- At least xlwings 0.27.0

- Either the xlwings add-in installed or a workbook that has been set up in standalone mode

Excel (via Office Scripts)

- At least xlwings 0.27.0

- You need the `Automate` tab enabled in order to access Office Scripts. Note that Office Scripts currently requires OneDrive for Business or SharePoint (it's not available on the free office.com), see also Office Scripts Requirements.

- The `fetch` command in Office Scripts must **not** be disabled by your Microsoft 365 administrator.

- Note that Office Scripts is available for Excel on the web and more recently also for Desktop Excel if you use Microsoft 365 (macOS and Windows), you may need to be on the beta channel though.

Google Sheets

- At least xlwings 0.27.0

- New sheets: no special requirements.

- Older sheets: make sure that Chrome V8 runtime is enabled under `Extensions > Apps Script > Project Settings > Enable Chrome V8 runtime`.

29.3 Introduction

xlwings Server consists of two parts:

- Backend: the Python part

- Frontend: the xlwings JavaScript module (for Google Sheets/Excel via Office Scripts) or the VBA code in the form of the add-in or standalone modules (Desktop Excel via VBA)

The backend exposes your Python functions by using a Python web framework. In more detail, you need to handle a POST request along these lines (the sample shows an excerpt that uses FastAPI as the web framework, but it works accordingly with any other web framework like Django or Flask):

```python
@app.post("/hello")
def hello(data: dict = Body):
    # Instantiate a Book object with the deserialized request body
    book = xw.Book(json=data)

    # Use xlwings as usual
    sheet = book.sheets[0]
    sheet["A1"].value = 'Hello xlwings!'

    # Pass the following back as the response
    return book.json()
```

- **For Desktop Excel, you can run the web server locally and call the respective function**

    - from VBA (requires the add-in installed) or

    - from Office Scripts

- For the cloud-based spreadsheets, you have to run this on a web server that can be reached from Google Sheets or Excel on the web, and you have to paste the xlwings JavaScript module into the respective editor. How this all works, will be shown in detail under *Cloud-based development with Gitpod*.

The next section shows you how you can play around with the xlwings Server on your local desktop before we'll dive into developing against the cloud-based spreadsheets.

29.4 Local Development with Desktop Excel

The easiest way to try things out is to run the web server locally against your Desktop version of Excel. We're going to use FastAPI as our web framework. While you can use any web framework you like, no quickstart command exists for these yet, so you'd have to set up the boilerplate yourself.

Start by running the following command on a Terminal/Command Prompt. Feel free to replace `demo` with another project name and make sure to run this command in the desired directory:

```
$ xlwings quickstart demo --fastapi
```

This creates a folder called `demo` in the current directory with the following files:

```
demo.xlsm
main.py
requirements.txt
```

I would recommend you to create a virtual or Conda environment where you install the dependencies via `pip install -r requirements.txt`. To run this server locally, run `python main.py` in your Terminal/Command Prompt or use your code editor/IDE's run button. You should see something along these lines:

```
$ python main.py
INFO:     Will watch for changes in these directories: ['/Users/fz/Dev/demo']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [36073] using watchgod
INFO:     Started server process [36075]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Your web server is now listening, so let's open `demo.xlsm`.

If you want to use VBA, press `Alt+F11` to open the VBA editor, and in `Module1`, place your cursor somewhere inside the following function:

```
Sub SampleRemoteCall()
    RunRemotePython "http://127.0.0.1:8000/hello"
End Sub
```

Then hit `F5` to run the function—you should see `Hello xlwings!` in cell A1 of the first sheet.

If, however, you want to use Office Scripts, you can basically start from an empty file (it can be `xlsx`, it doesn't have to be `xlsm`), and run *xlwings copy os* on the Terminal/Command Prompt/Anaconda Prompt. Then add a new Office Script and paste the code from the clipboard before clicking on *Run*.

To move this to production, you need to deploy the backend to a server, set up authentication, and point the URL to the production server, see *Production Deployment*.

The next sections, however, show you how you can make this work with Google Sheets and Excel on the web.

29.5 Cloud-based development with Gitpod

Using Gitpod is the easiest solution if you'd like to develop against either Google Sheets or Excel on the web.

If you want to have a development environment up and running in less than 5 minutes (even if you're new to web development), simply click the `Open in Gitpod` button to open a sample project in Gitpod (Gitpod is a cloud-based development environment with a generous free tier):

Opening the project in Gitpod will require you to sign in with your GitHub account. A few moments later, you should see an online version of VS Code. In the Terminal, it will ask you to paste the xlwings license key (get a free trial key if you want to try this out in a commercial context or use the `noncommercial` license key if your usage qualifies as noncommercial). Note that your browser will ask you for permission to paste. Once you confirm your license key by hitting `Enter`, the server will automatically start with everything properly configured. You can then open the `app` directory and look at the `main.py` file, where you'll see the `hello` function. This is the function we're going to call from Google Sheets/Excel on the web in just a moment. Let's now look at the `js` folder and open the file according to your platform:

Google Sheets

```
xlwings_google.js
```

Excel (via Office Scripts)

```
xlwings_excel.ts
```

Copy all the code, then switch to Google Sheets or Excel, respectively, and continue as follows:

Google Sheets

Click on `Extensions > Apps Script`. This will open a separate browser tab and open a file called `Code.gs` with a function stub. Replace this function stub with the copied code from `xlwings_google.js` and click on the `Save` icon. Then hit the `Run` button (the `hello` function should be automatically selected in the dropdown to the right of it). If you run this the very first time, Google Sheets will ask you for the permissions it needs. Once approved, the script will run the `hello` function and write `Hello xlwings!` into cell `A1`.

To add a button to a sheet to run this function, switch from the Apps Script editor back to Google Sheets, click on `Insert > Drawing` and draw a rounded rectangle. After hitting `Save and Close`, the rectangle will appear on the sheet. Select it so that you can click on the 3 dots on the top right of the shape. Select `Assign Script` and write `hello` in the text box, then hit `OK`.

Excel (via Office Scripts)

In the `Automate` tab, click on `New Script`. This opens a code editor pane on the right-hand side with a function stub. Replace this function stub with the copied code from `xlwings_excel.ts`. Make sure to click on `Save script` before clicking on `Run`: the script will run the `hello` function and write `Hello xlwings!` into cell `A1`.

To run this script from a button, click on the 3 dots in the Office Scripts pane (above the script), then select `+ Add button`.

Any changes you make to the `hello` function in `app/main.py` in Gitpod are automatically saved and reloaded by the web server and will be reflected the next time you run the script from Google Sheets or Excel on the web.

Note: While Excel on the web requires you to create a separate script with a function called `main` for each Python function, Google Sheets allows you to add multiple functions with any name.

Please note that clicking the Gitpod button gets you up and running quickly, but if you want to save your changes (i.e., commit them to Git), you should first fork the project on GitHub to your own account and open it by prepending `https://gitpod.io/#` to your GitHub URL instead of clicking the button (this works with GitLab and Bitbucket too). Or continue with the next section, which shows you how you can start a project from scratch on your local machine.

An alternative for Gitpod is GitHub Codespaces, but unlike Gitpod, GitHub Codespaces only works with GitHub.

29.6 Local Development with Google Sheets or Excel (via Office Scripts)

This section walks you through a local development workflow as an alternative to using Gitpod/GitHub Codespaces. What's making this a little harder than using a preconfigured online IDE like Gitpod is the fact that we need to expose our local web server to the internet for easy development (even if we use the Desktop version of Excel).

As before, we're going to use FastAPI as our web framework. While you can use any web framework you like, no quickstart command exists for these yet, so you'd have to set up the boilerplate yourself. Let's start with the server before turning our attention to the client side (i.e, Google Sheets or Excel on the web).

29.6.1 Part I: Backend

Start a new quickstart project by running the following command on a Terminal/Command Prompt. Feel free to replace demo with another project name and make sure to run this command in the desired directory:

```
$ xlwings quickstart demo --fastapi
```

This creates a folder called demo in the current directory with a few files:

```
main.py
demo.xlsm
requirements.txt
```

I would recommend you to create a virtual or Conda environment where you install the dependencies via `pip install -r requirements.txt`. In `app.py`, you'll find the FastAPI boilerplate code and in `main.py`, you'll find the `hello` function that is exposed under the `/hello` endpoint.

To run this server locally, run `python main.py` in your Terminal/Command Prompt or use your code editor/IDE's run button. You should see something along these lines:

```
$ python main.py
INFO:     Will watch for changes in these directories: ['/Users/fz/Dev/demo']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [36073] using watchgod
INFO:     Started server process [36075]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Your web server is now listening, however, to enable it to communicate with Google Sheets or Excel via Office Scripts, you need to expose the port used by your local server (port 8000 in your example) securely to the internet. There are many free and paid services available to help you do this. One of the more popular ones is ngrok whose free version will do the trick (for a list of ngrok alternatives, see Awesome Tunneling):

- ngrok Installation

- ngrok Tutorial

For the sake of this tutorial, let's assume you've installed ngrok, in which case you would run the following on your Terminal/Command Prompt to expose your local server to the public internet:

```
$ ngrok http 8000
```

Note that the number of the port (8000) has to correspond to the port that is configured on your local development server as specified at the bottom of `main.py`. ngrok will print something along these lines:

```
ngrok by @inconshreveable                                                    ↵
↪                       (Ctrl+C to quit)


Session Status                  online
Account                         name@domain.com (Plan: Free)
```

(continues on next page)

```
Version                         2.3.40
Region                          United States (us)
Web Interface                   http://127.0.0.1:4040
Forwarding                      http://xxxx-xxxx-xx-xx-xxx-xxxx-xxxx-xxxx-xxx.
↪ngrok.io -> http://localhost:8000
Forwarding                      https://xxxx-xxxx-xx-xx-xxx-xxxx-xxxx-xxxx-xxx.
↪ngrok.io -> http://localhost:8000
```

To configure the xlwings client in the next step, we'll need the `https` version of the Forwarding address that ngrok prints, i.e., `https://xxxx-xxxx-xx-xx-xxx-xxxx-xxxx-xxxx-xxx.ngrok.io`.

Note: When you're not actively developing, you should stop your ngrok session by hitting `Ctrl-C` in the Terminal/Command Prompt.

29.6.2 Part II: Frontend

Now it's time to switch to Google Sheets or Excel! To paste the xlwings JavaScript module, follow these 3 steps:

1. **Copy the xlwings JavaScript module**: On a Terminal/Command Prompt on your local machine, run the following command:

   #### Google Sheets

   ```
   $ xlwings copy gs
   ```

   #### Excel (via Office Scripts)

   ```
   $ xlwings copy os
   ```

   This will copy the correct xlwings JavaScript module to the clipboard so we can paste it in the next step.

2. **Paste the xlwings JavaScript module**

Google Sheets

Click on `Extensions > Apps Script`. This will open a separate browser tab and open a file called `Code.gs` with a function stub. Replace this function stub with the copied code from the previous step and click on the `Save` icon. Then hit the `Run` button (the `hello` function should be automatically selected in the dropdown to the right of it). If you run this the very first time, Google Sheets will ask you for the permissions it needs. Once approved, the script will run the `hello` function and write `Hello xlwings!` into cell A1.

To add a button to a sheet to run this function, switch from the Apps Script editor back to Google Sheets, click on `Insert > Drawing` and draw a rounded rectangle. After hitting `Save and Close`, the rectangle will appear on the sheet. Select it so that you can click on the 3 dots on the top right of the shape. Select `Assign Script` and write `hello` in the text box, then hit `OK`.

Excel (via Office Scripts)

In the `Automate` tab, click on `New Script`. This opens a code editor pane on the right-hand side with a function stub. Replace this function stub with the copied code from the previous step. Make sure to click on `Save script` before clicking on `Run`: the script will run the `hello` function and write `Hello xlwings!` into cell `A1`.

To run this script from a button, click on the 3 dots in the Office Scripts pane (above the script), then select `+ Add button`.

3. **Configuration**: The final step is to configure the xlwings JavaScript module properly, see the next section *Configuration*.

29.7 Configuration

xlwings can be configured in two ways:

- Via arguments in the `runPython` (via Apps Script / Office Scripts) or `RunRemotePython` (via VBA) function, respectively.

- Via `xlwings.conf` sheet (in this case, the keys are UPPER_CASE with underscore instead of camel-Case, see the screenshot below).

If you provide a value via config sheet and via function argument, the function argument wins. Let's see what the available settings are:

- `url` (required): This is the full URL of your function. In the above example under Local Development with Google Sheets or Excel on the web, this would be `https://xxxx-xxxx-xx-xx-xxx-xxxx-xxxx-xxxx-xxx.ngrok.io/hello`, i.e., the ngrok URL **with the /hello endpoint appended**.

- `auth` (optional): This is a shortcut to set the `Authorization` header. See the section about *Server Auth* for the options.

- `headers` (optional): A dictionary (VBA) or object literal (JS) with name/value pairs. If you set the `Authorization` header, the `auth` argument will be ignored.

- **exclude** (optional): By default, xlwings sends over the complete content of the whole workbook to the server. If you have sheets with big amounts of data, this can make the calls slow or you could even hit a timeout. If your backend doesn't need the content of certain sheets, you can exclude them from being sent over via this setting. Currently, you can only exclude entire sheets as comma-delimited string like so: `"Sheet1, Sheet2"`.

- **include** (optional): It's the counterpart to `exclude` and allows you to submit the names of the sheets that you want to send to the server. Like `exclude`, `include` accepts a comma-delimited string, e.g., `"Sheet1,Sheet2"`.

29.7.1 Configuration Examples: Function Arguments

Excel (via VBA)

Using only required arguments:

```
Sub Hello()
    RunRemotePython "http://127.0.0.1:8000/hello", auth:="xxxxxxxxxxxx"
End Sub
```

Additionally providing the `exclude` parameter to exclude the content of the `xlwings.conf` and `Sheet1` sheets:

```
Sub Hello()
    RunRemotePython "http://127.0.0.1:8000/hello", auth:="xxxxxxxxxxxx",␣
→exclude:="xlwings.conf, Sheet1"
End Sub
```

Excel (via Office Scripts)

Using only required arguments:

```
async function main(workbook: ExcelScript.Workbook) {
  await runPython(
    workbook,
    "https://xxxx-xxxx-xx-xx-xxx-xxxx-xxxx-xxxx-xxx.ngrok.io/hello",
    { auth: "xxxxxxxxxxxx" }
  );
}
```

Additionally providing the `exclude` parameter to exclude the content of the `xlwings.conf` and `Sheet1` sheets as well as a custom header:

```
async function main(workbook: ExcelScript.Workbook) {
  await runPython(
    workbook,
```

```
    "https://xxxx-xxxx-xx-xx-xxx-xxxx-xxxx-xxxx-xxx.ngrok.io/hello",
    {
      auth: "xxxxxxxxxxx",
      exclude: "xlwings.conf, Sheet1",
      headers: { MyHeader: "my value" },
    }
  );
}
```

Google Sheets

Using only required arguments:

```
function hello() {
  runPython("https://xxxx-xxxx-xx-xx-xxx-xxxx-xxxx-xxxx-xxx.ngrok.io/hello", {
    auth: "xxxxxxxxxxx",
  });
}
```

Additionally providing the `exclude` parameter to exclude the content of the `xlwings.conf` and `Sheet1` sheets as well as a custom header:

```
function hello() {
  runPython("https://xxxx-xxxx-xx-xx-xxx-xxxx-xxxx-xxxx-xxx.ngrok.io/hello", {
    auth: "xxxxxxxxxxx",
    exclude: "xlwings.conf, Sheet1",
    headers: { MyHeader: "my value" },
  });
}
```

29.7.2 Configuration Examples: xlwings.conf sheet

Create a sheet called `xlwings.conf` and fill in key/value pairs like so:

```
  | A       | B                |
  ---------------------------------
1 | AUTH    | xxxxxxxxxxx      |
2 | EXCLUDE | Sheet1,xlwings.conf|
```

29.8 Production Deployment

The xlwings web server can be built with any web framework and can therefore be deployed using any solution capable of running a Python backend or function. Here is a list for inspiration (non-exhaustive):

- **Fully-managed services**: Heroku, Render, Fly.io, etc.

- **Interactive environments**: PythonAnywhere, Anvil, etc.

- **Serverless functions**: AWS Lambda, Azure Functions, Google Cloud Functions, Vercel, etc.

- **Virtual Machines**: DigitalOcean, vultr, Linode, AWS EC2, Microsoft Azure VM, Google Cloud Compute Engine, etc.

- **Corporate servers**: Anything will work (including Kubernetes) as long as the respective endpoints can be accessed from your spreadsheet app.

Important: For production deployments, make sure to set up authentication, see *Server Auth*.

29.9 Triggers

Google Sheets

For Google Sheets, you can take advantage of the integrated Triggers (accessible from the menu on the left-hand side of the Apps Script editor). You can trigger your xlwings functions on a schedule or by an event, such as opening or editing a sheet.

Excel (via Office Scripts)

Normally, you would use Power Automate to achieve similar things as with Google Sheets Triggers, but unfortunately, Power Automate can't run Office Scripts that contain a `fetch` command like xlwings does, so for the time being, you can only trigger xlwings calls manually on Excel on the web. Alternatively, you can open your Excel file with Google Sheets and leverage the Triggers that Google Sheets offers. This, however, requires you to store your Excel file on Google Drive.

29.10 Limitations

- Currently, only a subset of the xlwings API is covered, mainly the Range and Sheet classes with a focus on reading and writing values and sending pictures (including Matplotlib plots). This, however, includes full support for type conversion including pandas DataFrames, NumPy arrays, datetime objects, etc.

- You are moving within the web's request/response cycle, meaning that values that you write to a range will only be written back to Google Sheets/Excel once the function call returns. Put differently, you'll

---

get the state of the sheets at the moment the call was initiated, but you can't read from a cell you've just written to until the next call.

- You will need to use the same xlwings version for the Python package and the JavaScript module, otherwise, the server will raise an error.

- Currently, custom functions (a.k.a. user-defined functions or UDFs) are not supported.

- For users with no experience in web development, this documentation may not be quite good enough just yet.

Platform-specific limitations:

Google Sheets

- Quotas for Google Services apply.

Excel on the web

- xlwings relies on the `fetch` command in Office Scripts that cannot be used via Power Automate and that can be disabled by your Microsoft 365 administrator.

- While Excel on the web feels generally slow, it seems to have an extreme lag depending on where in the world you open the browser with Excel on the web. For example, a hello world call takes ~4.5s if you open a browser in Amsterdam/Netherlands while it takes ~8.5s if you do it Buenos Aires/Argentina.

- Platform limits with Office Scripts apply.

29.11 Roadmap

- Complete the RunPython API by adding features that currently aren't supported yet, e.g., charts, shapes, tables, etc.

- Add support for UDFs/custom functions.

- Improve efficiency.

SERVER AUTH PRO

Authentication (and potentially authorization) is an important step in securing your xlwings Server app. On the server side, you can handle authentication

- within your app (via your web framework)

- outside of your app (via e.g. a reverse proxy such as nginx or oauth2-proxy that sits in front of your app)

Furthermore, you can use different authentication techniques such as HTTP Basic Auth or Bearer tokens in the form of API keys or OAuth2 access tokens.

On the client side, you set the `Authorization` header when you make a request from Excel or Google Sheets to your xlwings backend. To set the `Authorization` header, xlwings offers the `auth` parameter:

Excel (via VBA)

```
Sub Main()
    RunRemotePython "url", auth:="..."
End Sub
```

Excel (via Office Scripts)

```
async function main(workbook: ExcelScript.Workbook) {
  await runPython(workbook, "url", {
    auth: "...",
  });
}
```

Google Sheets

```
function main() {
  runPython("url", {
    auth: "...",
  });
}
```

Your backend will then have to validate the Authorization header. Let's get started with the simplest implementation of an API key before looking at HTTP Basic Auth and more advanced options like Azure AD and Google access tokens (for Google Sheets).

30.1 API Key

Generate a secure random string, for example by running the following from a Terminal/Command Prompt:

```
python -c "import secrets; print(secrets.token_hex(32))"
```

Provide this value as your `auth` argument in the `RunRemotePython` or `runPython` call and validate it on your backend.

For a sample backend implementation, see:

https://github.com/xlwings/xlwings-server-helloworld-fastapi

30.2 HTTP Basic Auth

Basic auth is a simple and popular method that sends the username and password via the Authorization header. Reverse proxies such as nginx allow you to easily protect your app with HTTP Basic Auth but you can also handle it directly in your app.

With your username and password, run the following Python script to get the value that you need to provide for `auth`:

```python
import base64
username = "myusername"
password = "mypassword"
print("Basic " + base64.b64encode(f"{username}:{password}".encode()).decode())
```

In this case, you'd provide `"Basic bXl1c2VybmFtZTpteXBhc3N3b3Jk"` as your `auth` argument.

- To validate HTTP Basic Auth with FastAPI, see: https://fastapi.tiangolo.com/advanced/security/http-basic-auth/

- If you use ngrok, there's an easy way to protect the exposed URL via Basic auth:

```
ngrok http 8000 -auth='myusername:mypassword'
```

> **Warning:** ngrok HTTP Basic auth will NOT work with Excel via Office Scripts as it doesn't support CORS. It's, however, an easy method for protecting your app during development if you use xlwings via VBA or Google Sheets.

30.3 Azure AD for Excel VBA

New in version 0.28.6:

> **Note:** Azure AD authentication is only available for Desktop Excel via VBA.

Azure Active Directory (Azure AD) is Microsoft's enterprise identity service. If you're using the xlwings add-in or VBA standalone module, xlwings allows you to comfortably log in users on their desktops, allowing you to securely validate their identity on the server and optionally implement role-base access control (RBAC).

Download `xlwings.exe`, the standalone xlwings CLI, from the GitHub Release page and place it in a specific folder, e.g., under `C:\Program and Files\xlwings\xlwings.exe` or `%LOCALAPPDATA%\xlwings\xlwings.exe`.

Now you can call the following function in VBA:

```
Sub Main()
  RunRemotePython "url", _
  auth:="Bearer " & GetAzureAdAccessToken( _
    tenantId:="...", _
    clientId:="...", _
    scopes:="...", _
    port:="...", _
    username:="...", _
    cliPath:="C:\Program and Files\xlwings\xlwings.exe" _
  )
End Sub
```

`port` and `username` are optional:

- Use `port` if the randomly assigned default port causes issues
- Use `username` if the user is logged in with multiple Microsoft accounts

> **Note:** Instead of relying on `xlwings.exe`, you could also use a normal Python installation with `xlwings` and `msal` installed. In this case, simply leave away the `cliPath` argument.

You can also use the `xlwings.conf` file or `xlwings.conf` sheet for configuration. In this case, the settings are the following:

```
AZUREAD_TENANT_ID
AZUREAD_CLIENT_ID
AZUREAD_SCOPES
AZUREAD_USERNAME
AZUREAD_PORT
CLI_PATH
```

Note that if you use the xlwings add-in rather than relying on the xlwings standalone VBA module, you will need to make sure that there's a reference set to xlwings in the VBA editor under `Tools > References`.

When you now call the `Main` function the very first time, a browser Window will open where the user needs to login to Azure AD. The acquired OAuth2 access token is then cached for 60-90 minutes. Once an access token has expired, a new one will be requested using the refresh token, i.e., without user intervention, but it will slow that that request.

For a complete walk-through on how to set up an app on Azure AD and how to validate the access token on the backend, see: https://github.com/xlwings/xlwings-server-auth-azuread

30.4 OAuth2 Access Token for Google Sheets

Google makes it easy to verify the logged-in user via OAuth2 access token. Simply provide the following as your `auth` argument:

```
ScriptApp.getOAuthToken()
```

To see how you can validate that token on the backend, see:

https://github.com/xlwings/xlwings-server-auth-google

WHAT'S NEW

31.1 v0.28.7 (Dec 27, 2022)

- Enhancement New CLI commands `xlwings copy vba` and `xlwings copy vba --addin`: They can help you to upgrade existing standalone projects and custom add-ins more easily (GH2129).

- Bug Fix PRO xlwings Server: Google Sheets was failing when cells contained a Date, caused by a recent Chromium V8 bug (GH2126).

- Bug Fix PRO xlwings Server: Writing `datetime` objects from Python to Google Sheets (with a time part not being zero) weren't formatting the cell properly as Date Time (GH2126).

31.2 v0.28.6 (Dec 15, 2022)

- Feature xlwings now allows to authenticate and authorize users via Azure AD in connection with the Ribbon add-in or VBA standalone module. This is useful in connection with a server component, such as xlwings Server, where the acquired access tokens can be validated, see *Server Auth* (GH2122).

- Enhancement PRO xlwings Server: added support for reading the Names collection via `mybook.names` and `mysheet.names` (GH2123).

- Feature The xlwings CLI (command-line interface) is now also available as a standalone executable for a limited set of uses cases. It can be downloaded from the GitHub Release page and can be useful to run `xlwings vba ...`, `xlwings auth ...`, and `xlwings addin ... -f` without having to install a full Python installation (GH2121).

- Breaking Change PRO: xlwings Server: `auth` replaces the `apiKey` argument in the `runPython` and `RunRemotePython` calls respectively. Technically it's only a deprecation, so `apiKey` still works for now (GH2104).

- Bug Fix PRO xlwings Server: Fixed an error with setting custom headers in VBA (GH2081).

31.3 v0.28.4 and v0.28.5 (Oct 29, 2022)

- Enhancement Added possibility to install the add-in globally for all users via `xlwings addin install -g` (GH2075).

- Enhancement Added `App.path` property (GH2074).

- Enhancement Build wheels for Python 3.11 (GH2071).

- Bug Fix 0.28.5 fixes an issue with the global add-in install (GH2076).

31.4 v0.28.3 (Oct 21, 2022)

- Bug Fix PRO xlwings File Reader: To be in line with the rest of the API, integers are now delivered as floats (GH2066).

- Bug Fix PRO xlwings File Reader: Fixed a bug that sometimes read in incorrect decimals with the legacy `xls` file formats (GH2062).

- Bug Fix PRO Fixed a bug introduced with 0.28.1 when `xlwings code embed` was run with the `--file` flag and a relative path (GH2061).

31.5 v0.28.2 (Oct 17, 2022)

- Breaking Change PRO xlwings File Reader: The reader was including Chartsheets etc. in `mybook.sheets`, which was inconsistent with the rest of the API. Accordingly, it now only shows Worksheets (GH2058).

- Bug Fix PRO xlwings File Reader: With `xlsb` formats, slightly unusual defined names caused the reader to fail (GH2057).

- Enhancement PRO xlwings Reports: the imports have been flattened. What previously was available via `xlwings.pro.reports` is now also available via `xlwings.reports` (GH2055).

- Enhancement PRO xlwings Reports: the registration of formatters for use with templates has been simplified by allowing you to use the `@formatter` decorator instead of having to register the function via `register_formatter(myfunc)` (GH2055).

31.6 v0.28.1 (Oct 10, 2022)

- Feature You can now use formatters to format the data you write to Excel or Google Sheets in a very flexible manner (see also *Default Converter*):

```python
import pandas as pd
import xlwings as xw

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

(continues on next page)

```python
def table(rng: xw.Range, df: pd.DataFrame):
    """This is the formatter function"""
    # Header
    rng[0, :].color = "#A9D08E"

    # Rows
    for ix, row in enumerate(rng.rows[1:]):
        if ix % 2 == 0:
            row.color = "#D0CECE"  # Even rows

    # Columns
    for ix, col in enumerate(df.columns):
        if "two" in col:
            rng[1:, ix].number_format = "0.0%"


df = pd.DataFrame(data={"one": [1, 2, 3, 4], "two": [5, 6, 7, 8]})
sheet["A1"].options(formatter=table, index=False).value = df
```


- Feature PRO Formatters are also available for xlwings Reports via filters: `{{ df | formatter("myformatter") }}`, see *DataFrames Filters*.

- Feature You can now export a sheet to an HTML page via `mysheet.to_html()`

- Feature New convenience property to get a list of the sheet names: `mybook.sheet_names`

- Enhancement PRO The Excel File Reader now supports the Names collection. I.e., you can now run code like this:

```python
with xw.Book("myfile.xlsx", mode="r") as book:
    for name in book.names:
        print(name.refers_to_range.value)
```

- Enhancement PRO Code embedding via `xlwings release` or `xlwings code embed` now allows you to work with Python packages, i.e., nested directories.

31.7 v0.28.0 (Oct 4, 2022)

- Feature PRO xlwings PRO adds an ultra fast file reader, allowing you to read Excel files much faster than via `pandas.read_excel()`:

```
with xw.Book("myfile.xlsx", mode="r") as book:
    sheet1 = book.sheets[0]
    df = sheet1["A1:B2"].options("df", index=False).value
```

For all the details, see *Excel File Reader*.

- Enhancement `Book` can now be used as context manager (i.e., with the `with` statement, see previous bullet point), which will close the book automatically when leaving the body of the `with` statement.

- Enhancement The new option `err_to_str` allows you to deliver cell errors like #N/A as strings instead of `None` (default): `xw.Book("mybook.xlsx").options(err_to_str=True).value`.

- Breaking Change PRO xlwings Server used to deliver cell errors as strings, which wasn't consistent with the rest of xlwings. This has now been fixed by delivering them as `None` by default. To get the previous behavior, use the `err_to_str` option, see the previous bullet point.

- Enhancement PRO The *Remote Interpreter* has been rebranded to *xlwings Server*.

31.8 v0.27.15 (Sep 16, 2022)

- Enhancement PRO Reports: Added new `vmerge` filter to vertically merge cells with the same values, for details, see *vmerge* (GH2020).

31.9 v0.27.14 (Aug 26, 2022)

- Enhancement Allow to install/remove the addin via `xlwings addin install` while Excel is running (GH1999).

31.10 v0.27.13 (Aug 22, 2022)

- Feature Add support for alerts: `myapp.alert("Hello World")`, see *myapp.alert()* for more details (GH756).

- Enhancement Handle Timedelta dtypes in pandas DataFrames and Series (GH1991).

- Enhancement PRO Remove the cryptography dependency from xlwings PRO (GH1992).

31.11 v0.27.12 (Aug 8, 2022)

- Enhancement PRO: xlwings Server: added support for named ranges via `mysheet["myname"]` or `mysheet.range("myname")` (GH1975).

- Enhancement PRO: xlwings Server: in addition to Google Sheets, `pictures.add()` is now also supported on Desktop Excel (Windows and macOS). This includes support for Matplotlib plots (GH1974).

- Enhancement Faster UDFs (GH1976).

- Bug Fix Made `myapp.range()` behave the same as `mysheet.range()` (GH1982).

- Bug Fix PRO: xlwings Server: cell errors were causing a bug with Desktop Excel (GH1968).

- Bug Fix PRO: xlwings Server: sending large payloads with Desktop Excel on macOS is now possible (GH1977).

31.12 v0.27.11 (Jul 6, 2022)

- Enhancement Added support for pandas `pd.NA` (GH1939).

- Bug Fix Empty cells in UDFs are now properly returned as `None` / `NaN` instead of an empty string (GH1947).

- Bug Fix Resolved an issue with OneDrive/SharePoint files that are unsynced locally (GH1946).

31.13 v0.27.10 (Jun 8, 2022)

- Bug Fix PRO This release fixes a `FileNotFound` error that could sometimes happen with embedded code (GH1931).

31.14 v0.27.9 (Jun 4, 2022)

- Bug Fix Fixes a bug on Windows that caused an Excel Zombie process with pywin32 > v301 (GH1929).

31.15 v0.27.8 (May 22, 2022)

- Enhancement Smarter shrinking of Excel tables when using `mytable.update(df)` as it doesn't delete rows below the table anymore (GH1908).

- Bug Fix Fixed a regression when `RunPyhon` was used with `Use UDF Server = True` (introduced in v0.26.2) (GH1912).

- Bug Fix PRO The `xlwings release` command would sometimes incorrectly show a version mismatch error (GH1918).

- Bug Fix PRO xlwings Reports now raises an explicit error when Jinja2 is missing (GH1637).

31.16 v0.27.7 (May 1, 2022)

- Feature PRO Google Sheets now support pictures via `mysheet.pictures.add()` incl. Matplotlib/Plotly (note that Excel on the web and Desktop Excel via xlwings Server are not yet supported). Also note that Google Sheets allows a maximum of 1 million pixels as calculated by (width in inches * dpi) * (height in inches * dpi), see also *Matplotlib & Plotly Charts* (GH1906).

- Breaking Change Matplotlib plots are now written to Excel/Google Sheets with a default of 200 dpi instead of 300 dpi. You can change this (and all other options that Matplotlib's `savefig()` and Plotly's `write_image()` offer via `sheet.pictures.add(image=myfigure, export_options={"bbox_inches": "tight", "dpi": 300})` (GH665, GH519).

31.17 v0.27.6 (Apr 11, 2022)

- Bug Fix macOS: Python modules on OneDrive Personal are now found again in the default setup even if they have been migrated to the new location (GH1891).

- Enhancement PRO xlwings Server now shows nicely formatted error messages across all platforms (GH1889).

31.18 v0.27.5 (Apr 1, 2022)

- Enhancement PRO xlwings Server: added support for setting the number format of a range via `myrange.number_format = "..."` (GH1887).

- Bug Fix PRO xlwings Server: Google Sheets/Excel on the web were formatting strings like `"1"` as date (GH1885).

31.19 v0.27.4 (Mar 29, 2022)

- Enhancement Further SharePoint enhancements on Windows, increasing the chance that `mybook.fullname` returns the proper local filepath (by taking into account the info in the registry) (GH1829).

- Enhancement The ribbon, i.e., the config, now allows you to uncheck the box `Add workbook to PYTHONPATH` to not automatically add the directory of your workbook to the PYTHONPATH. The respective config is called `ADD_WORKBOOK_TO_PYTHONPATH`. This can be helpful if you experience issues with OneDrive/SharePoint: uncheck this box and provide the path where your source file is manually via the PYTHONPATH setting (GH1873).

- Enhancement PRO Added support for `myrange.add_hyperlink()` with remote interpreter (GH1882).

- Enhancement PRO Added a new optional parameter `include` in connection with `runPython` (JS) and `RunRemotePython` (VBA), respectively. It's the counterpart to `exclude` and allows you to submit the names of the sheets that you want to send to the server. Like `exclude`, `include` accepts a comma-delimited string, e.g., "Sheet1,Sheet2" (GH1882).

- Enhancement PRO On Google Sheets, the xlwings JS module now automatically asks for the proper permission to allow authentication based on OAuth Token (GH1876).

31.20 v0.27.3 (Mar 18, 2022)

- Bug Fix PRO Fixes an issue with Date formatting on Google Sheets in case you're not using the U.S. locale (GH1866).

- Bug Fix PRO Fixes the truncating of ranges with xlwings Server in case the range was partly outside the used range (GH1822).

31.21 v0.27.2 (Mar 11, 2022)

- Bug Fix PRO Fixes an issue with xlwings Server that occurred on 64-bit versions of Excel.

31.22 v0.27.0 and v0.27.1 (Mar 8, 2022)

- Feature PRO This release adds support for xlwings Server to the Excel Desktop apps on both Windows and macOS. The new VBA function `RunRemotePython` is equivalent to `runPython` in the JavaScript modules of Google Sheets and Excel on the web, see *xlwings Server* (GH1841).

- Enhancement The xlwings package is now uploaded as wheel to PyPI in addition to the source format (GH1855).

- Enhancement The xlwings package is now compatible with Poetry (GH1265).

- Enhancement The add-in and the dll files are now code signed (GH1848).

- Breaking Change PRO The JavaScript modules (Google Sheet/Excel on the web) changed the parameters in `runPython`, see *xlwings Server* (GH1852).

- Breaking Change `xlwings vba edit` has been refactored and there is an additional command `xlwings vba import` to edit your VBA code outside of the VBA editor, e.g., in VS Code or any other editor, see *Command Line Client (CLI)* (GH1843).

- Breaking Change The `--unprotected` flag has been removed from the `xlwings addin install` command. You can still manually remove the password (`xlwings`) though (GH1850).

- Bug Fix PRO The `Markdown` class has been fixed in case the first line was empty (GH1856).

- Bug Fix PRO 0.27.1 fixes an issue with the version string in the new RunRemotePython VBA call (GH1859).

31.23 v0.26.3 (Feb 19, 2022)

- Feature If you still have to write VBA code, you can now use the new CLI command `xlwings vba edit`: this will export all the VBA modules locally so that you can edit them with any editor like e.g., VS Code. Every local change is synced back whenever you save the local file, see *Command Line Client (CLI)* (GH1839).

- Enhancement PRO The permissioning feature now allows you to send an Authorization header via the new `PERMISSION_CHECK_AUTHORIZATION` setting (GH1840).

31.24 v0.26.2 (Feb 10, 2022)

- Feature Added support for `myrange.clear_formats` and `mysheet.clear_formats` (GH1802).

- Feature Added support for `mychart.to_pdf()` and `myrange.to_pdf()` (GH1708).

- Feature PRO xlwings Server: added support for `mybook.selection` (GH1819).

- Enhancement The `quickstart` command now makes sure that the project name is a valid Python module name (GH1773).

- Enhancement The `to_pdf` method now accepts an additional parameter `quality` that defaults to `"standard"` but can be set to `"minimum"` for smaller PDFs (GH1697).

- Bug Fix Allow space in path to Python interpreter when using UDFs / UDF Server (GH974).

- Bug Fix A few issues were fixed in case your files are synced with OneDrive or SharePoint (GH1813 and GH1810).

- Bug Fix PRO Reports: fixed the `aggsmall` filter to work without the optional `min_rows` parameter (GH1824).

31.25 v0.26.0 and v0.26.1 (Feb 1, 2022)

- PRO Feature Added experimental support for Google Sheets and Excel on the web via a remote Python interpreter. For all the details, see *xlwings Server*.

- PRO Bug Fix 0.26.1 fixes an issue with the `xlwings copy gs` command.

- xlwings PRO is now free for noncommercial usage under the PolyForm Noncommercial License 1.0.0, see *xlwings PRO* for the details.

31.26 v0.25.3 (Dec 16, 2021)

- PRO Bug Fix The xlwings Reports filters `aggsmall` and `maxrows` don't fail with empty DataFrames anymore (GH1788).

31.27 v0.25.2 (Dec 3, 2021)

- PRO Enhancement xlwings Reports now ignores sheets whose name start with `##` for both rendering and printing to PDF (GH1779).
- PRO Enhancement The `aggsmall` filter in xlwings Reports now accepts a new parameter `min_rows` (GH1780).

31.28 v0.25.1 (Nov 21, 2021)

- Enhancement `mybook.save()` now supports the `password` parameter (GH1568).
- PRO Bug Fix xlwings Reports would sometimes cause a `Could not activate App instance` error (GH1764).
- PRO Enhancement xlwings now warns about expiring developer license keys 30 days before they expire (GH1758).

31.29 v0.25.0 (Oct 27, 2021)

- Bug Fix Finally, xlwings adds proper support for OneDrive, OneDrive for Business, and SharePoint. This means that the `quickstart` setup (Excel file and Python file in the same folder with the same name) works even if the files are stored on OneDrive/SharePoint—as long as they are being synced locally. It also makes `mybook.fullname` return the local file path instead of a URL. Sometimes, this requires editing the configuration, see: *OneDrive and SharePoint* for the details (GH1630).
- Feature The `update()` method of Excel tables has been moved from PRO to open source. You can now easily update an existing table in Excel with the data from a new pandas DataFrame without messing up any formulas that reference that table: `mytable.update(df)`, see: *Table.update()* (GH1751).
- PRO Breaking Change: Reports: `create_report()` is now deprecated in favor of `render_template()` that is available via `app`, `book` (new), and `sheet` objects, see: *xlwings Reports PRO* (GH1738).
- Bug Fix Running UDFs from other Office apps has been fixed (GH1729).
- Bug Fix Writing to a protected sheet or using an invalid sheet name etc. caused xlwings to hang instead of raising an Exception (GH1725).

31.30 v0.24.9 (Aug 26, 2021)

- Bug Fix Fixed a regression introduced with 0.24.8 that was causing an error with pandas DataFrames that have repeated column headers (GH1711).

31.31 v0.24.8 (Aug 25, 2021)

- Feature New methods *mychart.to_png()*, *myrange.to_png()* and *myrange.copy_picture()* (GH1707 and GH582).

- Enhancement You can now use the alias `'df'` to convert to a pandas DataFrame: `mysheet['A1:C3'].options('df').value` is equivalent to `import pandas as pd; mysheet['A1:C3'].options(pd.DataFrame).value` (GH1533).

- Enhancement Added `--dir` option to `xlwings addin install` to allow the installation of all files in a directory as add-ins (GH1702).

- Bug Fix Pandas DataFrames now properly work with `PeriodIndex` / `PeriodDtype` (GH1084).

- PRO Reports: If there's just one Frame, keep height of rows (GH1698).

31.32 v0.24.7 (Aug 5, 2021)

- PRO Breaking Change: Reports: Changed the order of the arguments of the arithmetic DataFrame filters: `sum`, `div`, `mul` and `div` to align them with the other filters. E.g., to multiply column 2 by 100, you now have to write your filter as `{{ df | mul(100, 2) }}` (GH1696).

- PRO Bug Fix Reports: Fixed an issue with images when pillow wasn't installed (GH1695).

31.33 v0.24.6 (Jul 31, 2021)

- Enhancement You can now also define the color of cells, shapes and font objects with a hex string instead of just an RGB tuple, e.g., `mysheet["A1"].color = "#efefef"` (GH1535).

- Enhancement When you print a workbook or sheet to a pdf, you can now automatically open the PDF document via the new `show` argument: `mybook.to_pdf(show=True)` (GH1683).

- Bug Fix: This release includes another round of fixing the cleanup actions of the App() context manager (GH1687).

- PRO Enhancement Reports: New filter `fontcolor`, allowing you to write text in black and turn it into e.g., white for the report. This gets around the issue that white text isn't visible in Excel on a white background: `{{ myplaceholder | fontcolor("white") }}`. Alternatively, you can also use a hex color (GH1692).

- PRO Bug Fix Positioning shapes wasn't always respecting the top/left filters (GH1687).

- PRO Bug Fix Fixed a bug with non-string headers when calling `table.update` (GH1687).

31.34 v0.24.5 (Jul 27, 2021)

- PRO Bug Fix Reports: Using the `header` filter in a Frame was causing rows to be inserted (GH1681).

31.35 v0.24.4 (Jul 26, 2021)

- Feature `myapp.properties` is a new context manager that allows you to easily change the app's properties temporarily. Once the code leaves the with block, the properties are changed back to their previous state (GH254). For example:

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

with app.properties(display_alerts=False):
    # Alerts are disabled until you leave the with block again
```

- Enhancement The app properties `myapp.enable_events` and `myapp.interactive` are now supported (GH254).

- Enhancement `mybook.to_pdf` now ignores sheet names that start with a `#`. This can be changed by setting the new parameter `exclude_start_string` (GH1667).

- Enhancement New method `mytable.resize()` (GH1662).

- Bug Fix The new App context manager introduced with v0.24.3 was sometimes causing an error on Windows during the cleanup actions (GH1668).

PRO **xlwings.pro.reports:**

- Breaking Change: DataFrame placeholders will now ignore the DataFrame's index. If you need the index, reset it via : `df.reset_index()` before passing the DataFrame to `create_report` or `render_template`. This was required as the same column index used in filters would point to seemingly different columns in Excel depending on whether the index was included or not. This also means that the `noindex` and `body` filters are no obsolete and have been removed (GH1676).

- Enhancement Dataframe filters now respect the order in which they are called and can be used multiple times (GH1675).

- Enhancement New filters: `format` (to apply f-string like formatting), `datetime` (to format datetime objects), `top` and `left` (to position graphics outside of the grid structure) `header`, `add`, `sub`, `mul`, `div` (to only return the header of a DataFrame or apply an arithmetic operation, respectively) (GH1666, GH1660, GH1677).

- Enhancement: `create_report` can now be accessed as method of the app object like so: `myapp.create_report` (GH1665).

- Bug Fix: Excel tables that had the Header Row unchecked were sometimes causing row shifts in the template (GH1663).

- Bug Fix: Rendering a template was sometimes causing the following error `PasteSpecial method of Range class failed` (GH1672).

31.36 v0.24.3 (Jul 15, 2021)

- Enhancement *xlwings.App()* can now be used as context manager, making sure that there are no zombie processes left over on Windows, even if you use a hidden instance and your code fails. It is therefore recommended to use it whenever you can, like so:

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

- Enhancement *mysheet.pictures.add* now accepts a new `anchor` argument that you can use as an alternative to `top/left` to position the picture by providing an anchor range object, e.g.: `mysheet.pictures.add(img, anchor=mysheet['A1'])` (GH1648).

- Bug Fix macOS: Plots are now sent to Excel in PDF format when you set `format='vector'` which is supporting transparency unlike the previously used eps format (GH1647).

- PRO Enhancement *mybook.to_pdf* now accepts a `layout` parameter so you can "print" your reports onto a PDF with your corporate layout including headers, footers and borderless graphics. See *PDF Layout*.

31.37 v0.24.2 (Jul 6, 2021)

- Feature Added very basic support for *mysheet.page_setup* and *myrange.note* (GH1551 and GH896).

- Enhancement DataFrames are now displayed in Excel tables with empty column names if the DataFrame doesn't have a column or index name. This effect is e.g. visible when using `xw.view()` (GH1643).

- Enhancement `mysheet.pictures.add()` now supports `format='vector'` which translates to `'svg'` on Windows and `'eps'` on macOS (GH1640).

- PRO Enhancement: The reports package now offers the additional DataFrame filters `rowslice` and `colslice`, see *xlwings Reports* (GH1645).

- PRO Bug Fix: Bug fix with handling Excel tables without headers.

Breaking Change

- PRO Enhancement: `<frame>` markers now have to be defined as cell notes in the first row, see *Frames: Multi-column Layout*. This has the advantage that the Layout view corresponds to the print view (GH1641). Also, the print area is now preserved even if you use Frames.

31.38 v0.24.1 (Jun 27, 2021)

- PRO Enhancement: The reports package now offers the additional DataFrame filters `head` and `tail`, see *xlwings Reports* (GH1633).

31.39 v0.24.0 (Jun 25, 2021)

- Enhancement `pictures.add()` now accepts every picture format (including vector-based formats) that your Excel version supports. For example, on Windows you can use the `svg` format (only supported with Excel that comes with Microsoft 365) and on macOS, you can use `eps` (GH1624).

- [Enhancements] Support for Plotly images was moved from PRO to the Open Source version, i.e. you can now provide a Plotly image directly to `pictures.add()`.

- Enhancement Matplotlib and Plotly plots can now be sent to Excel in a vector-based format by providing the `format` argument, e.g. `svg` on Windows or `eps` on macOS.

- Enhancement Removed dependency on pillow/PIL to properly size images via `pictures.add()`.

- Bug Fix Various fixes with scaling and positioning images via `pictures.add()` (GH1491).

- Feature New methods *mypicture.lock_aspect_ratio* and *myapp.cut_copy_mode* (GH1622 and GH1625).

- PRO Feature: Reports: DataFrames and Images are now offering various filters to influence the behavior of how DataFrames and Images are displayed, giving the template designer the ability to change a lot of things that previously had to be taken care of by the Python developer. For example, to hide a DataFrame's index, you can now do `{{ df | noindex }}` or to scale the image to double its size, you can do `{{ img | scale(2) }}`. You'll find all available filters under *xlwings Reports* (GH1602).

Breaking Change

- Enhancement: When using `pictures.add()`, pictures arrive now in Excel in the same size as if you would manually add them via the Excel UI and setting width/height now behaves consistently during initial adding and resizing. Consequently, you may have to fix your image sizes when you upgrade. (GH1491).

- PRO The default MarkdownStyle removed the empty space after a h1 heading. You can always reintroduce it by applying a custom style (GH1628).

31.40 v0.23.4 (Jun 15, 2021)

- Bug Fix Windows: Fixed the ImportUDFs function in the VBA standalone module (GH1601).

- Bug Fix Fixed configuration hierarchy: if you have a setting with an empty value in the `xlwings.conf` sheet, it will not be overridden by the same key in the directory or user config file anymore. If you wanted it to be overridden, you'd have to get the key out of the "xlwings.conf" sheet (GH1617).

- PRO Feature Added the ability to block the execution of Python modules based on the file hash and/or machine name (GH1586), see *Permissioning PRO*.

- PRO Feature Added the `xlwings release` command for an easy release management in connection with the one-click installer, see *1-click installer PRO*. (GH1429).

31.41 v0.23.3 (May 17, 2021)

- Bug Fix Windows: UDFs returning a `pandas.NaT` were causing a `#VALUE!` error (GH1590).

31.42 v0.23.2 (May 7, 2021)

- Feature Added support for *myrange.wrap_text* (GH173).
- Enhancement *xlwings.view()* and *xlwings.load()* now use chunking by default (GH1570).
- Bug Fix Allow to save non-Excel file formats (GH1569)
- Bug Fix Calculate formulas by default in the Function Wizard (GH1574).
- PRO Bug Fix Properly embed code with unicode characters (GH1575).

31.43 v0.23.1 (Apr 19, 2021)

- Feature You can now save your workbook in any format you want, simply by specifying its extension:

```
mybook.save('binaryfile.xlsb')
mybook.save('macroenabled.xlsm')
```

- Feature Added support for the `chunksize` option: 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 (GH77):

```
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
```

- Enhancement `xw.load()` now expands to the `current_region` instead of relying on `expand()` (GH1565).

- Enhancement The OneDrive setting has been split up into a Windows and macOS-specific paths: `ONEDRIVE_WIN` and `ONEDRIVE_MAC` (GH1556).

- Bug Fix macOS: There are no more timeouts when opening or saving large workbooks that take longer than 60 seconds (GH618).

- Bug Fix `RunPython` was failing when there was a & in the Excel file name (GH1557).

31.44 v0.23.0 (Mar 5, 2021)

- PRO Feature: This release adds support for Markdown-based formatting of text, both in cells as well as in shapes, see *Markdown* for the details. This is also supported for template-based reports.

```
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]
sheet['A1'].value = Markdown(mytext)
sheet.shapes[0].text = Markdown(mytext)
```

Running this code will give you this nicely formatted text, but you can also define your own style to match your corporate style guide as explained under *Markdown*:

- Feature Added support for the `Font` object via `range` or `shape` objects, see *Font* (GH897 and GH559).

- Feature Added support for the `Characters` object via `range` or `shape` objects, see *Characters*.

31.45 v0.22.3 (Mar 3, 2021)

- Enhancement As a convenience method, you can now directly export sheets to PDF instead of having to go through the book: `mysheet.to_pdf()` (GH1517).

- PRO Bug Fix Running `RunPython` with embedded code was broken in 0.22.0 (GH1530).

31.46 v0.22.2 (Feb 8, 2021)

- Bug Fix Windows: If the path of the Excel file included a single quote, UDFs were failing (GH1511).

- Bug Fix macOS: Prevent Excel from showing up when using hidden Excel instances via `xw.App(visible=False)` (GH1508).

31.47 v0.22.1 (Feb 4, 2021)

- PRO Bug Fix: `Table.update` has been fixed so it also works when the table is the data source of a chart (GH1507).

- PRO [Docs]: New documentation about how to work with Excel charts in templates; see *xlwings Reports PRO*.

31.48 v0.22.0 (Jan 29, 2021)

- Feature While it's always been possible to *somehow* create your own xlwings-based add-ins, this release adds a toolchain to make it a lot easier to create your own white-labeled add-in, see *Custom Add-ins* (GH1488).

- Enhancement `xw.view` now formats the pandas DataFrames as Excel table and with the new `xw.load` function, you can easily load a DataFrame from your active workbook into a Jupyter notebook. See *Jupyter Notebooks: Interact with Excel* for a full tutorial (GH1487).

- Feature New method `mysheet.copy()` (GH123).

- PRO Feature: in addition to `xw.create_report()`, you can now also work within a workbook by using the new `mysheet.render_template()` method, see also *xlwings Reports PRO* (GH1478).

31.49 Older Releases

v0.21.4 (Nov 23, 2020)

- Enhancement New property `Shape.text` to read and write text to the text frame of shapes (GH1456).

- PRO Feature: xlwings Reports now supports template text in shapes, see *xlwings Reports*.

v0.21.3 (Nov 22, 2020)

- PRO Breaking Change: The `Table.update` method has been changed to treat the DataFrame's index consistently whether or not it's being written to an Excel table: by default, the index is now transferred to Excel in both cases.

v0.21.2 (Nov 15, 2020)

- Bug Fix The default `quickstart` setup now also works when you store your workbooks on OneDrive (GH1275)

- Bug Fix Excel files that have single quotes in their paths are now working correctly (GH1021)

v0.21.1 (Nov 13, 2020)

- Enhancement Added new method `Book.to_pdf()` to easily export PDF reports. Needless to say, this integrates very nicely with *xlwings Reports* (GH1363).

- Enhancement Added support for `Sheet.visible` (GH1459).

v0.21.0 (Nov 9, 2020)

- Enhancement Added support for Excel tables, see: `Table` and `Tables` and `range.table` (GH47 and GH1364)

- Enhancement: When using UDFs, you can now use `'range'` for the `convert` argument where you would use before `xw.Range`. The latter will be removed in a future version (GH1455).

- Enhancement Windows: The `comtypes` requirement has been dropped (GH1443).

- PRO Feature: `Table.update` offers an easy way to keep your Excel tables in sync with your DataFrame source (GH1454).

- PRO Enhancement: The reports package now supports Excel tables in the templates. This is e.g. helpful to style the tables with striped rows, see *Excel Tables* (GH1364).

v0.20.8 (Oct 18, 2020)

- Enhancement Windows: With UDFs, you can now get easy access to the caller (an xlwings range object) by using `caller` as a function argument (GH1434). In that sense, `caller` is now a reserved argument by xlwings and if you have any existing arguments with this name, you'll need to rename them:

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

- Bug Fix Windows: The setting `Show Console` now also shows/hides the command prompt properly when using the UDF server with Conda. There is no more switching between `python` and `pythonw` required (GH1435 and GH1421).

- Bug Fix Windows: Functions called via `RunPython` with `Use UDF Server` activated don't require the `xw.sub` decorator anymore (GH1418).

v0.20.7 (Sep 3, 2020)

- Bug Fix Windows: Fix a regression introduced with 0.20.0 that would cause an `AttributeError: Range.CLSID` with async and legacy dynamic array UDFs (GH1404).

- Enhancement: Matplotlib figures are now converted to 300 dpi pictures for better quality when using them with `pictures.add` (GH1402).

v0.20.6 (Sep 1, 2020)

- Bug Fix macOS: `App(visible=False)` has been fixed (GH652).

- Bug Fix macOS: The regression with `Book.fullname` that was introduced with 0.20.1 has been fixed (GH1390).

- Bug Fix Windows: The retry mechanism has been improved (GH1398).

v0.20.5 (Aug 27, 2020)

- Bug Fix The conda version check was failing with spaces in the installation path (GH1396).

- Bug Fix Windows: when running `app.quit()`, the application is now properly closed without leaving a zombie process behind (GH1397).

v0.20.4 (Aug 20, 2020)

- Enhancement The add-in can now optionally be installed without the password protection: `xlwings addin install --unprotected` (GH1392).

v0.20.3 (Aug 15, 2020)

- Bug Fix The conda version check was erroneously triggered when importing UDFs on systems without conda. (GH1389).

v0.20.2 (Aug 13, 2020)

- PRO Feature: Code can now be embedded by calling the new `xlwings code embed [--file]` CLI command (GH1380).

- Bug Fix Made the import UDFs functionality more robust to prevent an Automation 440 error that some users would see (GH1381).

- Enhancement The standalone Excel file now includes all VBA dependencies to make it work on Windows and macOS (GH1349).

- Enhancement xlwings now blocks the call if the Conda Path/Env settings are used with legacy Conda installations (GH1384).

v0.20.1 (Aug 7, 2020)

- Bug Fix macOS: password-protected sheets caused an alert when calling `xw.Book` (GH1377).

- Bug Fix macOS: calling `wb.save('newname.xlsx')` wasn't updating the `wb` object properly and caused an alert (GH1129 and GH626 and GH957).

v0.20.0 (Jul 22, 2020)

This version drops support for Python 3.5

- Feature New property `xlwings.App.status_bar` (GH1362).

- Enhancement `xlwings.view()` now becomes the active window, making it easier to work with in interactive workflows (please speak up if you feel differently) (GH1353).

- Bug Fix The UDF server has received a serious upgrade by njwhite, getting rid of the many issues that were around with using a combination of async functions and legacy dynamic arrays. You can now also call functions defined via `async def`, although for the time being they are still called synchronously from Excel (GH1010 and GH1164).

v0.19.5 (Jul 5, 2020)

- Enhancement When you install the add-in via `xlwings addin install`, it autoconfigures the add-in if it can't find an existing user config file (GH1322).

- Feature New `xlwings config create [--force]` command that autogenerates the user config file with the Python settings from which you run the command. Can be used to reset the add-in settings with the `--force` option (GH1322).

- Feature: There is a new option to show/hide the console window. Note that with `Conda Path` and `Conda Env` set, the console always pops up when using the UDF server. Currently only available on Windows (GH1182).

- Enhancement The `Interpreter` setting has been deprecated in favor of platform-specific settings: `Interpreter_Win` and `Interpreter_Mac`, respectively. This allows you to use the sheet config unchanged on both platforms (GH1345).

- Enhancement On macOS, you can now use a few environment-like variables in your settings: `$HOME`, `$APPLICATIONS`, `$DOCUMENTS`, `$DESKTOP` (GH615).

- Bug Fix: Async functions sometimes caused an error on older Excel versions without dynamic arrays (GH1341).

v0.19.4 (May 20, 2020)

- Feature `xlwings addin install` is now available on macOS. On Windows, it has been fixed so it should now work reliably (GH704).

- Bug Fix Fixed a `dll load failed` issue with `pywin32` when installed via `pip` on Python 3.8 (GH1315).

v0.19.3 (May 19, 2020)

- PRO Feature: Added possibility to create deployment keys.

v0.19.2 (May 11, 2020)

- Feature New methods `xlwings.Shape.scale_height()` and `xlwings.Shape.scale_width()` (GH311).

- Bug Fix Using `Pictures.add` is not distorting the proportions anymore (GH311).

- PRO Feature: Added support for *Plotly* (GH1309).
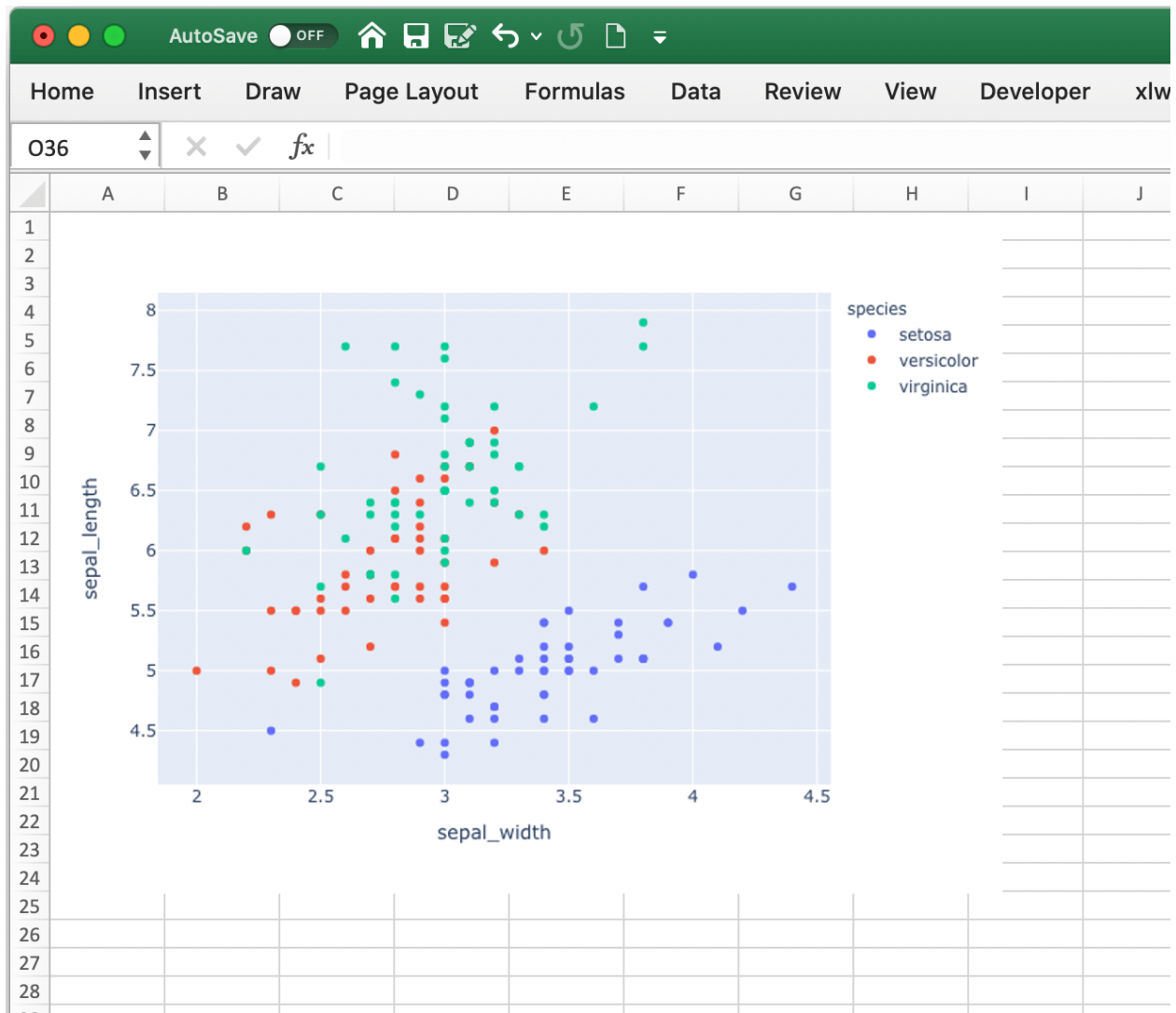
v0.19.1 (May 4, 2020)

- Bug Fix Fixed an issue with the xlwings PRO license key when there was no `xlwings.conf` file ([GH1308](#)).

v0.19.0 (May 2, 2020)

- Bug Fix Native dynamic array formulas can now be used with async formulas ([GH1277](#))

- Enhancement Quickstart references the project's name when run from Python instead of the active book ([GH1307](#))

Breaking Change:

- `Conda Base` has been renamed into `Conda Path` to reduce the confusion with the `Conda Env` called `base`. Please adjust your settings accordingly! ([GH1194](#))

v0.18.0 (Feb 15, 2020)

- Feature Added support for merged cells: *xlwings.Range.merge_area*, *xlwings.Range.merge_cells*, *xlwings.Range.merge() xlwings.Range.unmerge()* ([GH21](#)).

- Bug Fix `RunPython` now works properly with files that have a URL as `fullname`, i.e. OneDrive and SharePoint ([GH1253](#)).

- Bug Fix Fixed a bug with `wb.names['...'].refers_to_range` on macOS ([GH1256](#)).

v0.17.1 (Jan 31, 2020)

- Bug Fix Handle `np.float64('nan')` correctly ([GH1116](#)).

v0.17.0 (Jan 6, 2020)

This release drops support for Python 2.7 in xlwings CE. If you still rely on Python 2.7, you will need to stick to v0.16.6.

v0.16.6 (Jan 5, 2020)

- Enhancement CLI changes with respect to `xlwings license` ([GH1227](#)).

v0.16.5 (Dec 30, 2019)

- Enhancement Improvements with regards to the `Run main` ribbon button ([GH1207](#) and [GH1222](#)).

v0.16.4 (Dec 17, 2019)

- Enhancement Added support for *xlwings.Range.copy()* ([GH1214](#)).

- Enhancement Added support for *xlwings.Range.paste()* ([GH1215](#)).

- Enhancement Added support for *xlwings.Range.insert()* ([GH80](#)).

- Enhancement Added support for *xlwings.Range.delete()* ([GH862](#)).

v0.16.3 (Dec 12, 2019)

- Bug Fix Sometimes, xlwings would show an error of a previous run. Moreover, 0.16.2 introduced an issue that would not show errors at all on non-conda setups ([GH1158](#) and [GH1206](#))

- Enhancement The xlwings CLI now prints the version number ([GH1200](#))

Breaking Change

- `LOG FILE` has been retired and removed from the configuration/add-in.

v0.16.2 (Dec 5, 2019)

- Bug Fix `RunPython` can now be called in parallel from different Excel instances ([GH1196](#)).

v0.16.1 (Dec 1, 2019)

- Enhancement *xlwings.Book()* and `myapp.books.open()` now accept parameters like `update_links`, `password` etc. ([GH1189](#)).
- Bug Fix `Conda Env` now works correctly with `base` for UDFs, too ([GH1110](#)).
- Bug Fix `Conda Base` now allows spaces in the path ([GH1176](#)).
- Enhancement The UDF server timeout has been increased to 2 minutes ([GH1168](#)).

v0.16.0 (Oct 13, 2019)

This release adds a small but very powerful feature: There's a new `Run main` button in the add-in. With that, you can run your Python scripts from standard `xlsx` files - no need to save your workbook as macro-enabled anymore!

The only condition to make that work is that your Python script has the same name as your workbook and that it contains a function called `main`, which will be called when you click the `Run` button. All settings from your config file or config sheet are still respected, so this will work even if you have the source file in a different directory than your workbook (as long as that directory is added to the `PYTHONPATH` in your config).

The `xlwings quickstart myproject` has been updated accordingly. It still produces an `xlsm` file at the moment but you can save it as `xlsx` file if you intend to run it via the new `Run` button.
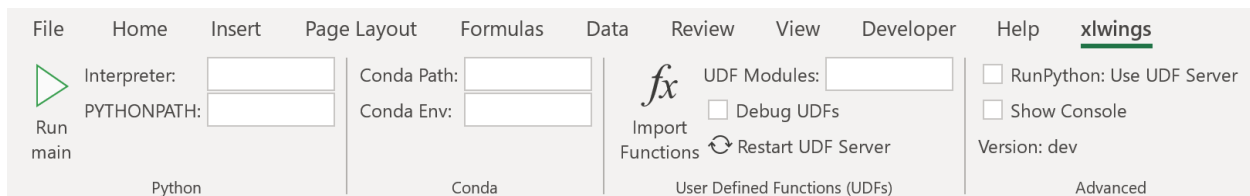
v0.15.10 (Aug 31, 2019)

- Bug Fix Fixed a Python 2.7 incompatibility introduced with 0.15.9.

v0.15.9 (Aug 31, 2019)

- Enhancement The `sql` extension now uses the native dynamic arrays if available ([GH1138](#)).
- Enhancement xlwings now support `Path` objects from `pathlib` for all file paths ([GH1126](#)).
- Bug Fix Various bug fixes: ([GH1118](#)), ([GH1131](#)), ([GH1102](#)).

v0.15.8 (May 5, 2019)

- Bug Fix Fixed an issue introduced with the previous release that always showed the command prompt when running UDFs, not just when using conda envs ([GH1098](#)).

v0.15.7 (May 5, 2019)

- Bug Fix `Conda Base` and `Conda Env` weren't stored correctly in the config file from the ribbon ([GH1090](#)).

- Bug Fix UDFs now work correctly with `Conda Base` and `Conda Env`. Note, however, that currently there is no way to hide the command prompt in that configuration ([GH1090](#)).

- Enhancement `Restart UDF Server` now actually does what it says: it stops and restarts the server. Previously it was only stopping the server and only when the first call to Python was made, it was started again ([GH1096](#)).

v0.15.6 (Apr 29, 2019)

- Feature New default converter for `OrderedDict` ([GH1068](#)).

- Enhancement `Import Functions` now restarts the UDF server to guarantee a clean state after importing. ([GH1092](#))

- Enhancement The ribbon now shows tooltips on Windows ([GH1093](#))

- Bug Fix RunPython now properly supports conda environments on Windows (they started to require proper activation with packages like numpy etc). Conda >=4.6. required. A fix for UDFs is still pending ([GH954](#)).

Breaking Change

- Bug Fix `RunFronzenPython` now accepts spaces in the path of the executable, but in turn requires to be called with command line arguments as a separate VBA argument. Example: `RunFrozenPython "C:\path\to\frozen_executable.exe", "arg1 arg2"` ([GH1063](#)).

v0.15.5 (Mar 25, 2019)

- Enhancement `wb.macro()` now accepts xlwings objects as arguments such as `range`, `sheet` etc. when the VBA macro expects the corresponding Excel object (e.g. `Range`, `Worksheet` etc.) ([GH784](#) and [GH1084](#))

Breaking Change

- Cells that contain a cell error such as `#DIV/0!`, `#N/A`, `#NAME?`, `#NULL!`, `#NUM!`, `#REF!`, `#VALUE!` return now `None` as value in Python. Previously they were returned as constant on Windows (e.g. `-2146826246`) or `k.missing_value` on Mac.

v0.15.4 (Mar 17, 2019)

- [Win] BugFix: The ribbon was not showing up in Excel 2007. ([GH1039](#))

- Enhancement: Allow to install xlwings on Linux even though it's not a supported platform: `export INSTALL_ON_LINUX=1; pip install xlwings` ([GH1052](#))

v0.15.3 (Feb 23, 2019)

Bug Fix release:

- [Mac] *RunPython* was broken by the previous release. If you install via `conda`, make sure to run `xlwings runpython install` again! ([GH1035](#))

- [Win] Sometimes, the ribbon was throwing errors ([GH1041](#))

v0.15.2 (Feb 3, 2019)

Better support and docs for deployment, see *Deployment*:

- You can now package your python modules into a zip file for easier distribution (GH1016).

- `RunFrozenPython` now allows to includes arguments, e.g. `RunFrozenPython "C:\path\to\my.exe arg1 arg2"` (GH588).

Breaking Change

- Accessing a not existing PID in the `apps` collection raises now a `KeyError` instead of an `Exception` (GH1002).

v0.15.1 (Nov 29, 2018)

Bug Fix release:

- [Win] Calling Subs or UDFs from VBA was causing an error (GH998).

v0.15.0 (Nov 20, 2018)

Dynamic Array Refactor

While we're all waiting for the new native dynamic arrays, it's still going to take another while until the majority can use them (they are not yet part of Office 2019).

In the meantime, this refactor improves the current xlwings dynamic arrays in the following way:

- Use of native ("legacy") array formulas instead of having a normal formula in the top left cell and writing around it

- It's up to 2x faster

- There's no empty row/col required outside of the dynamic array anymore

- It continues to overwrite existing cells (no change there)

- There's a small breaking change in the unlikely case that you were assigning values with the expand option: `myrange.options(expand='table').value = [['b'] * 3] * 3`. This was previously clearing contiguous cells to the right and bottom (or one of them depending on the option), now you have to do that explicitly.

Bug Fixes:

- Importing multiple UDF modules has been fixed (GH991).

v0.14.1 (Nov 9, 2018)

This is a bug fix release:

- [Win] Fixed an issue when the new `async_mode` was used together with numpy arrays (GH984)

- [Mac] Fixed an issue with multiple arguments in `RunPython` (GH905)

- [Mac] Fixed an issue with the config file (GH982)

v0.14.0 (Nov 5, 2018)

Features:

This release adds support for asynchronous functions (like all UDF related functionality, this is only available on Windows). Making a function asynchronous is as easy as:

```python
import xlwings as xw
import time


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

See *Asynchronous UDFs* for the full docs.

Bug Fixes:

- See GH970 and GH973.

v0.13.0 (Oct 22, 2018)

Features:

This release adds a REST API server to xlwings, allowing you to easily expose your workbook over the internet, see *REST API* for all the details!

Enhancements:

- Dynamic arrays are now more robust. Before, they often didn't manage to write everything when there was a lot going on in the workbook (GH880)

- Jagged arrays (lists of lists where not all rows are of equal length) now raise an error (GH942)

- xlwings can now be used with threading, see the docs: *Threading* (GH759).

- [Win] xlwings now enforces pywin32 224 when installing xlwings on Python 3.7 (GH959)

- New `xlwings.Sheet.used_range` property (GH112)

Bug Fixes:

- The current directory is now inserted in front of everything else on the PYTHONPATH (GH958)

- The standalone files had an issue in the VBA module (GH960)

Breaking Change

- Members of the `xw.apps` collection are now accessed by key (=PID) instead of index, e.g.: `xw.apps[12345]` instead of `xw.apps[0]`. The apps collection also has a new `xw.apps.keys()` method. (GH951)

v0.12.1 (Oct 7, 2018)

[Py27] Bug Fix for a Python 2.7 glitch.

v0.12.0 (Oct 7, 2018)

Features:

This release adds support to call Python functions from VBA in all Office apps (e.g. Access, Outlook etc.), not just Excel. As this uses UDFs, it is only available on Windows. See the docs: *xlwings with other Office Apps*.

Breaking Change

Previously, Python functions were always returning 2d arrays when called from VBA, no matter whether it was actually a 2d array or not. Now you get the proper dimensionality which makes it easier if the return value is e.g. a string or scalar as you don't have to unpack it anymore.

Consider the following example using the VBA Editor's Immediate Window after importing UDFs from a project created using by `xlwings quickstart`:

Old behaviour

```
?TypeName(hello("xlwings"))
Variant()
?hello("xlwings")(0,0)
hello xlwings
```

New behaviour

```
?TypeName(hello("xlwings"))
String
?hello("xlwings")
hello xlwings
```

Bug Fixes:

- [Win] Support expansion of environment variables in config values (GH615)

- Other bug fixes: GH889, GH939, GH940, GH943.

v0.11.8 (May 13, 2018)

- [Win] pywin32 is now automatically installed when using pip (GH827)

- *xlwings.bas* has been readded to the python package. This facilitates e.g. the use of xlwings within other addins (GH857)

v0.11.7 (Feb 5, 2018)

- [Win] This release fixes a bug introduced with v0.11.6 that wouldn't allow to open workbooks by name (GH804)

v0.11.6 (Jan 27, 2018)

Bug Fixes:

- [Win] When constantly writing to a spreadsheet, xlwings now correctly resumes after clicking into cells, previously it was crashing. (GH587)

- Options are now correctly applied when writing to a sheet (GH798)

v0.11.5 (Jan 7, 2018)

This is mostly a bug fix release:

- Config files can now additionally be saved in the directory of the workbooks, overriding the global Ribbon config, see *Making use of Environment Variables* (GH772)

- Reading Pandas DataFrames with a simple index was creating a MultiIndex with Pandas > 0.20 (GH786)

- [Win] The xlwings dlls are now properly versioned, allowing to use pre 0.11 releases in parallel with >0.11 releases (GH743)

- [Mac] Sheet.names.add() was always adding the names on workbook level (GH771)

- [Mac] UDF decorators now don't cause errors on Mac anymore (GH780)

v0.11.4 (Jul 23, 2017)

This release brings further improvements with regards to the add-in:

- The add-in now shows the version on the ribbon. This makes it easy to check if you are using the correct version (GH724):
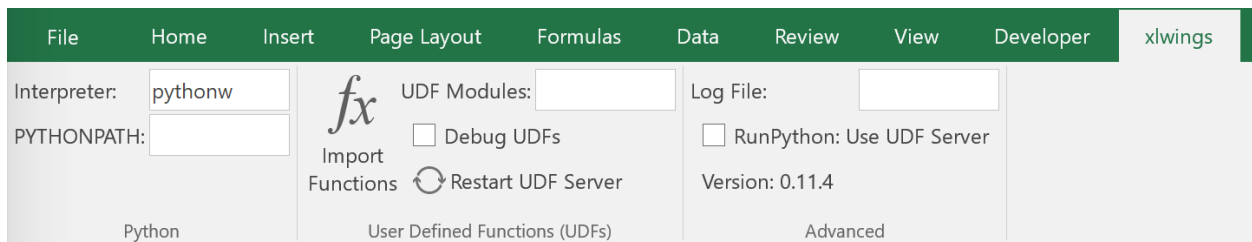
- [Mac] On Mac Excel 2016, the ribbon now only shows the available functionality (GH723):
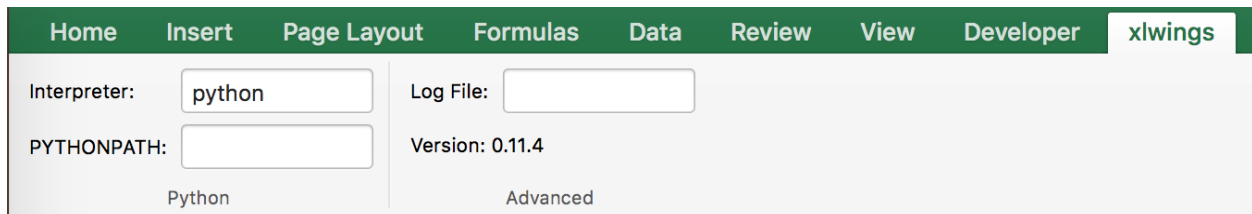
- [Mac] Mac Excel 2011 is now supported again with the new add-in. However, since Excel 2011 doesn't support the ribbon, the config file has been created/edited manually, see *Making use of Environment Variables* (GH714).

Also, some new docs:

- [Win] How to use imported functions in VBA, see *Call UDFs from VBA*.

- For more up-to-date installations via conda, use the `conda-forge` channel, see *Installation*.

- A troubleshooting section: *Troubleshooting*.

v0.11.3 (Jul 14, 2017)

- Bug Fix: When using the `xlwings.conf` sheet, there was a subscript out of range error (GH708)

- Enhancement: The add-in is now password protected (pw: `xlwings`) to declutter the VBA editor ([GH710](#))

You need to update your xlwings add-in to get the fixes!

v0.11.2 (Jul 6, 2017)

- Bug Fix: The sql extension was sometimes not correctly assigning the table aliases ([GH699](#))

- Bug Fix: Permission errors during pip installation should be resolved now ([GH693](#))

v0.11.1 (Jul 5, 2017)

- Bug Fix: The sql extension installs now correctly ([GH695](#))

v0.11.0 (Jul 2, 2017)

Big news! This release adds a full blown **add-in**! We also throw in a great **In-Excel SQL Extension** and a few **bug fixes**:

Add-in
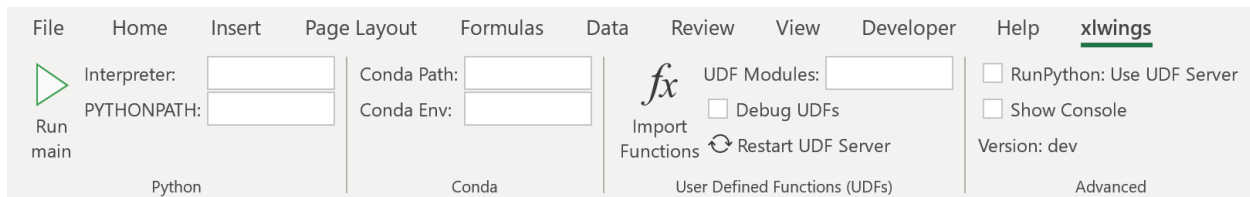
A few highlights:

- Settings don't have to be manipulated in VBA code anymore, but can be either set globally via Ribbon/config file or for the workbook via a special worksheet

- UDF server can be restarted directly from the add-in

- You can still use a VBA module instead of the add-in, but the recommended way is the add-in

- Get all the details here: *Add-in & Settings*

In-Excel SQL Extension

The add-in can be extended with own code. We throw in an `sql` function, that allows you to perform SQL queries on data in your spreadsheets. It's pretty awesome, get the details here: *Extensions*.

Bug Fixes

- [Win]: Running `Debug` > `Compile` is not throwing errors anymore ([GH678](#))

- Pandas deprecation warnings have been fixed ([GH675](#) and [GH664](#))

- [Mac]: Errors are again shown correctly in a pop up ([GH660](#))

- [Mac]: Like Windows, Mac now also only shows errors in a popup. Before it was including stdout, too ([GH666](#))

Breaking Change

- `RunFrozenPython` now requires the full path to the executable.

- The xlwings CLI `xlwings template` functionality has been removed. Use `quickstart` instead.

Migrate to v0.11 (Add-in)

This migration guide shows you how you can start using the new xlwings add-in as opposed to the old xlwings VBA module (and the old add-in that consisted of just a single import button).

Upgrade the xlwings Python package

1. Check where xlwings is currently installed

```
>>> import xlwings
>>> xlwings.__path__
```

2. If you installed xlwings with pip, for once, you should first uninstall xlwings: `pip uninstall xlwings`

3. Check the directory that you got under 1): if there are any files left over, delete the `xlwings` folder and the remaining files manually

4. Install the latest xlwings version: `pip install xlwings`

5. Verify that you have >= 0.11 by doing

```
>>> import xlwings
>>> xlwings.__version__
```

Install the add-in

1. If you have the old xlwings addin installed, find the location and remove it or overwrite it with the new version (see next step). If you installed it via the xlwings command line client, you should be able to do: `xlwings addin remove`.

2. Close Excel. Run `xlwings addin install` from a command prompt. Reopen Excel and check if the xlwings Ribbon appears. If not, copy `xlwings.xlam` (from your xlwings installation folder under `addin\xlwings.xlam` manually into the `XLSTART` folder. You can find the location of this folder under Options > Trust Center > Trust Center Settings... > Trusted Locations, under the description `Excel default location:  User StartUp`. Restart Excel and you should see the add-in.

Upgrade existing workbooks

1. Make a backup of your Excel file

2. Open the file and go to the VBA Editor (`Alt-F11`)

3. Remove the xlwings VBA module

4. Add a reference to the xlwings addin, see *Installation*

5. If you want to use workbook specific settings, add a sheet `xlwings.conf`, see *Workbook Config: xlwings.conf Sheet*

Note: To import UDFs, you need to have the reference to the xlwings add-in set!

v0.10.4 (Feb 19, 2017)

- [Win] Bug Fix: v0.10.3 introduced a bug that imported UDFs by default with *volatile=True*, this has now been fixed. You will need to reimport your functions after upgrading the xlwings package.

v0.10.3 (Jan 28, 2017)

This release adds new features to User Defined Functions (UDFs):

- categories

- volatile option

- suppress calculation in function wizard

Syntax:

```python
import xlwings as xw
@xw.func(category="xlwings", volatile=False, call_in_wizard=True)
def myfunction():
    return ...
```

For details, check out the (also new) and comprehensive API docs about the decorators: *UDF decorators*

v0.10.2 (Dec 31, 2016)

- [Win] Python 3.6 is now supported (GH592)

v0.10.1 (Dec 5, 2016)

- Writing a Pandas Series with a MultiIndex header was not writing out the header (GH572)

- [Win] Docstrings for UDF arguments are now working (GH367)

- [Mac] Range.clear_contents() has been fixed (it was doing clear() instead) (GH576)

- xw.Book(...) and xw.books.open(...) raise now the same error in case the file doesn't exist (GH540)

v0.10.0 (Sep 20, 2016)

Dynamic Array Formulas

This release adds an often requested & powerful new feature to User Defined Functions (UDFs): Dynamic expansion for array formulas. While Excel offers array formulas, you need to specify their dimensions up front by selecting the result array first, then entering the formula and finally hitting Ctrl-Shift-Enter. While this makes sense from a data integrity point of view, in practice, it often turns out to be a cumbersome limitation, especially when working with dynamic arrays such as time series data.

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))
```

B4	f_x	=dynamic_array(B2,C2)		

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

B4	f_x	=dynamic_array(B2,C2)			

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

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

Bug Fixes

- The `int` converter works now always as you would expect (e.g.: `xw.Range('A1').options(numbers=int).value`). Before, it could happen that the number was off by 1 due to floating point issues (GH554).

v0.9.3 (Aug 22, 2016)

- [Win] `App.visible` wasn't behaving correctly (GH551).

- [Mac] Added support for the new 64bit version of Excel 2016 on Mac (GH549).

- Unicode book names are again supported (GH546).

- `xlwings.Book.save()` now supports relative paths. Also, when saving an existing book under a new name without specifying the full path, it'll be saved in Python's current working directory instead of in Excel's default directory (GH185).

v0.9.2 (Aug 8, 2016)

Another round of bug fixes:

- [Mac]: Sometimes, a column was referenced instead of a named range (GH545)

- [Mac]: Python 2.7 was raising a `LookupError:  unknown encoding:  mbcs` (GH544)

- Fixed docs regarding set_mock_caller (GH543)

v0.9.1 (Aug 5, 2016)

This is a bug fix release: As to be expected after a rewrite, there were some rough edges that have now been taken care of:

- [Win] Opening a file via `xw.Book()` was causing an additional `Book1` to be opened in case Excel was not running yet (GH531)

- [Win] Some users were getting an ImportError (GH533)

- [PY 2.7] `RunPython` was broken with Python 2.7 (GH537)

- Some corrections in the docs (GH538 and GH536)

v0.9.0 (Aug 2, 2016)

Exciting times! v0.9.0 is a complete rewrite of xlwings with loads of syntax changes (hence the version jump). But more importantly, this release adds a ton of new features and bug fixes that would have otherwise been impossible. Some of the highlights are listed below, but make sure to check out the full *migration guide* for the syntax changes in details. Note, however, that the syntax for user defined functions (UDFs) did not change. At this point, the API is fairly stable and we're expecting only smaller changes on our way towards a stable v1.0 release.

- **Active** book instead of **current** book: `xw.Range('A1')` goes against the active sheet of the active book like you're used to from VBA. Instantiating an explicit connection to a Book is not necessary anymore:

```
>>> import xlwings as xw
>>> xw.Range('A1').value = 11
>>> xw.Range('A1').value
11.0
```

- Excel Instances: Full support of multiple Excel instances (even on Mac!)

```
>>> app1 = xw.App()
>>> app2 = xw.App()
>>> xw.apps
Apps([<Excel App 1668>, <Excel App 1644>])
```

- New powerful object model based on collections and close to Excel's original, allowing to fully qualify objects: `xw.apps[0].books['MyBook.xlsx'].sheets[0].range('A1:B2').value`

  It supports both Python indexing (square brackets) and Excel indexing (round brackets):

  `xw.books[0].sheets[0]` is the same as `xw.books(1).sheets(1)`

  It also supports indexing and slicing of range objects:

```
>>> rng = xw.Range('A1:E10')
>>> rng[1]
<Range [Workbook1]Sheet1!$B$1>
>>> rng[:2, :2]
<Range [Workbook1]Sheet1!$A$1:$B$2>
```

  For more details, see *Syntax Overview*.

- UDFs can now also be imported from packages, not just modules (GH437)

- Named Ranges: Introduction of full object model and proper support for sheet and workbook scope (GH256)

- Excel doesn't become the active window anymore so the focus stays on your Python environment (GH414)

- When writing to ranges while Excel is busy, xlwings is now retrying until Excel is idle again (GH468)

- *xlwings.view()* has been enhanced to accept an optional sheet object (GH469)

- Objects like books, sheets etc. can now be compared (e.g. `wb1 == wb2`) and are properly hashable

- Note that support for Python 2.6 has been dropped

Some of the new methods/properties worth mentioning are:

- *xlwings.App.display_alerts*

- *xlwings.App.macro()* in addition to *xlwings.Book.macro()*

- *xlwings.App.kill()*

- *xlwings.Sheet.cells*

- *xlwings.Range.rows*

- *xlwings.Range.columns*

- *xlwings.Range.end()*

- *xlwings.Range.raw_value*

Bug Fixes

- See here for details about which bugs have been fixed.

Migrate to v0.9

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

Full qualification: Using collections

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

- **old**: `xw.Range('Sheet1', 'A1', wkb=xw.Workbook('Book1'))`

- **new**: `xw.apps[0].books['Book1'].sheets['Sheet1'].range('A1')`

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

Connecting to Books

- **old**: `xw.Workbook()`

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

See *Connect to a Book* for the details.

Active Objects

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

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

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

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

Round vs. Square Brackets

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

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

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

Access the underlying Library/Engine

- **old**: `xw.Range('A1').xl_range` and `xl_sheet` etc.

- **new**: `xw.Range('A1').api`, same for all other objects

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

Cheat sheet

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

	v0.9.0	v0.7.2
Active Excel instance	`xw.apps.active`	unsupported
New Excel instance	`app = xw.App()`	unsupported
Get app from book	`app = wb.app`	`app = xw.Application(wb)`
Target installation (Mac)	`app = xw.App(spec=...)`	`wb = xw.Workbook(app_targe`
Hide Excel Instance	`app = xw.App(visible=False)`	`wb = xw.Workbook(app_visib`
Selected Range	`app.selection`	`wb.get_selection()`
Calculation mode	`app.calculation = 'manual'`	`app.calculation = xw.const`
All books in app	`app.books`	unsupported
Fully qualified book	`app.books['Book1']`	unsupported
Active book in active app	`xw.books.active`	`xw.Workbook.active()`
New book in active app	`wb = xw.Book()`	`wb = xw.Workbook()`
New book in specific app	`wb = app.books.add()`	unsupported
All sheets in book	`wb.sheets`	`xw.Sheet.all(wb)`
Call a macro in an addin	`app.macro('MacroName')`	unsupported
First sheet of book wb	`wb.sheets[0]`	`xw.Sheet(1, wkb=wb)`
Active sheet	`wb.sheets.active`	`xw.Sheet.active(wkb=wb) or`
Add sheet	`wb.sheets.add()`	`xw.Sheet.add(wkb=wb)`
Sheet count	`wb.sheets.count or len(wb.sheets)`	`xw.Sheet.count(wb)`
Add chart to sheet	`chart = wb.sheets[0].charts.add()`	`chart = xw.Chart.add(sheet`
Existing chart	`wb.sheets['Sheet 1'].charts[0]`	`xw.Chart('Sheet 1', 1)`
Chart Type	`chart.chart_type = '3d_area'`	`chart.chart_type = xw.cons`
Add picture to sheet	`wb.sheets[0].pictures.add('path/to/pic')`	`xw.Picture.add('path/to/pi`
Existing picture	`wb.sheets['Sheet 1'].pictures[0]`	`xw.Picture('Sheet 1', 1)`
Matplotlib	`sht.pictures.add(fig, name='x', update=True)`	`xw.Plot(fig).show('MyPlot'`

Table 1 – continued from previous page

	v0.9.0	v0.7.2
Table expansion	`sht.range('A1').expand('table')`	`xw.Range(sht, 'A1', wkb=wk`
Vertical expansion	`sht.range('A1').expand('down')`	`xw.Range(sht, 'A1', wkb=wk`
Horizontal expansion	`sht.range('A1').expand('right')`	`xw.Range(sht, 'A1', wkb=wk`
Set name of range	`sht.range('A1').name = 'name'`	`xw.Range(sht, 'A1', wkb=wk`
Get name of range	`sht.range('A1').name.name`	`xw.Range(sht, 'A1', wkb=wk`
mock caller	`xw.Book('file.xlsm').set_mock_caller()`	`xw.Workbook.set_mock_calle`

v0.7.2 (May 18, 2016)

Bug Fixes

- [Win] UDFs returning Pandas DataFrames/Series containing `nan` were failing (GH446).

- [Win] `RunFrozenPython` was not finding the executable (GH452).

- The xlwings VBA module was not finding the Python interpreter if `PYTHON_WIN` or `PYTHON_MAC` contained spaces (GH461).

v0.7.1 (April 3, 2016)

Enhancements

- [Win]: User Defined Functions (UDFs) support now optional/default arguments (GH363)

- [Win]: User Defined Functions (UDFs) support now multiple source files, see also under API changes below. For example (VBA settings): `UDF_MODULES="common;myproject"`

- VBA Subs & Functions are now callable from Python:

    As an example, this VBA function:

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

    can be accessed like this:

    ```
    >>> import xlwings as xw
    >>> wb = xw.Workbook.active()
    >>> my_sum = wb.macro('MySum')
    >>> my_sum(1, 2)
    3.0
    ```

- New `xw.view` method: This opens a new workbook and displays an object on its first sheet. E.g.:

    ```
    >>> import xlwings as xw
    >>> import pandas as pd
    ```

(continues on next page)

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

- New docs about *Matplotlib* and *Custom Converter*

- New method: *xlwings.Range.formula_array()* (GH411)

API changes

- VBA settings: `PYTHON_WIN` and `PYTHON_MAC` must now include the interpreter if you are not using the default (`PYTHON_WIN = ""`) (GH289). E.g.:

```
PYTHON_WIN: "C:\Python35\pythonw.exe"
PYTHON_MAC: "/usr/local/bin/python3.5"
```

- [Win]: VBA settings: `UDF_PATH` has been replaced with `UDF_MODULES`. The default behaviour doesn't change though (i.e. if `UDF_MODULES = ""`, then a Python source file with the same name as the Excel file, but with `.py` ending will be imported from the same directory as the Excel file).

  New:

```
UDF_MODULES: "mymodule"
PYTHONPATH: "C:\path\to"
```

  Old:

```
UDF_PATH: "C:\path\to\mymodule.py"
```

Bug Fixes

- Numpy scalars issues were resolved (GH415)

- [Win]: xlwings was failing with freezers like cx_Freeze (GH413)

- [Win]: UDFs were failing if they were returning `None` or `np.nan` (GH390)

- Multiindex Pandas Series have been fixed (GH383)

- [Mac]: `xlwings runpython install` was failing (GH424)

v0.7.0 (March 4, 2016)

This version marks an important first step on our path towards a stable release. It introduces **converters**, a new and powerful concept that brings a consistent experience for how Excel Ranges and their values are treated both when **reading** and **writing** but also across **xlwings.Range** objects and **User Defined Functions** (UDFs).

As a result, a few highlights of this release include:

- Pandas DataFrames and Series are now supported for reading and writing, both via Range object and UDFs

- New Range converter options: `transpose`, `dates`, `numbers`, `empty`, `expand`

- New dictionary converter

- New UDF debug server

- No more pyc files when using `RunPython`

Converters are accessed via the new `options` method when dealing with `xlwings.Range` objects or via the `@xw.arg` and `@xw.ret` decorators when using UDFs. As an introductory sample, let's look at how to read and write Pandas DataFrames:

Range object:

```
>>> import xlwings as xw
>>> import pandas as pd
>>> wb = xw.Workbook()
>>> df = xw.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:
```

```
>>> Range('A1').value = df

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

UDFs:

This is the same sample as above (starting in `Range('A13')` on screenshot). If you wanted to return a DataFrame with the defaults, the `@xw.ret` decorator can be left away.

```
@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
```

Enhancements

- Dictionary (`dict`) converter:


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

- `transpose` option: This works in both directions and finally allows us to e.g. write a list in column orientation to Excel (GH11):

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

- `dates` option: This allows us to read Excel date-formatted cells in specific formats:

```
>>> import datetime as dt
>>> Range('A1').value
datetime.datetime(2015, 1, 13, 0, 0)
>>> Range('A1').options(dates=dt.date).value
datetime.date(2015, 1, 13)
```

- `empty` option: This allows us to override the default behavior for empty cells:

```
>>> Range('A1:B1').value
[None, None]
>>> Range('A1:B1').options(empty='NA')
['NA', 'NA']
```

- `numbers` option: This transforms all numbers into the indicated type.

```
>>> xw.Range('A1').value = 1
>>> type(xw.Range('A1').value)  # Excel stores all numbers interally as
↪floats
float
>>> type(xw.Range('A1').options(numbers=int).value)
int
```

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

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

All these options work the same with decorators for UDFs, e.g. for transpose:

```
@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
```

Note: These options (`dates`, `empty`, `numbers`) currently apply to the whole Range and can't be selectively applied to e.g. only certain columns.

- UDF debug server

    The new UDF debug server allows you to easily debug UDFs: just set `UDF_DEBUG_SERVER = True` in the VBA Settings, at the top of the xlwings VBA module (make sure to update it to the latest version!). Then add the following lines to your Python source file and run it:

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

When you recalculate the Sheet, the code will stop at breakpoints or print any statements that you may have. For details, see: *Debugging*.

- pyc files: The creation of pyc files has been disabled when using `RunPython`, leaving your directory in an uncluttered state when having the Python source file next to the Excel workbook (GH326).

API changes

- UDF decorator changes (it is assumed that xlwings is imported as `xw` and numpy as `np`):

New	Old
@xw.func	@xw.xlfunc
@xw.arg	@xw.xlarg
@xw.ret	@xw.xlret
@xw.sub	@xw.xlsub

Pay attention to the following subtle change:

New	Old
@xw.arg("x", np.array)	@xw.xlarg("x", "nparray")

- Samples of how the new options method replaces the old Range keyword arguments:

New	Old
Range('A1:A2').options(ndim=2)	Range('A1:A2', atleast_2d=True)
Range('A1:B2').options(np.array)	Range('A1:B2', asarray=True)
Range('A1').options(index=False, header=False).value = df	Range('A1', index=False, header=False).value = df

- Upon writing, Pandas Series are now shown by default with their name and index name, if they exist. This can be changed using the same options as for DataFrames (GH276):

```python
import pandas as pd

# unchanged behaviour
Range('A1').value = pd.Series([1,2,3])

# Changed behaviour: This will print a header row in Excel
s = pd.Series([1,2,3], name='myseries', index=pd.Index([0,1,2], name=
→'myindex'))
Range('A1').value = s

# Control this behaviour like so (as with DataFrames):
Range('A1').options(header=False, index=True).value = s
```

- NumPy scalar values

  Previously, NumPy scalar values were returned as `np.atleast_1d`. To keep the same behaviour, this now has to be set explicitly using `ndim=1`. Otherwise they're returned as numpy scalar values.

New	Old
`Range('A1').options(np.array, ndim=1).value`	`Range('A1', asarray=True).value`

Bug Fixes

A few bugfixes were made: GH352, GH359.

v0.6.4 (January 6, 2016)

API changes

None

Enhancements

- Quickstart: It's now easier than ever to start a new xlwings project, simply use the command line client (GH306):

  `xlwings quickstart myproject` will produce a folder with the following files, ready to be used (see *Command Line Client (CLI)*):

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

- New documentation about how to use xlwings with other languages like R and Julia.

Bug Fixes

- [Win]: Importing UDFs with the add-in was throwing an error if the filename was including characters like spaces or dashes (GH331). To fix this, close Excel completely and run `xlwings addin update`.

- [Win]: `Workbook.caller()` is now also accessible within functions that are decorated with `@xlfunc`. Previously, it was only available with functions that used the `@xlsub` decorator (GH316).

- Writing a Pandas DataFrame failed in case the index was named the same as a column (GH334).

v0.6.3 (December 18, 2015)

Bug Fixes

- [Mac]: This fixes a bug introduced in v0.6.2: When using `RunPython` from VBA, errors were not shown in a pop-up window (GH330).

v0.6.2 (December 15, 2015)

API changes

- LOG_FILE: So far, the log file has been placed next to the Excel file per default (VBA settings). This
  has been changed as it was causing issues for files on SharePoint/OneDrive and Mac Excel 2016: The
  place where `LOG_FILE = ""` refers to depends on the OS and the Excel version.

Enhancements

- [Mac]: This version adds support for the VBA module on Mac Excel 2016 (i.e. the `RunPython` com-
  mand) and is now feature equivalent with Mac Excel 2011 (GH206).

Bug Fixes

- [Win]: On certain systems, the xlwings dlls weren't found (GH323).

v0.6.1 (December 4, 2015)

Bug Fixes

- [Python 3]: The command line client has been fixed (GH319).

- [Mac]: It now works correctly with `psutil>=3.0.0` (GH315).

v0.6.0 (November 30, 2015)

API changes

None

Enhancements

- **User Defined Functions (UDFs) - currently Windows only**

  The ExcelPython project has been fully merged into xlwings. This means that on Windows, UDF's
  are now supported via decorator syntax. A simple example:

  ```python
  from xlwings import xlfunc


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

  For **array formulas** with or without **NumPy**, see the docs: *User Defined Functions (UDFs)*

- **Command Line Client**

  The new xlwings command line client makes it easy to work with the xlwings **template** and the de-
  veloper **add-in** (the add-in is currently Windows-only). E.g. to create a new Excel spreadsheet from
  the template, run:

  ```
  xlwings template open
  ```

  For all commands, see the docs: *Command Line Client (CLI)*

- **Other enhancements**:

  - New method: *xlwings.Sheet.delete()*

  - New method: *xlwings.Range.top()*
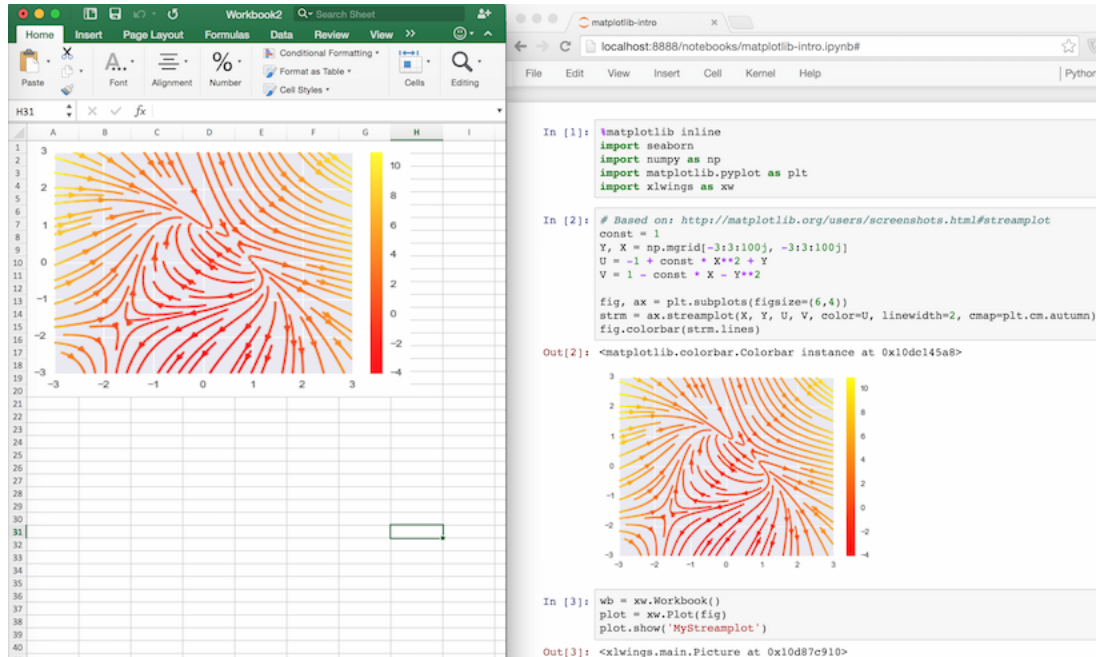
– New method: *xlwings.Range.left()*

v0.5.0 (November 10, 2015)

API changes

None

Enhancements

This version adds support for Matplotlib! Matplotlib figures can be shown in Excel as pictures in just 2 lines of code:

1) Get a matplotlib `figure` object:

- via PyPlot interface:

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

- via object oriented interface:

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

- via Pandas:

```
import pandas as pd
import numpy as np
```

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

2) Show it in Excel as picture:

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

See the full API: `xlwings.Plot()`. There's also a new example available both on GitHub and as download on the homepage.

Other enhancements:

- New *xlwings.Shape()* class

- New *xlwings.Picture()* class

- The `PYTHONPATH` in the VBA settings now accepts multiple directories, separated by `;` (GH258)

- An explicit exception is raised when `Range` is called with 0-based indices (GH106)

Bug Fixes

- `Sheet.add` was not always acting on the correct workbook (GH287)

- Iteration over a `Range` only worked the first time (GH272)

- [Win]: Sometimes, an error was raised when Excel was not running (GH269)

- [Win]: Non-default Python interpreters (as specified in the VBA settings under `PYTHON_WIN`) were not found if the path contained a space (GH257)

v0.4.1 (September 27, 2015)

API changes

None

Enhancements

This release makes it easier than ever to connect to Excel from Python! In addition to the existing ways, you can now connect to the active Workbook (on Windows across all instances) and if the Workbook is already open, it's good enough to refer to it by name (instead of having to use the full path). Accordingly, this is how you make a connection to... (GH30 and GH226):

- a new workbook: `wb = Workbook()`

- the active workbook [New!]: `wb = Workbook.active()`

- an unsaved workbook: `wb = Workbook('Book1')`

- a saved (open) workbook by name (incl. xlsx etc.) [New!]: `wb = Workbook('MyWorkbook.xlsx')`

- a saved (open or closed) workbook by path: `wb = Workbook(r'C:\\path\\to\\file.xlsx')`

Also, there are some new docs:

- *Connect to a Book*

- *Missing Features*

Bug Fixes

- The Excel template was updated to the latest VBA code (GH234).

- Connections to files that are saved on OneDrive/SharePoint are now working correctly (GH215).

- Various issues with timezone-aware objects were fixed (GH195).

- [Mac]: A certain range of integers were not written to Excel (GH227).

v0.4.0 (September 13, 2015)

API changes

None

Enhancements

The most important update with this release was made on Windows: The methodology used to make a connection to Workbooks has been completely replaced. This finally allows xlwings to reliably connect to multiple instances of Excel even if the Workbooks are opened from untrusted locations (network drives or files downloaded from the internet). This gets rid of the dreaded `Filename is already open...` error message that was sometimes shown in this context. It also allows the VBA hooks (`RunPython`) to work correctly if the very same file is opened in various instances of Excel.

Note that you will need to update the VBA module and that apart from `pywin32` there is now a new dependency for the Windows version: `comtypes`. It should be installed automatically though when installing/upgrading xlwings with `pip`.

Other updates:

- Added support to manipulate named Ranges (GH92):

```
>>> wb = Workbook()
>>> Range('A1').name = 'Name1'
>>> Range('A1').name
>>> 'Name1'
>>> del wb.names['Name1']
```

- **New `Range` properties (GH81):**

  - *xlwings.Range.column_width()*

  - *xlwings.Range.row_height()*

  - *xlwings.Range.width()*

  - *xlwings.Range.height()*

- `Range` now also accepts `Sheet` objects, the following 3 ways are hence all valid (GH92):

```
r = Range(1, 'A1')
r = Range('Sheet1', 'A1')
sheet1 = Sheet(1)
r = Range(sheet1, 'A1')
```

- [Win]: Error pop-ups show now the full error message that can also be copied with `Ctrl-C` (GH221).

Bug Fixes

- The VBA module was not accepting lower case drive letters (GH205).

- Fixed an error when adding a new Sheet that was already existing (GH211).

v0.3.6 (July 14, 2015)

API changes

`Application` as attribute of a `Workbook` has been removed (`wb` is a `Workbook` object):

Correct Syntax (as before)	Removed
`Application(wkb=wb)`	`wb.application`

Enhancements

Excel 2016 for Mac Support (GH170)

Excel 2016 for Mac is finally supported (Python side). The VBA hooks (`RunPython`) are currently not yet supported. In more details:

- This release allows Excel 2011 and Excel 2016 to be installed in parallel.

- `Workbook()` will open the default Excel installation (usually Excel 2016).

- The new keyword argument `app_target` allows to connect to a different Excel installation, e.g.:

```
Workbook(app_target='/Applications/Microsoft Office 2011/Microsoft Excel')
```

  Note that `app_target` is only available on Mac. 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.

- The `RunPython` calls in VBA are not yet available through Excel 2016 but Excel 2011 doesn't get confused anymore if Excel 2016 is installed on the same system - make sure to update your VBA module!

Other enhancements

- New method: `xlwings.Application.calculate()` (GH207)

Bug Fixes

- [Win]: When using the `OPTIMIZED_CONNECTION` on Windows, Excel left an orphaned process running after closing (GH193).

Various improvements regarding unicode file path handling, including:

- [Mac]: Excel 2011 for Mac now supports unicode characters in the filename when called via VBA's `RunPython` (but not in the path - this is a limitation of Excel 2011 that will be resolved in Excel 2016) (GH154).

- [Win]: Excel on Windows now handles unicode file paths correctly with untrusted documents. (GH154).

v0.3.5 (April 26, 2015)

API changes

`Sheet.autofit()` and `Range.autofit()`: The integer argument for the axis has been removed (GH186). Use string arguments `rows` or `r` for autofitting rows and `columns` or `c` for autofitting columns (as before).

Enhancements

New methods:

- *xlwings.Range.row()* (GH143)

- *xlwings.Range.column()* (GH143)

- *xlwings.Range.last_cell()* (GH142)

Example:

```
>>> rng = Range('A1').table
>>> rng.row, rng.column
(1, 1)
>>> rng.last_cell.row, rng.last_cell.column
(4, 5)
```

Bug Fixes

- The `unicode` bug on Windows/Python3 has been fixed (GH161)

v0.3.4 (March 9, 2015)

Bug Fixes

- The installation error on Windows has been fixed (GH160)

v0.3.3 (March 8, 2015)

API changes

None

Enhancements

- New class `Application` with `quit` method and properties `screen_updating` und `calculation` (GH101, GH158, GH159). It can be conveniently accessed from within a Workbook (on Windows, `Application` is instance dependent). A few examples:

```
>>> from xlwings import Workbook, Calculation
>>> wb = Workbook()
>>> wb.application.screen_updating = False
```

(continues on next page)

```
>>> wb.application.calculation = Calculation.xlCalculationManual
>>> wb.application.quit()
```

- New headless mode: The Excel application can be hidden either during `Workbook` instantiation or through the `application` object:

```
>>> wb = Workbook(app_visible=False)
>>> wb.application.visible
False
>>> wb.application.visible = True
```

- Newly included Excel template which includes the xlwings VBA module and boilerplate code. This is currently accessible from an interactive interpreter session only:

```
>>> from xlwings import Workbook
>>> Workbook.open_template()
```

Bug Fixes

- [Win]: `datetime.date` objects were causing an error (GH44).

- Depending on how it was instantiated, Workbook was sometimes missing the `fullname` attribute (GH76).

- `Range.hyperlink` was failing if the hyperlink had been set as formula (GH132).

- A bug introduced in v0.3.0 caused frozen versions (eg. with `cx_Freeze`) to fail (GH133).

- [Mac]: Sometimes, xlwings was causing an error when quitting the Python interpreter (GH136).

v0.3.2 (January 17, 2015)

API changes

None

Enhancements

None

Bug Fixes

- The `xlwings.Workbook.save()` method has been fixed to show the expected behavior (GH138): Previously, calling *save()* without a *path* argument would always create a new file in the current working directory. This is now only happening if the file hasn't been previously saved.

v0.3.1 (January 16, 2015)

API changes

None

Enhancements

- New method `xlwings.Workbook.save()` (GH110).

- New method `xlwings.Workbook.set_mock_caller()` (GH129). This makes calling files from both Excel and Python much easier:

```python
import os
from xlwings import Workbook, Range


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


if __name__ == '__main__':
    # To run from Python, not needed when called from Excel.
    # Expects the Excel file next to this source file, adjust accordingly.
    path = os.path.abspath(os.path.join(os.path.dirname(__file__), 'myfile.
↪xlsm'))
    Workbook.set_mock_caller(path)
    my_macro()
```

- The `simulation` example on the homepage works now also on Mac.

Bug Fixes

- [Win]: A long-standing bug that caused the Excel file to close and reopen under certain circumstances has been fixed (GH10): Depending on your security settings (Trust Center) and in connection with files downloaded from the internet or possibly in connection with some add-ins, Excel was either closing the file and reopening it or giving a "file already open" warning. This has now been fixed which means that the examples downloaded from the homepage should work right away after downloading and unzipping.

v0.3.0 (November 26, 2014)

API changes

- To reference the calling Workbook when running code from VBA, you now have to use `Workbook.caller()`. This means that `wb = Workbook()` is now consistently creating a new Workbook, whether the code is called interactively or from VBA.

New	Old
`Workbook.caller()`	`Workbook()`

Enhancements

This version adds two exciting but still **experimental** features from *ExcelPython* (**Windows only!**):

- Optimized connection: Set the `OPTIMIZED_CONNECTION = True` in the VBA settings. This will use a COM server that will keep the connection to Python alive between different calls and is therefore much more efficient. However, changes in the Python code are not being picked up until the `pythonw.exe` process is restarted by killing it manually in the Windows Task Manager. The suggested workflow is hence to set `OPTIMIZED_CONNECTION = False` for development and only set it to `True` for production - keep in mind though that this feature is still experimental!

- User Defined Functions (UDFs): Using ExcelPython's wrapper syntax in VBA, you can expose Python functions as UDFs, see *User Defined Functions (UDFs)* for details.

Note: ExcelPython's developer add-in that autogenerates the VBA wrapper code by simply using Python decorators isn't available through xlwings yet.

Further enhancements include:

- New method *xlwings.Range.resize()* (GH90).

- New method *xlwings.Range.offset()* (GH89).

- New property *xlwings.Range.shape* (GH109).

- New property *xlwings.Range.size* (GH109).

- New property *xlwings.Range.hyperlink* and new method *xlwings.Range.add_hyperlink()* (GH104).

- New property *xlwings.Range.color* (GH97).

- The `len` built-in function can now be used on `Range` (GH109):

```
>>> len(Range('A1:B5'))
5
```

- The `Range` object is now iterable (GH108):

```
for cell in Range('A1:B2'):
    if cell.value < 2:
        cell.color = (255, 0, 0)
```

- [Mac]: The VBA module finds now automatically the default Python installation as per `PATH` variable on `.bash_profile` when `PYTHON_MAC = ""` (the default in the VBA settings) (GH95).

- The VBA error pop-up can now be muted by setting `SHOW_LOG = False` in the VBA settings. To be used with care, but it can be useful on Mac, as the pop-up window is currently showing printed log messages even if no error occurred(GH94).

Bug Fixes

- [Mac]: Environment variables from `.bash_profile` are now available when called from VBA, e.g. by using: `os.environ['USERNAME']` (GH95)

v0.2.3 (October 17, 2014)

API changes

None

Enhancements

- New method `Sheet.add()` (GH71):

```
>>> Sheet.add()  # Place at end with default name
>>> Sheet.add('NewSheet', before='Sheet1')  # Include name and position
```

(continues on next page)

```
>>> new_sheet = Sheet.add(after=3)
>>> new_sheet.index
4
```

- New method `Sheet.count()`:

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

- `autofit()` works now also on `Sheet` objects, not only on `Range` objects (GH66):

```
>>> Sheet(1).autofit()  # autofit columns and rows
>>> Sheet('Sheet1').autofit('c')  # autofit columns
```

- New property `number_format` for `Range` objects (GH60):

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

  Works also with the `Range` properties `table`, `vertical`, `horizontal`:

```
>>> Range('A1').value = [1,2,3,4,5]
>>> Range('A1').table.number_format = '0.00%'
```

- New method `get_address` for `Range` objects (GH7):

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

- New method `Sheet.all()` returning a list with all Sheet objects:

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

Bug Fixes

- xlwings works now also with NumPy < 1.7.0. Before, doing something like `Range('A1').value =` `'Foo'` was causing a `NotImplementedError:  Not implemented for this type` error when NumPy < 1.7.0 was installed (GH73).

- [Win]: The VBA module caused an error on the 64bit version of Excel (GH72).

- [Mac]: The error pop-up wasn't shown on Python 3 (GH85).

- [Mac]: Autofitting bigger Ranges, e.g. `Range('A:D').autofit()` was causing a time out (GH74).

- [Mac]: Sometimes, calling xlwings from Python was causing Excel to show old errors as pop-up alert (GH70).

v0.2.2 (September 23, 2014)

API changes

- The `Workbook` qualification changed: It now has to be specified as keyword argument. Assume we have instantiated two Workbooks like so: `wb1 = Workbook()` and `wb2 = Workbook()`. Sheet, `Range` and `Chart` classes will default to `wb2` as it was instantiated last. To target `wb1`, use the new `wkb` keyword argument:

New	Old
`Range('A1', wkb=wb1).value`	`wb1.range('A1').value`
`Chart('Chart1', wkb=wb1)`	`wb1.chart('Chart1')`

Alternatively, simply set the current Workbook before using the `Sheet`, `Range` or `Chart` classes:

```
wb1.set_current()
Range('A1').value
```

- Through the introduction of the `Sheet` class (see Enhancements), a few methods moved from the `Workbook` to the `Sheet` class. Assume the current Workbook is: `wb = Workbook()`:
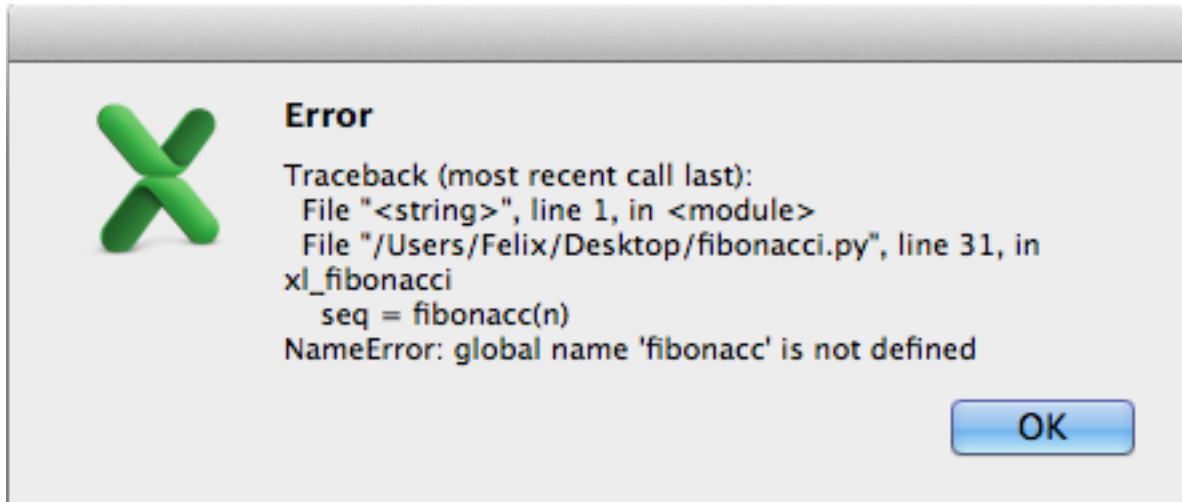
New	Old
`Sheet('Sheet1').activate()`	`wb.activate('Sheet1')`
`Sheet('Sheet1').clear()`	`wb.clear('Sheet1')`
`Sheet('Sheet1').clear_contents()`	`wb.clear_contents('Sheet1')`
`Sheet.active().clear_contents()`	`wb.clear_contents()`

- The syntax to add a new Chart has been slightly changed (it is a class method now):

New	Old
`Chart.add()`	`Chart().add()`

Enhancements

- [Mac]: Python errors are now also shown in a Message Box. This makes the Mac version feature equivalent with the Windows version (GH57):

- New `Sheet` class: The new class handles everything directly related to a Sheet. See the Python API section about `Sheet` for details (GH62). A few examples:

```
>>> Sheet(1).name
'Sheet1'
>>> Sheet('Sheet1').clear_contents()
>>> Sheet.active()
<Sheet 'Sheet1' of Workbook 'Book1'>
```

- The `Range` class has a new method `autofit()` that autofits the width/height of either columns, rows or both (GH33).

  Arguments:

```
axis : string or integer, default None
    - To autofit rows, use one of the following: 'rows' or 'r'
    - To autofit columns, use one of the following: 'columns' or 'c'
    - To autofit rows and columns, provide no arguments
```

  Examples:

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

- The `Workbook` class has the following additional methods: `current()` and `set_current()`. They determine the default Workbook for `Sheet`, `Range` or `Chart`. On Windows, in case there are various Excel instances, when creating new or opening existing Workbooks, they are being created in the same

instance as the current Workbook.

```
>>> wb1 = Workbook()
>>> wb2 = Workbook()
>>> Workbook.current()
<Workbook 'Book2'>
>>> wb1.set_current()
>>> Workbook.current()
<Workbook 'Book1'>
```

- If a `Sheet`, `Range` or `Chart` object is instantiated without an existing `Workbook` object, a user-friendly error message is raised (GH58).

- New docs about *Debugging* and *Data Structures Tutorial*.

Bug Fixes

- The `atleast_2d` keyword had no effect on Ranges consisting of a single cell and was raising an error when used in combination with the `asarray` keyword. Both have been fixed (GH53):

```
>>> Range('A1').value = 1
>>> Range('A1', atleast_2d=True).value
[[1.0]]
>>> Range('A1', atleast_2d=True, asarray=True).value
array([[1.]])
```

- [Mac]: After creating two new unsaved Workbooks with `Workbook()`, any `Sheet`, `Range` or `Chart` object would always just access the latest one, even if the Workbook had been specified (GH63).

- [Mac]: When xlwings was imported without ever instantiating a `Workbook` object, Excel would start upon quitting the Python interpreter (GH51).

- [Mac]: When installing xlwings, it now requires `psutil` to be at least version `2.0.0` (GH48).

v0.2.1 (August 7, 2014)

API changes

None

Enhancements

- All VBA user settings have been reorganized into a section at the top of the VBA xlwings module:

```
PYTHON_WIN = ""
PYTHON_MAC = GetMacDir("Home") & "/anaconda/bin"
PYTHON_FROZEN = ThisWorkbook.Path & "\build\exe.win32-2.7"
PYTHONPATH = ThisWorkbook.Path
LOG_FILE = ThisWorkbook.Path & "\xlwings_log.txt"
```

- Calling Python from within Excel VBA is now also supported on Mac, i.e. Python functions can be called like this: `RunPython("import bar; bar.foo()")`. Running frozen executables (`RunFrozenPython`) isn't available yet on Mac though.

Note that there is a slight difference in the way that this functionality behaves on Windows and Mac:

- **Windows**: After calling the Macro (e.g. by pressing a button), Excel waits until Python is done. In case there's an error in the Python code, a pop-up message is being shown with the traceback.

- **Mac**: After calling the Macro, the call returns instantly but Excel's Status Bar turns into "Running…" during the duration of the Python call. Python errors are currently not shown as a pop-up, but need to be checked in the log file. I.e. if the Status Bar returns to its default ("Ready") but nothing has happened, check out the log file for the Python traceback.

Bug Fixes

None

Special thanks go to Georgi Petrov for helping with this release.

v0.2.0 (July 29, 2014)

API changes

None

Enhancements

- Cross-platform: xlwings is now additionally supporting Microsoft Excel for Mac. The only functionality that is not yet available is the possibility to call the Python code from within Excel via VBA macros.

- The `clear` and `clear_contents` methods of the `Workbook` object now default to the active sheet (GH5):

```
wb = Workbook()
wb.clear_contents()  # Clears contents of the entire active sheet
```

Bug Fixes

- DataFrames with MultiHeaders were sometimes getting truncated (GH41).

v0.1.1 (June 27, 2014)

API Changes

- If `asarray=True`, NumPy arrays are now always at least 1d arrays, even in the case of a single cell (GH14):

```
>>> Range('A1', asarray=True).value
array([34.])
```

- Similar to NumPy's logic, 1d Ranges in Excel, i.e. rows or columns, are now being read in as flat lists or 1d arrays. If you want the same behavior as before, you can use the `atleast_2d` keyword (GH13).

Note: The `table` property is also delivering a 1d array/list, if the table Range is really a column or row.

```
>>> Range('A1').vertical.value
[1.0, 2.0, 3.0, 4.0]
>>> Range('A1', atleast_2d=True).vertical.value
[[1.0], [2.0], [3.0], [4.0]]
>>> Range('C1').horizontal.value
[1.0, 2.0, 3.0, 4.0]
>>> Range('C1', atleast_2d=True).horizontal.value
[[1.0, 2.0, 3.0, 4.0]]
>>> Range('A1', asarray=True).table.value
array([ 1.,  2.,  3.,  4.])
>>> Range('A1', asarray=True, atleast_2d=True).table.value
array([[ 1.],
       [ 2.],
       [ 3.],
       [ 4.]])
```

- The single file approach has been dropped. xlwings is now a traditional Python package.

Enhancements

- xlwings is now officially suppported on Python 2.6-2.7 and 3.1-3.4

- Support for Pandas `Series` has been added (GH24):

```
>>> import numpy as np
>>> import pandas as pd
>>> from xlwings import Workbook, Range
>>> wb = Workbook()
>>> s = pd.Series([1.1, 3.3, 5., np.nan, 6., 8.])
>>> s
0    1.1
1    3.3
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
>>> Range('A1').value = s
>>> Range('D1', index=False).value = s
```

- Excel constants have been added under their original Excel name, but categorized under their enum

(GH18), e.g.:

```
# Extra long version
import xlwings as xl
xl.constants.ChartType.xlArea

# Long version
from xlwings import constants
constants.ChartType.xlArea

# Short version
from xlwings import ChartType
ChartType.xlArea
```

- Slightly enhanced Chart support to control the `ChartType` (GH1):

```
>>> from xlwings import Workbook, Range, Chart, ChartType
>>> wb = Workbook()
>>> Range('A1').value = [['one', 'two'],[10, 20]]
>>> my_chart = Chart().add(chart_type=ChartType.xlLine,
                           name='My Chart',
                           source_data=Range('A1').table)
```

alternatively, the properties can also be set like this:

```
>>> my_chart = Chart().add()  # Existing Charts: my_chart = Chart('My Chart')
>>> my_chart.name = 'My Chart'
>>> my_chart.chart_type = ChartType.xlLine
>>> my_chart.set_source_data(Range('A1').table)
```

- `pytz` is no longer a dependency as `datetime` object are now being read in from Excel as time-zone naive (Excel doesn't know timezones). Before, `datetime` objects got the UTC timezone attached.

- The `Workbook` class has the following additional methods: `close()`

- The `Range` class has the following additional methods: `is_cell()`, `is_column()`, `is_row()`, `is_table()`

Bug Fixes
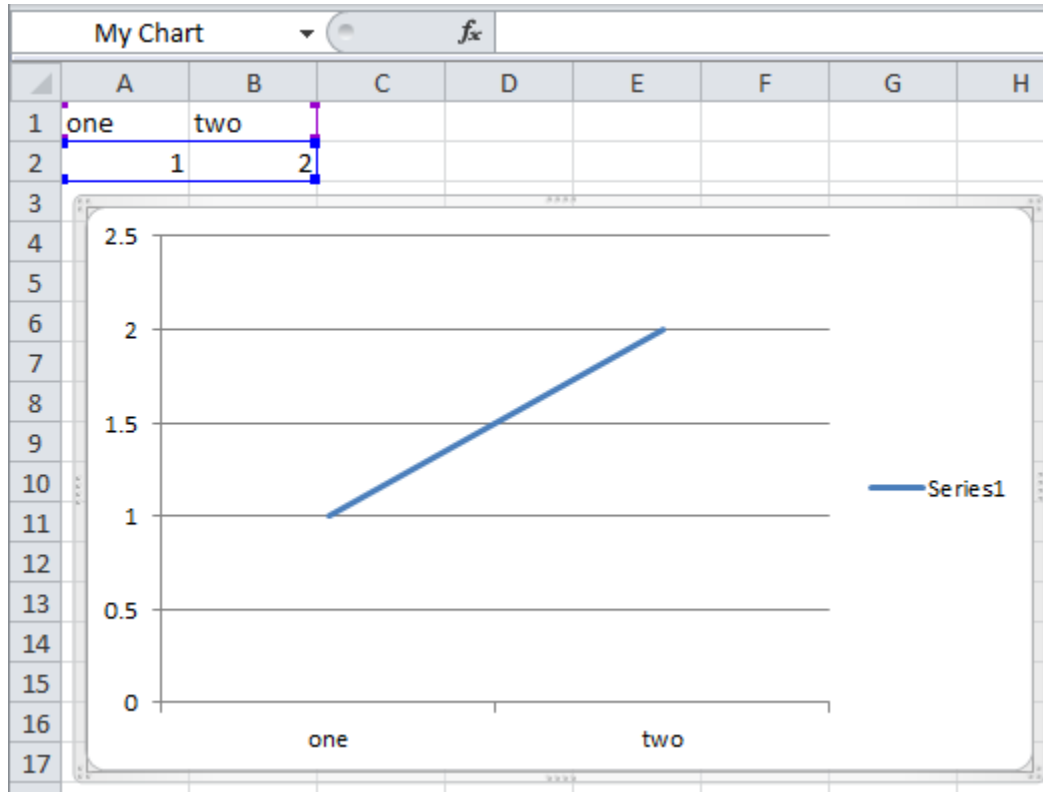
- Writing `None` or `np.nan` to Excel works now (GH16 & GH15).

- The import error on Python 3 has been fixed (GH26).

- Python 3 now handles Pandas DataFrames with MultiIndex headers correctly (GH39).

- Sometimes, a Pandas DataFrame was not handling `nan` correctly in Excel or numbers were being truncated (GH31) & (GH35).

- Installation is now putting all files in the correct place (GH20).

v0.1.0 (March 19, 2014)

Initial release of xlwings.

LICENSE

32.1 xlwings (Open Source)

xlwings (Open Source) is a distributed under a BSD-3-Clause license. xlwings (Open Source) includes all files in the xlwings package except the `pro` folder, i.e., the `xlwings.pro` subpackage.

- xlwings (Open Source) License

32.2 xlwings PRO

xlwings PRO is source available and dual-licensed under one of the following licenses:

- PolyForm Noncommercial License 1.0.0 (noncommercial use is free)
- xlwings PRO License (commercial use requires a paid plan)

OPEN SOURCE LICENSES

Depending on the platform and features that you use, xlwings requires various Open Source dependencies.

- The licenses of the compiled code are available in a separate document

- All other licenses are listed below

33.1 pywin32 (Windows only)

com subpackage

Unless stated in the specific source file, this work is Copyright (c) 1996-2008, Greg Stein and Mark Hammond. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither names of Greg Stein, Mark Hammond nor the name of contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ''AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

win32 subpackage

Unless stated in the specfic source file, this work is Copyright (c) 1994-2008, Mark Hammond All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither name of Mark Hammond nor the name of contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ''AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IM-PLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAM-AGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SER-VICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Pythonwin subpackage

Unless stated in the specfic source file, this work is Copyright (c) 1994-2008, Mark Hammond All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither name of Mark Hammond nor the name of contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ''AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IM-PLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAM-AGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SER-VICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

33.2 psutil (macOS only)

BSD 3-Clause License

Copyright (c) 2009, Jay Loden, Dave Daeschler, Giampaolo Rodola' All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of the psutil authors nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

33.3 Appscript (macOS only)

Appscript is released into the public domain, except for the following code:

- portions of ae.c, which are Copyright (C) the original authors:
  Original code taken from _AEmodule.c, _CFmodule.c, _Launchmodule.c
  Copyright (C) 2001-2008 Python Software Foundation.
  License: https://docs.python.org/3/license.html.

- SendThreadSafe.h/SendThreadSafe.m, which are modified versions of Apple
  code (https://developer.apple.com/library/archive/samplecode/AESendThreadSafe):
  Written by: DTS
  Copyright: Copyright (c) 2007 Apple Inc. All Rights Reserved.
  Disclaimer: IMPORTANT: This Apple software is supplied to you by Apple Inc.

  ("Apple") in consideration of your agreement to the following terms, and your use, installation, modification or redistribution of this Apple software constitutes acceptance of these terms. If you do not agree with these terms, please do not use, install, modify or redistribute this Apple software. In consideration of your agreement to abide by the following terms, and subject to these terms, Apple grants you a personal, non-exclusive license, under Apple's copyrights in this original Apple software (the

"Apple Software"), to use, reproduce, modify and redistribute the Apple Software, with or without modifications, in source and/or binary forms; provided that if you redistribute the Apple Software in its entirety and without modifications, you must retain this notice and the following text and disclaimers in all such redistributions of the Apple Software. Neither the name, trademarks, service marks or logos of Apple Inc. may be used to endorse or promote products derived from the Apple Software without specific prior written permission from Apple. Except as expressly stated in this notice, no other rights or licenses, express or implied, are granted by Apple herein, including but not limited to any patent rights that may be infringed by your derivative works or by other works in which the Apple Software may be incorporated. The Apple Software is provided by Apple on an "AS IS" basis.

APPLE MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE APPLE SOFTWARE OR ITS USE AND OPERATION ALONE OR IN COMBINATION WITH YOUR PRODUCTS. IN NO EVENT SHALL APPLE BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) ARISING IN ANY WAY OUT OF THE USE, REPRODUCTION, MODIFICATION AND/OR DISTRIBUTION OF THE APPLE SOFTWARE, HOWEVER CAUSED AND WHETHER UNDER THEORY OF CONTRACT, TORT (INCLUDING NEGLIGENCE), STRICT LIABILITY OR OTHERWISE, EVEN IF APPLE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

33.4 Mistune

BSD 3-Clause License

Copyright (c) 2014, Hsiaoming Yang

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of the creator nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,

OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

33.5 VBA-Dictionary

The MIT License (MIT)

Copyright (c) 2020 Tim Hall

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

33.6 VBA-Web

The MIT License (MIT)

Copyright (c) 2016-2019 Tim Hall

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

33.7 watchgod (only used in xlwings.exe)

The MIT License (MIT)

Copyright (c) 2017, 2018, 2019, 2020, 2021, 2022 Samuel Colvin

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

33.8 msal (only used in xlwings.exe)

The MIT License (MIT)

Copyright (c) Microsoft Corporation. All rights reserved.

This code is licensed under the MIT License.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files(the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and / or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions :

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

PYTHON API

34.1 Top-level functions

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

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

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

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

> **Examples**

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

> See also: *view*

> Changed in version 0.23.1.

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

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

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

Parameters

- **obj** (`any type with built-in converter`) – the object to display, e.g. numbers, strings, lists, numpy arrays, pandas DataFrames

- **sheet** (`Sheet, default None`) – Sheet object. If none provided, the first sheet of a new workbook is used.

- **table** (`bool, default True`) – If your object is a pandas DataFrame, by default it is formatted as an Excel Table

- **chunksize** (`int, default 5000`) – Chunks the loading of big arrays.

Examples

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

See also: *load*

Changed in version 0.22.0.

34.2 Object model

34.2.1 Apps

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

A collection of all *app* objects:

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

property active

Returns the active app.

New in version 0.9.0.

add(***kwargs*)

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

---

property count

>   Returns the number of apps.

>   New in version 0.9.0.

keys()

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

>   New in version 0.13.0.

34.2.2 App

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

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

```python
import xlwings as xw

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

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 frontmost application and hand over focus from Python to Excel.
>
> New in version 0.9.0.

alert(*prompt*, *title=None*, *buttons='ok'*, *mode=None*, *callback=None*)

> This corresponds to `MsgBox` in VBA, shows an alert/message box and returns the value of the pressed button. For the remote interpreter, instead of returning a value, the function accepts the name of a callback to which it will supply the value of the pressed button.
>
> > **Parameters**
> >
> > > - **prompt** (`str, default None`) – The message to be displayed.
> > > - **title** (`str, default None`) – The title of the alert.
> > > - **buttons** (str, default "ok") – Can be either "ok", "ok_cancel", "yes_no", or "yes_no_cancel".
> > > - **mode** (`str, default None`) – Can be "info" or "critical". Not supported by Google Sheets.
> > > - **callback** (`str, default None`) – Only used by the remote interpreter: you can provide the name of a function that will be called with the value of the pressed button as argument. The function has to exist on the client side, i.e., in VBA or JavaScript.
> >
> > **Returns**
> >
> > > **button_value** – Returns `None` when used with the remote interpreter, otherwise the value of the pressed button in lowercase: "ok", "cancel", "yes", "no".
> >
> > **Return type**
> >
> > > str or None
>
> New in version 0.27.13.

property api

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

property books

> A collection of all Book objects that are currently open.
>
> New in version 0.9.0.

---

calculate()

> Calculates all open books.
>
> New in version 0.3.6.

property calculation

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

> **Examples**

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

> Changed in version 0.9.0.

property cut_copy_mode

> Gets or sets the status of the cut or copy mode. Accepts `False` for setting and returns `None`, `copy` or `cut` when getting the status.
>
> New in version 0.24.0.

property display_alerts

> The default value is True. Set this property to False to suppress prompts and alert messages while code is running; when a message requires a response, Excel chooses the default response.
>
> New in version 0.9.0.

property enable_events

> `True` if events are enabled. Read/write boolean.
>
> New in version 0.24.4.

property hwnd

> Returns the Window handle (Windows-only).
>
> New in version 0.9.0.

property interactive

> `True` if Excel is in interactive mode. If you set this property to `False`, Excel blocks all input from the keyboard and mouse (except input to dialog boxes that are displayed by your code). Read/write Boolean. NOTE: Not supported on macOS.
>
> New in version 0.24.4.

kill()

> Forces the Excel app to quit by killing its process.
>
> New in version 0.9.0.

macro(*name*)

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

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

Examples

This VBA function:

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

can be accessed like this:

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

Types are supported too:

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

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

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

```
Function MySum(arr() as integer)
    ' code here
End Function
```

See also: *Book.macro()*

New in version 0.9.0.

property path

Returns the path to where the App is installed.

New in version 0.28.4.

property pid

> Returns the PID of the app.
>
> New in version 0.9.0.

properties(***kwargs*)

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

> **Examples**

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

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

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

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

> New in version 0.24.4.

quit()

> Quits the application without saving any workbooks.
>
> New in version 0.3.3.

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

> Range object from the active sheet of the active book, see *Range()*.
>
> New in version 0.9.0.

render_template(*template=None*, *output=None*, *book_settings=None*, ***data*)

> This function requires xlwings PRO.
>
> This is a convenience wrapper around *mysheet.render_template*
>
> Writes the values of all key word arguments to the `output` file according to the `template` and the variables contained in there (Jinja variable syntax). Following variable types are supported:
>
> strings, numbers, lists, simple dicts, NumPy arrays, Pandas DataFrames, pictures and Matplotlib/Plotly figures.

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

> New in version 0.24.4.

property screen_updating

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

New in version 0.3.3.

property selection

Returns the selected cells as Range.

New in version 0.9.0.

property startup_path

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

New in version 0.19.4.

property status_bar

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

New in version 0.20.0.

property version

Returns the Excel version number object.

Examples

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

Changed in version 0.9.0.

property visible

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

New in version 0.3.3.

34.2.3 Books

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

A collection of all *book* objects:

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

New in version 0.9.0.

property active

Returns the active Book.

add()

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

open(*fullname=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*, *json=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.

34.2.4 Book

class xlwings.**Book**(*fullname=None*, *update_links=None*, *read_only=None*, *format=None*,
password=None, *write_res_password=None*,
ignore_read_only_recommended=None, *origin=None*, *delimiter=None*,
editable=None, *notify=None*, *converter=None*, *add_to_mru=None*, *local=None*,
corrupt_load=None, *impl=None*, *json=None*, *mode=None*, *engine=None*)

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

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

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

```
>>> app = xw.App()   # or xw.apps[10559] (get the PIDs via xw.apps.keys())
>>> app.books['Book1']
```

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

> **Parameters**
>
>> - **fullname** (*str or path-like object, default None*) – Full path or name (incl. xlsx, xlsm etc.) of existing workbook or name of an unsaved workbook. Without a full path, it looks for the file in the current working directory.
>>
>> - **update_links** (*bool, default None*) – If this argument is omitted, the user is prompted to specify how links will be updated
>>
>> - **read_only** (*bool, default False*) – True to open workbook in read-only mode
>>
>> - **format** (*str*) – If opening a text file, this specifies the delimiter character
>>
>> - **password** (*str*) – Password to open a protected workbook
>>
>> - **write_res_password** (*str*) – Password to write to a write-reserved workbook

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

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

- **json** (*dict*) – A JSON object as delivered by the MS Office Scripts or Google Apps Script xlwings module but in a deserialized form, i.e., as dictionary.

  New in version 0.26.0.

- **mode** (*str, default None*) – Either "i" (interactive (default)) or "r" (read). In interactive mode, xlwings opens the workbook in Excel, i.e., Excel needs to be installed. In read mode, xlwings reads from the file directly, without requiring Excel to be installed. Read mode requires xlwings PRO.

  New in version 0.28.0.

activate(*steal_focus=False*)

Activates the book.

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

property api

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

New in version 0.9.0.

property app

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

New in version 0.9.0.

classmethod caller()

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

```python
import xlwings as xw

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

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

New in version 0.3.0.

close()

Closes the book without saving it.

New in version 0.1.1.

property fullname

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

json()

Returns a JSON serializable object as expected by the MS Office Scripts or Google Apps Script xlwings module. Only available with book objects that have been instantiated via `xw.Book(json=...)`.

New in version 0.26.0.

macro(*name*)

Runs a Sub or Function in Excel VBA.

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

Examples

This VBA function:

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

can be accessed like this:

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

See also: *App.macro()*

New in version 0.7.1.

property name
Returns the name of the book as str.

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

Changed in version 0.9.0.

render_template(***data*)
This method requires xlwings PRO.

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

New in version 0.25.0.

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

Examples

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

save(*path=None*, *password=None*)
Saves the Workbook. If a path is provided, this works like SaveAs() in Excel. If no path is specified and if the file hasn't been saved previously, it's saved in the current working directory with the current filename. Existing files are overwritten without prompting. To change the file type, provide the appropriate extension, e.g. to save myfile.xlsx in the xlsb format, provide myfile.xlsb as path.

> **Parameters**
> - **path** (*str or path-like object, default None*) – Path where you want to save the Book.
> - **password** (*str, default None*) – Protection password with max. 15 characters

New in version 0.25.1.

Example

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

New in version 0.3.1.

`property selection`

Returns the selected cells as Range.

New in version 0.9.0.

`set_mock_caller()`

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

Examples

```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.

`property sheet_names`

> **Returns**
>   **sheet_names** – List of sheet names in order of appearance.
>
> **Return type**
>   List

New in version 0.28.1.

property sheets

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

to_pdf(*path=None*, *include=None*, *exclude=None*, *layout=None*, *exclude_start_string='#'*, *show=False*, *quality='standard'*)

> Exports the whole Excel workbook or a subset of the sheets to a PDF file. If you want to print hidden sheets, you will need to list them explicitly under `include`.
>
> > **Parameters**
> >
> > - **path** (`str or path-like object, default None`) – Path to the PDF file, defaults to the same name as the workbook, in the same directory. For unsaved workbooks, it defaults to the current working directory instead.
> >
> > - **include** (`int or str or list, default None`) – Which sheets to include: provide a selection of sheets in the form of sheet indices (1-based like in Excel) or sheet names. Can be an int/str for a single sheet or a list of int/str for multiple sheets.
> >
> > - **exclude** (`int or str or list, default None`) – Which sheets to exclude: provide a selection of sheets in the form of sheet indices (1-based like in Excel) or sheet names. Can be an int/str for a single sheet or a list of int/str for multiple sheets.
> >
> > - **layout** (`str or path-like object, default None`) – This argument requires xlwings PRO.
> >
> >   Path to a PDF file on which the report will be printed. This is ideal for headers and footers as well as borderless printing of graphics/artwork. The PDF file either needs to have only 1 page (every report page uses the same layout) or otherwise needs the same amount of pages as the report (each report page is printed on the respective page in the layout PDF).
> >
> >   New in version 0.24.3.
> >
> > - **exclude_start_string** (str, default '#') – Sheet names that start with this character/string will not be printed.
> >
> >   New in version 0.24.4.
> >
> > - **show** (`bool, default False`) – Once created, open the PDF file with the default application.
> >
> >   New in version 0.24.6.
> >
> > - **quality** (str, default 'standard') – Quality of the PDF file. Can either be 'standard' or 'minimum'.
> >
> >   New in version 0.26.2.

Examples

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

See also *xlwings.Sheet.to_pdf()*

New in version 0.21.1.

34.2.5 PageSetup

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

>   property **api**

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

>   New in version 0.24.2.

>   property **print_area**

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

>   **Examples**

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

>   New in version 0.24.2.

34.2.6 Sheets

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

>   A collection of all *sheet* objects:

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

>   New in version 0.9.0.

>   property **active**

>   Returns the active Sheet.

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

> Creates a new Sheet and makes it the active sheet.
>
> > **Parameters**
> >
> > - **name** (`str, default None`) – Name of the new sheet. If None, will default to Excel's default name.
> >
> > - **before** (`Sheet, default None`) – An object that specifies the sheet before which the new sheet is added.
> >
> > - **after** (`Sheet, default None`) – An object that specifies the sheet after which the new sheet is added.
> >
> > **Returns**
> >
> > > **sheet** – Added sheet object
> >
> > **Return type**
> >
> > > *Sheet*

34.2.7 Sheet

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

> A sheet object is a member of the `sheets` collection:

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

> Changed in version 0.9.0.

activate()

> Activates the Sheet and returns it.

property api

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

autofit(*axis=None*)

> Autofits the width of either columns, rows or both on a whole Sheet.
>
> > **Parameters**
> >
> > > **axis** (`string, default None`) –
> > >
> > > - To autofit rows, use one of the following: `rows` or `r`
> > >
> > > - To autofit columns, use one of the following: `columns` or `c`

- To autofit rows and columns, provide no arguments

Examples

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

New in version 0.2.3.

property book

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

property cells

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

New in version 0.9.0.

property charts

See *Charts*

New in version 0.9.0.

clear()

Clears the content and formatting of the whole sheet.

clear_contents()

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

clear_formats()

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

New in version 0.26.2.

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

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

New in version 0.22.0.

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

Returns

Sheet object – The copied sheet

Return type

Sheet

Examples

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

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

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

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

delete()

Deletes the Sheet.

New in version 0.6.0.

property index

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

property name

Gets or sets the name of the Sheet.

property names

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

New in version 0.9.0.

property page_setup

Returns a PageSetup object.

New in version 0.24.2.

property pictures

See *Pictures*

New in version 0.9.0.

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

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

> New in version 0.9.0.

render_template(***data*)

> This method requires xlwings PRO.

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

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

> New in version 0.22.0.

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

> **Examples**

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

select()

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

> New in version 0.9.0.

property shapes

> See *Shapes*

> New in version 0.9.0.

property tables

> See *Tables*

> New in version 0.21.0.

to_html(*path=None*)

> Export a Sheet as HTML page.

> > **Parameters**
> > > **path** (*str or path-like, default None*) – Path where you want to save the HTML file. Defaults to Sheet name in the current working directory.

> New in version 0.28.1.

to_pdf(*path=None*, *layout=None*, *show=False*, *quality='standard'*)

> Exports the sheet to a PDF file.

Parameters

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

- **layout** (*str or path-like object, default None*) – This argument requires xlwings PRO.

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

  New in version 0.24.3.

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

  New in version 0.24.6.

- **quality** (str, default `'standard'`) – Quality of the PDF file. Can either be `'standard'` or `'minimum'`.

  New in version 0.26.2.

Examples

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

See also *xlwings.Book.to_pdf()*

New in version 0.22.3.

property used_range

Used Range of Sheet.

Returns

Return type
    xw.Range

New in version 0.13.0.

property visible

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

New in version 0.21.1.

34.2.8 Range

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

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

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

Examples

```python
import xlwings as xw
sheet1 = xw.Book("MyBook.xlsx").sheets[0]

sheet1.range("A1")
sheet1.range("A1:C3")
sheet1.range((1,1))
sheet1.range((1,1), (3,3))
sheet1.range("NamedRange")

# Or using index/slice notation
sheet1["A1"]
sheet1["A1:C3"]
sheet1[0, 0]
sheet1[0:4, 0:4]
sheet1["NamedRange"]
```

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

Adds a hyperlink to the specified Range (single Cell)

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

New in version 0.3.0.

property address

> Returns a string value that represents the range reference. Use `get_address()` to be able to provide paramaters.
>
> New in version 0.9.0.

property api

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

autofit()

> Autofits the width and height of all cells in the range.
>
> - To autofit only the width of the columns use `myrange.columns.autofit()`
>
> - To autofit only the height of the rows use `myrange.rows.autofit()`
>
> Changed in version 0.9.0.

clear()

> Clears the content and the formatting of a Range.

clear_contents()

> Clears the content of a Range but leaves the formatting.

clear_formats()

> Clears the format of a Range but leaves the content.
>
> New in version 0.26.2.

property color

> Gets and sets the background color of the specified Range.
>
> To set the color, either use an RGB tuple (`0, 0, 0`) or a hex string like `#efefef` or an Excel color constant. To remove the background, set the color to `None`, see Examples.
>
> > **Returns**
> >> **RGB**
> >
> > **Return type**
> >> tuple
>
> **Examples**

```
>>> import xlwings as xw
>>> wb = xw.Book()
>>> sheet1 = xw.sheets[0]
>>> sheet1.range('A1').color = (255, 255, 255)  # or '#ffffff'
>>> sheet1.range('A2').color
(255, 255, 255)
>>> sheet1.range('A2').color = None
```

(continues on next page)

```
>>> sheet1.range('A2').color is None
True
```

New in version 0.3.0.

property column

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

> **Returns**

> **Return type**
> > Integer

New in version 0.3.5.

property column_width

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

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

column_width must be in the range: 0 <= 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.

property columns

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

New in version 0.9.0.

copy(*destination=None*)

Copy a range to a destination range or clipboard.

> **Parameters**
> > **destination** (*xlwings.Range*) – xlwings Range to which the specified range will be copied. If omitted, the range is copied to the clipboard.

> **Returns**

> **Return type**
> > None

copy_picture(*appearance='screen'*, *format='picture'*)

Copies the range to the clipboard as picture.

Parameters

- **appearance** (*str, default 'screen'*) – Either 'screen' or 'printer'.

- **format** (*str, default 'picture'*) – Either 'picture' or 'bitmap'.

New in version 0.24.8.

property count

Returns the number of cells.

property current_region

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

Returns

Return type

Range object

delete(*shift=None*)

Deletes a cell or range of cells.

Parameters

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

Returns

Return type

None

end(*direction*)

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

Parameters

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

Examples

```
>>> import xlwings as xw
>>> wb = xw.Book()
>>> sheet1 = xw.sheets[0]
>>> sheet1.range('A1:B2').value = 1
>>> sheet1.range('A1').end('down')
<Range [Book1]Sheet1!$A$2>
>>> sheet1.range('B2').end('right')
<Range [Book1]Sheet1!$B$2>
```

New in version 0.9.0.

expand(*mode='table'*)

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

> **Parameters**
>> **mode** (`str, default 'table'`) – One of `'table'` (=down and right), `'down'`, `'right'`.
>
> **Returns**
>
> **Return type**
>> *Range*

Examples

```
>>> import xlwings as xw
>>> wb = xw.Book()
>>> sheet1 = wb.sheets[0]
>>> sheet1.range('A1').value = [[None, 1], [2, 3]]
>>> sheet1.range('A1').expand().address
$A$1:$B$2
>>> sheet1.range('A1').expand('right').address
$A$1:$B$1
```

New in version 0.9.0.

property formula

Gets or sets the formula for the given Range.

property formula2

Gets or sets the formula2 for the given Range.

property formula_array

Gets or sets an array formula for the given Range.

New in version 0.7.1.

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

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

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

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

> **Returns**

> **Return type**
>> str

Examples

```
>>> import xlwings as xw
>>> wb = xw.Book()
>>> sheet1 = wb.sheets[0]
>>> sheet1.range((1,1)).get_address()
'$A$1'
>>> sheet1.range((1,1)).get_address(False, False)
'A1'
>>> sheet1.range((1,1), (3,3)).get_address(True, False, True)
'Sheet1!A$1:C$3'
>>> sheet1.range((1,1), (3,3)).get_address(True, False, external=True)
'[Book1]Sheet1!A$1:C$3'
```

New in version 0.2.3.

property has_array

> True if the range is part of a legacy CSE Array formula and False otherwise.

property height

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

>> **Returns**

>> **Return type**
>>> float

> New in version 0.4.0.

property hyperlink

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

Examples

```
>>> import xlwings as xw
>>> wb = xw.Book()
>>> sheet1 = wb.sheets[0]
>>> sheet1.range('A1').value
'www.xlwings.org'
>>> sheet1.range('A1').hyperlink
'http://www.xlwings.org'
```

New in version 0.3.0.

insert(*shift=None*, *copy_origin='format_from_left_or_above'*)

>   Insert a cell or range of cells into the sheet.

>   **Parameters**

>   - **shift** (`str, default None`) – Use `right` or `down`. If omitted, Excel decides based on the shape of the range.

>   - **copy_origin** (`str, default format_from_left_or_above`) – Use `format_from_left_or_above` or `format_from_right_or_below`. Note that this is not supported on macOS.

>   **Returns**

>   **Return type**
>   None

property last_cell

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

>   **Returns**

>   **Return type**
>   *Range*

>   **Example**

```
>>> import xlwings as xw
>>> wb = xw.Book()
>>> sheet1 = wb.sheets[0]
>>> myrange = sheet1.range('A1:E4')
>>> myrange.last_cell.row, myrange.last_cell.column
(4, 5)
```

>   New in version 0.3.5.

property left

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

>   **Returns**

>   **Return type**
>   float

>   New in version 0.6.0.

merge(*across=False*)

>   Creates a merged cell from the specified Range object.

Parameters

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

property merge_area

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

property merge_cells

> Returns `True` if the Range contains merged cells, otherwise `False`

property name

> Sets or gets the name of a Range.
>
> New in version 0.4.0.

property note

> Returns a Note object. Before the introduction of threaded comments, a Note was called a Comment.
>
> New in version 0.24.2.

property number_format

> Gets and sets the number_format of a Range.

> **Examples**

```
>>> import xlwings as xw
>>> wb = xw.Book()
>>> sheet1 = wb.sheets[0]
>>> sheet1.range('A1').number_format
'General'
>>> sheet1.range('A1:C3').number_format = '0.00%'
>>> sheet1.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.
> >
> > - **err_to_str** (`Boolean, default False`) – If True, will include cell errors such as `#N/A` as strings. By default, they will be converted to `None`.
> >
> >   New in version 0.28.0.
>
> :keyword => For converter-specific options, see *Converters and Options*.:
>
> > **Returns**
> >
> > **Return type**
> > > Range object

paste(*paste=None*, *operation=None*, *skip_blanks=False*, *transpose=False*)

> Pastes a range from the clipboard into the specified range.
>
> **Parameters**
> > - **paste** (`str, default None`) – One of `all_merging_conditional_formats`, `all`, `all_except_borders`, `all_using_source_theme`, `column_widths`, `comments`, `formats`, `formulas`, `formulas_and_number_formats`, `validation`, `values`, `values_and_number_formats`.
> >
> > - **operation** (`str, default None`) – One of "add", "divide", "multiply", "subtract".
> >
> > - **skip_blanks** (`bool, default False`) – Set to True to skip over blank cells
> >
> > - **transpose** (`bool, default False`) – Set to True to transpose rows and columns.

> **Returns**
>
> **Return type**
> None

property raw_value

> Gets and sets the values directly as delivered from/accepted by the engine that s 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 num-
>   ber of rows in the range is unchanged).
>
> - **column_size** (`int > 0`) – The number of columns in the new range (if None,
>   the number of columns in the range is unchanged).
>
> **Returns**
> **Range object**
>
> **Return type**
> *Range*
>
> New in version 0.3.0.

property row

> Returns the number of the first row in the specified range. Read-only.
>
> **Returns**
>
> **Return type**
> Integer
>
> New in version 0.3.5.

property row_height

> Gets or sets the height, in points, of a Range. If all rows in the Range have the same height,
> returns the height. If rows in the Range have different heights, returns None.
>
> row_height must be in the range: 0 <= 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.

property rows

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

>   New in version 0.9.0.

select()

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

>   New in version 0.9.0.

property shape

>   Tuple of Range dimensions.

>   New in version 0.3.0.

property sheet

>   Returns the Sheet object to which the Range belongs.

>   New in version 0.9.0.

property size

>   Number of elements in the Range.

>   New in version 0.3.0.

property table

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

>   New in version 0.21.0.

to_pdf(*path=None*, *layout=None*, *show=None*, *quality='standard'*)

>   Exports the range as PDF.

>   **Parameters**

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

>   - **layout** (`str or path-like object, default None`) – This argument requires xlwings PRO.

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

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

>   - **quality** (str, default `'standard'`) – Quality of the PDF file. Can either be `'standard'` or `'minimum'`.

>   New in version 0.26.2.

to_png(*path=None*)

Exports the range as PNG picture.

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

New in version 0.24.8.

property top

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

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

New in version 0.6.0.

unmerge()

Separates a merged area into individual cells.

property value

Gets and sets the values for the given Range. See *xlwings.Range.options()* about how to set options, e.g., to transform it into a DataFrame or how to set a chunksize.

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

property width

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

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

New in version 0.4.0.

property wrap_text

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

New in version 0.23.2.

34.2.9 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

wb = xw.Book("MyBook.xlsx")
sheet1 = wb.sheets[0]
myrange = sheet1.range('A1:C4')

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

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

assert myrange.rows[2] == sheet1.range('A3:C3')
assert myrange.rows(2) == sheet1.range('A2:C2')

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

autofit()

> Autofits the height of the rows.

property count

> Returns the number of rows.

> New in version 0.9.0.

34.2.10 RangeColumns

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

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

Example

```python
import xlwings as xw

wb = xw.Book("MyFile.xlsx")
sheet1 = wb.sheets[0]
myrange = sheet1.range('A1:C4')
```

```
assert len(myrange.columns) == 3  # or myrange.columns.count

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

assert myrange.columns[2] == sheet1.range('C1:C4')
assert myrange.columns(2) == sheet1.range('B1:B4')

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

autofit()

> Autofits the width of the columns.

property count

> Returns the number of columns.

> New in version 0.9.0.

34.2.11 Shapes

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

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

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

> New in version 0.9.0.

property api

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

property count

> Returns the number of objects in the collection.

34.2.12 Shape

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

> The shape object is a member of the *shapes* collection:

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

Changed in version 0.9.0.

`activate()`

> Activates the shape.
>
> New in version 0.5.0.

`property api`

> Returns the native object (`pywin32` or `appscript` obj) of the engine being used.
>
> New in version 0.19.2.

`delete()`

> Deletes the shape.
>
> New in version 0.5.0.

`property height`

> Returns or sets the number of points that represent the height of the shape.
>
> New in version 0.5.0.

`property left`

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

`property name`

> Returns or sets the name of the shape.
>
> New in version 0.5.0.

`property parent`

> Returns the parent of the shape.
>
> New in version 0.9.0.

`scale_height`(*factor*, *relative_to_original_size=False*, *scale='scale_from_top_left'*)

> **factor**
> > [float] For example 1.5 to scale it up to 150%
>
> **relative_to_original_size**
> > [bool, optional] If `False`, it scales relative to current height (default). For `True` must be a picture or OLE object.
>
> **scale**
> > [str, optional] One of `scale_from_top_left` (default), `scale_from_bottom_right`, `scale_from_middle`
>
> New in version 0.19.2.

`scale_width`(*factor*, *relative_to_original_size=False*, *scale='scale_from_top_left'*)

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

relative_to_original_size
> [bool, optional] If `False`, it scales relative to current width (default). For `True` must be a picture or OLE object.

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

New in version 0.19.2.

property text
> Returns or sets the text of a shape.

> New in version 0.21.4.

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

> New in version 0.5.0.

property type
> Returns the type of the shape.

> New in version 0.9.0.

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

> New in version 0.5.0.

34.2.13 Charts

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

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

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

> New in version 0.9.0.

> **add**(*left=0*, *top=0*, *width=355*, *height=211*)
>> Creates a new chart on the specified sheet.
>>
>> **Parameters**
>>
>> - **left** (*float, default 0*) – left position in points
>>
>> - **top** (*float, default 0*) – top position in points
>>
>> - **width** (*float, default 355*) – width in points
>>
>> - **height** (*float, default 211*) – height in points

> **Returns**

> **Return type**
> *Chart*

Examples

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

property api

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

property count

Returns the number of objects in the collection.

34.2.14 Chart

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

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

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

property api

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

New in version 0.9.0.

property chart_type

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

3d_area, 3d_area_stacked, 3d_area_stacked_100, 3d_bar_clustered, 3d_bar_stacked, 3d_bar_stacked_100, 3d_column, 3d_column_clustered, 3d_column_stacked, 3d_column_stacked_100, 3d_line, 3d_pie, 3d_pie_exploded, area, area_stacked, area_stacked_100, bar_clustered, bar_of_pie, bar_stacked, bar_stacked_100, bubble, bubble_3d_effect, column_clustered, column_stacked, column_stacked_100, combination, cone_bar_clustered, cone_bar_stacked, cone_bar_stacked_100, cone_col, cone_col_clustered, cone_col_stacked, cone_col_stacked_100, cylinder_bar_clustered,

cylinder_bar_stacked, cylinder_bar_stacked_100, cylinder_col, cylinder_col_clustered, cylinder_col_stacked, cylinder_col_stacked_100, doughnut, doughnut_exploded, line, line_markers, line_markers_stacked, line_markers_stacked_100, line_stacked, line_stacked_100, pie, pie_exploded, pie_of_pie, pyramid_bar_clustered, pyramid_bar_stacked, pyramid_bar_stacked_100, pyramid_col, pyramid_col_clustered, pyramid_col_stacked, pyramid_col_stacked_100, radar, radar_filled, radar_markers, stock_hlc, stock_ohlc, stock_vhlc, stock_vohlc, surface, surface_top_view, surface_top_view_wireframe, surface_wireframe, xy_scatter, xy_scatter_lines, xy_scatter_lines_no_markers, xy_scatter_smooth, xy_scatter_smooth_no_markers

New in version 0.1.1.

delete()

Deletes the chart.

property height

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

property left

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

property name

Returns or sets the name of the chart.

property parent

Returns the parent of the chart.

New in version 0.9.0.

set_source_data(*source*)

Sets the source data range for the chart.

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

to_pdf(*path=None*, *show=None*, *quality='standard'*)

Exports the chart as PDF.

> **Parameters**
> - **path** (`str or path-like, default None`) – Path where you want to store the pdf. Defaults to the name of the chart in the same directory as the Excel file if the Excel file is stored and to the current working directory otherwise.
> - **show** (`bool, default False`) – Once created, open the PDF file with the default application.
> - **quality** (str, default `'standard'`) – Quality of the PDF file. Can either be `'standard'` or `'minimum'`.

New in version 0.26.2.

to_png(*path=None*)

    Exports the chart as PNG picture.

        Parameters

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

    New in version 0.24.8.

property top

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

property width

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

34.2.15 Pictures

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

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

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

    New in version 0.9.0.

add(*image*, *link_to_file=False*, *save_with_document=True*, *left=None*, *top=None*, *width=None*, *height=None*, *name=None*, *update=False*, *scale=None*, *format=None*, *anchor=None*, *export_options=None*)

    Adds a picture to the specified sheet.

        Parameters

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

            • **left** (`float, default None`) – Left position in points, defaults to 0. If you use `top`/`left`, you must not provide a value for `anchor`.

            • **top** (`float, default None`) – Top position in points, defaults to 0. If you use `top`/`left`, you must not provide a value for `anchor`.

            • **width** (`float, default None`) – Width in points. Defaults to original width.

            • **height** (`float, default None`) – Height in points. Defaults to original height.

            • **name** (`str, default None`) – Excel picture name. Defaults to Excel standard name if not provided, e.g., 'Picture 1'.

- **update** (*bool, default False*) – Replace an existing picture with the same name. Requires `name` to be set.

- **scale** (*float, default None*) – Scales your picture by the provided factor.

- **format** (*str, default None*) – Only used if image is a Matplotlib or Plotly plot. By default, the plot is inserted in the "png" format, but you may want to change this to a vector-based format like "svg" on Windows (may require Microsoft 365) or "eps" on macOS for better print quality. If you use `'vector'`, it will be using `'svg'` on Windows and `'eps'` on macOS. To find out which formats your version of Excel supports, see: https://support.microsoft.com/en-us/topic/support-for-eps-images-has-been-turned-off-in-office-a069d664-4bcf-415e-a1b5-cbb0c334a840

- **anchor** (*xw.Range, default None*) – The xlwings Range object of where you want to insert the picture. If you use `anchor`, you must not provide values for `top`/`left`.

  New in version 0.24.3.

- **export_options** (*dict, default None*) – For Matplotlib plots, this dictionary is passed on to `image.savefig()` with the following defaults: `{"bbox_inches": "tight", "dpi": 200}`, so if you want to leave the picture uncropped and increase dpi to 300, use: `export_options={"dpi": 300}`. For Plotly, the options are passed to `write_image()`.

  New in version 0.27.7.

Returns

Return type
    Picture

Examples

1. Picture

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

2. Matplotlib

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

property api

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

property count

> Returns the number of objects in the collection.

34.2.16 Picture

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

> The picture object is a member of the *pictures* collection:

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

> Changed in version 0.9.0.

property api

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

delete()

> Deletes the picture.
>
> New in version 0.5.0.

property height

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

property left

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

property lock_aspect_ratio

> `True` will keep the original proportion, `False` will allow you to change height and width independently of each other (read/write).
>
> New in version 0.24.0.

property name

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

property parent

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

property top

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

New in version 0.5.0.

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

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

> **Parameters**
>
> - **image** (*str or path-like object or matplotlib.figure.Figure*) – Either a filepath or a Matplotlib figure object.
>
> - **format** (*str, default None*) – See under `Pictures.add()`
>
> - **export_options** (*dict, default None*) – See under `Pictures.add()`

New in version 0.5.0.

property width

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

New in version 0.5.0.

34.2.17 Names

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

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

```
>>> import xlwings as xw
>>> book = xw.books['Book1']  # book scope and sheet scope
>>> book.names
[<Name 'MyName': =Sheet1!$A$3>]
>>> book.sheets[0].names  # sheet scope only
```

New in version 0.9.0.

add(*name*, *refers_to*)

Defines a new name for a range of cells.

> **Parameters**
>
> - **name** (*str*) – Specifies the text to use as the name. Names cannot include spaces and cannot be formatted as cell references.
>
> - **refers_to** (*str*) – Describes what the name refers to, in English, using A1-style notation.
>
> **Returns**
>
> **Return type**
> > *Name*

New in version 0.9.0.

property **api**

> Returns the native object (`pywin32` or `appscript` obj) of the engine beingused.
>
> New in version 0.9.0.

property **count**

> Returns the number of objects in the collection.

34.2.18 Name

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

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

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

New in version 0.9.0.

property **api**

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

delete()

> Deletes the name.
>
> New in version 0.9.0.

property **name**

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

property **refers_to**

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

property **refers_to_range**

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

34.2.19 Note

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

> **property api**
>> Returns the native object (`pywin32` or `appscript` obj) of the engine being used.
>>
>> New in version 0.24.2.
>
> **delete()**
>> Delete the note.
>>
>> New in version 0.24.2.
>
> **property text**
>> Gets or sets the text of a note. Keep in mind that the note must already exist!
>>
>> **Examples**
>>
>> ```
>> >>> sheet = xw.Book(...).sheets[0]
>> >>> sheet['A1'].note.text = 'mynote'
>> >>> sheet['A1'].note.text
>> >>> 'mynote'
>> ```
>>
>> New in version 0.24.2.

34.2.20 Tables

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

> A collection of all *table* objects on the specified sheet:
>
> ```
> >>> import xlwings as xw
> >>> xw.books['Book1'].sheets[0].tables
> Tables([<Table 'Table1' in <Sheet [Book11]Sheet1>>,
>         <Table 'Table2' in <Sheet [Book11]Sheet1>>])
> ```
>
> New in version 0.21.0.
>
> **add**(*source=None*, *name=None*, *source_type=None*, *link_source=None*, *has_headers=True*, *destination=None*, *table_style_name='TableStyleMedium2'*)
>> Creates a Table to the specified sheet.
>>
>> **Parameters**
>>> - **source** (*xlwings range, default None*) – An xlwings range object, representing the data source.
>>>
>>> - **name** (*str, default None*) – The name of the Table. By default, it uses the autogenerated name that is assigned by Excel.

- **source_type** (*str, default None*) – This currently defaults to xlSrcRange, i.e. expects an xlwings range object. No other options are allowed at the moment.

- **link_source** (*bool, default None*) – Currently not implemented as this is only in case `source_type` is `xlSrcExternal`.

- **has_headers** (*bool or str, default True*) – Indicates whether the data being imported has column labels. Defaults to `True`. Possible values: `True`, `FAlse`, `'guess'`

- **destination** (*xlwings range, default None*) – Currently not implemented as this is used in case `source_type` is `xlSrcExternal`.

- **table_style_name** (*str, default 'TableStyleMedium2'*) – Possible strings: `'TableStyleLightN''` (where N is 1-21), `'TableStyleMediumN'` (where N is 1-28), *'TableStyleDarkN'`* (where N is 1-11)

> **Returns**
>
> **Return type**
>   *Table*

Examples

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

34.2.21 Table

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

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

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

> New in version 0.21.0.

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

property data_body_range

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

property display_name

Returns or sets the display name for the specified Table object

property header_row_range

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

property insert_row_range

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

property name

Returns or sets the name of the Table.

property parent

Returns the parent of the table.

property range

Returns an xlwings range object of the table.

resize(*range*)

Resize a Table by providing an xlwings range object

New in version 0.24.4.

property show_autofilter

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

property show_headers

Show or hide the header (read/write)

property show_table_style_column_stripes

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

property show_table_style_first_column

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

property show_table_style_last_column

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

property show_table_style_row_stripes

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

property show_totals

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

property table_style

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

property **totals_row_range**

>   Returns an xlwings range object representing the Total row

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

>   Updates the Excel table with the provided data. Currently restricted to DataFrames.
>
>   Changed in version 0.24.0.
>
>   >   **Parameters**
>   >
>   >   - **data** (*pandas DataFrame*) – Currently restricted to pandas DataFrames.
>   >
>   >   - **index** (*bool, default True*) – Whether or not the index of a pandas DataFrame should be written to the Excel table.
>   >
>   >   **Returns**
>   >
>   >   **Return type**
>   >   > *Table*

Examples

```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)
```

34.2.22 Font

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

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

- mysheet['A1'].font

- mysheet.shapes[0].font

New in version 0.23.0.

property api

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

    New in version 0.23.0.

property bold

    Returns or sets the bold property (boolean).

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

    New in version 0.23.0.

property color

    Returns or sets the color property (tuple).

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

    New in version 0.23.0.

property italic

    Returns or sets the italic property (boolean).

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

    New in version 0.23.0.

property name

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

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

    New in version 0.23.0.

property size

Returns or sets the size (float).

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

New in version 0.23.0.

34.2.23 Characters

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

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

- mysheet['A1'].characters

- mysheet.shapes[0].characters

Note: On macOS, characters are currently not supported due to bugs/lack of support in Apple-Script.

New in version 0.23.0.

property api

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

New in version 0.23.0.

property font

Returns or sets the text property of a characters object.

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

New in version 0.23.0.

property text

Returns or sets the text property of a characters object.

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

New in version 0.23.0.

34.2.24 Markdown

34.2.25 MarkdownStyle

34.3 UDF decorators

xlwings.**func**(*category='xlwings'*, *volatile=False*, *call_in_wizard=True*)

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

34.3.1 Arguments

category

> [int or str, default "xlwings"] 1-14 represent built-in categories, for user-defined categories use strings
>
> New in version 0.10.3.

volatile

> [bool, default False] Marks a user-defined function as volatile. A volatile function must be recalculated whenever calculation occurs in any cells on the worksheet. A nonvolatile function is recalculated only when the input variables change. This method has no effect if it's not inside a user-defined function used to calculate a worksheet cell.
>
> New in version 0.10.3.

call_in_wizard

> [bool, default True] Set to False to suppress the function call in the function wizard.
>
> New in version 0.10.3.

xlwings.**sub**()

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

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

> Apply converters and options to arguments, see also *Range.options()*.
>
> **Examples:**
>
> Convert `x` into a 2-dimensional numpy array:

```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) Suppress 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.

34.4 Reports

REST API

Note: This is an experimental feature and may be removed in the future.

New in version 0.13.0.

35.1 Quickstart

xlwings offers an easy way to expose an Excel workbook via REST API both on Windows and macOS. This can be useful when you have a workbook running on a single computer and want to access it from another computer. Or you can build a Linux based web app that can interact with a legacy Excel application while you are in the progress of migrating the Excel functionality into your web app (if you need help with that, give us a shout).

You can run the REST API server from a command prompt or terminal as follows (this requires Flask>=1.0, so make sure to `pip install Flask`):

```
xlwings restapi run
```

Then perform a GET request e.g. via PowerShell on Windows or Terminal on Mac (while having an unsaved "Book1" open). Note that you need to run the server and the GET request from two separate terminals (or you can use something more convenient like Postman or Insomnia for testing the API):

```
$ curl "http://127.0.0.1:5000/book/book1/sheets/0/range/A1:B2"
{
  "address": "$A$1:$B$2",
  "color": null,
  "column": 1,
  "column_width": 10.0,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    [
      "1",
      "2"
```

```
      ],
      [
        "3",
        "4"
      ]
    ],
    "formula_array": null,
    "height": 32.0,
    "last_cell": "$B$2",
    "left": 0.0,
    "name": null,
    "number_format": "General",
    "row": 1,
    "row_height": 16.0,
    "shape": [
      2,
      2
    ],
    "size": 4,
    "top": 0.0,
    "value": [
      [
        1.0,
        2.0
      ],
      [
        3.0,
        4.0
      ]
    ],
    "width": 130.0
}
```

In the command prompt where your server is running, press `Ctrl-C` to shut it down again.

The xlwings REST API is a thin wrapper around the *Python API* which makes it very easy if you have worked previously with xlwings. It also means that the REST API does require the Excel application to be up and running which makes it a great choice if the data in your Excel workbook is constantly changing as the REST API will always deliver the current state of the workbook without the need of saving it first.

Note: Currently, we only provide the GET methods to read the workbook. If you are also interested in the POST methods to edit the workbook, let us know via GitHub issues. Some other things will also need improvement, most notably exception handling.

35.2 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):

```python
from xlwings.rest.api import api
from waitress import serve
serve(wsgiapp, host='127.0.0.1', port=5000)
```

35.3 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`.

35.4 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.

35.5 Endpoint overview

End-point	Corresponds to	Short Description
Book	*Book*	Finds your workbook across all open instances of Excel and will open it if it can't find it
Books	*Books*	Books collection of the active Excel instance
Apps	*Apps*	This allows you to specify the Excel instance you want to work with

35.6 Endpoint details

35.6.1 /book

GET /book/<fullname_or_name>

Example response:

```
{
  "app": 1104,
  "fullname": "C:\\Users\\felix\\DEV\\xlwings\\scripts\\Book1.xlsx",
  "name": "Book1.xlsx",
  "names": [
    "Sheet1!myname1",
    "myname2"
  ],
  "selection": "Sheet2!$A$1",
  "sheets": [
    "Sheet1",
    "Sheet2"
  ]
}
```

GET /book/<fullname_or_name>/names

Example response:

```
{
  "names": [
    {
      "name": "Sheet1!myname1",
      "refers_to": "=Sheet1!$B$2:$C$3"
    },
    {
      "name": "myname2",
      "refers_to": "=Sheet1!$A$1"
    }
  ]
}
```

GET /book/<fullname_or_name>/names/<name>

Example response:

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

`GET /book/<fullname_or_name>/names/<name>/range`

Example response:

```
{
  "address": "$A$1",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 1,
  "current_region": "$A$1:$B$2",
  "formula": "=1+1.1",
  "formula_array": "=1+1,1",
  "height": 14.25,
  "last_cell": "$A$1",
  "left": 0.0,
  "name": "myname2",
  "number_format": "General",
  "row": 1,
  "row_height": 14.3,
  "shape": [
    1,
    1
  ],
  "size": 1,
  "top": 0.0,
  "value": 2.1,
  "width": 51.0
}
```

`GET /book/<fullname_or_name>/sheets`

Example response:

```
{
  "sheets": [
    {
      "charts": [
        "Chart 1"
      ],
      "name": "Sheet1",
      "names": [
        "Sheet1!myname1"
      ],
      "pictures": [
        "Picture 3"
      ],
      "shapes": [
```

```
        "Chart 1",
        "Picture 3"
      ],
      "used_range": "$A$1:$B$2"
    },
    {
      "charts": [],
      "name": "Sheet2",
      "names": [],
      "pictures": [],
      "shapes": [],
      "used_range": "$A$1"
    }
  ]
}
```

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

Example response:

```
{
  "charts": [
    "Chart 1"
  ],
  "name": "Sheet1",
  "names": [
    "Sheet1!myname1"
  ],
  "pictures": [
    "Picture 3"
  ],
  "shapes": [
    "Chart 1",
    "Picture 3"
  ],
  "used_range": "$A$1:$B$2"
}
```

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

Example response:

```
{
  "charts": [
    {
      "chart_type": "line",
      "height": 211.0,
```

```
        "left": 0.0,
        "name": "Chart 1",
        "top": 0.0,
        "width": 355.0
    }
  ]
}
```

GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>/charts/<chart_name_or_ix>

Example response:

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

GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>/names

Example response:

```
{
  "names": [
    {
      "name": "Sheet1!myname1",
      "refers_to": "=Sheet1!$B$2:$C$3"
    }
  ]
}
```

GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>/names/<sheet_scope_name>

Example response:

```
{
  "name": "Sheet1!myname1",
  "refers_to": "=Sheet1!$B$2:$C$3"
}
```

GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>/names/<sheet_scope_name>/ range

Example response:

```
{
  "address": "$B$2:$C$3",
  "color": null,
  "column": 2,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    [
      "",
      ""
    ],
    [
      "",
      ""
    ]
  ],
  "formula_array": "",
  "height": 28.5,
  "last_cell": "$C$3",
  "left": 51.0,
  "name": "Sheet1!myname1",
  "number_format": "General",
  "row": 2,
  "row_height": 14.3,
  "shape": [
    2,
    2
  ],
  "size": 4,
  "top": 14.25,
  "value": [
    [
      null,
      null
    ],
    [
      null,
      null
    ]
  ],
  "width": 102.0
}
```

GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>/pictures

Example response:

```
{
  "pictures": [
    {
      "height": 100.0,
      "left": 0.0,
      "name": "Picture 3",
      "top": 0.0,
      "width": 100.0
    }
  ]
}
```

GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>/pictures/
<picture_name_or_ix>

Example response:

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

GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>/range

Example response:

```
{
  "address": "$A$1:$B$2",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    [
      "=1+1.1",
      "a string"
    ],
    [
      "43395.0064583333",
      ""
    ]
  ],
  "formula_array": null,
```

```
  "height": 28.5,
  "last_cell": "$B$2",
  "left": 0.0,
  "name": null,
  "number_format": null,
  "row": 1,
  "row_height": 14.3,
  "shape": [
    2,
    2
  ],
  "size": 4,
  "top": 0.0,
  "value": [
    [
      2.1,
      "a string"
    ],
    [
      "Mon, 22 Oct 2018 00:09:18 GMT",
      null
    ]
  ],
  "width": 102.0
}
```

GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>/range/<address>

Example response:

```
{
  "address": "$A$1:$B$2",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    [
      "=1+1.1",
      "a string"
    ],
    [
      "43395.0064583333",
      ""
    ]
```

```
  ],
  "formula_array": null,
  "height": 28.5,
  "last_cell": "$B$2",
  "left": 0.0,
  "name": null,
  "number_format": null,
  "row": 1,
  "row_height": 14.3,
  "shape": [
    2,
    2
  ],
  "size": 4,
  "top": 0.0,
  "value": [
    [
      2.1,
      "a string"
    ],
    [
      "Mon, 22 Oct 2018 00:09:18 GMT",
      null
    ]
  ],
  "width": 102.0
}
```

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

Example response:

```
{
  "shapes": [
    {
      "height": 211.0,
      "left": 0.0,
      "name": "Chart 1",
      "top": 0.0,
      "type": "chart",
      "width": 355.0
    },
    {
      "height": 100.0,
      "left": 0.0,
      "name": "Picture 3",
```

```
        "top": 0.0,
        "type": "picture",
        "width": 100.0
      }
    ]
}
```

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

Example response:

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

35.6.2 /books

GET /books

Example response:

```
{
  "books": [
    {
      "app": 1104,
      "fullname": "Book1",
      "name": "Book1",
      "names": [],
      "selection": "Sheet2!$A$1",
      "sheets": [
        "Sheet1"
      ]
    },
    {
      "app": 1104,
      "fullname": "C:\\Users\\felix\\DEV\\xlwings\\scripts\\Book1.xlsx",
      "name": "Book1.xlsx",
      "names": [
        "Sheet1!myname1",
        "myname2"
```

```
    ],
    "selection": "Sheet2!$A$1",
    "sheets": [
      "Sheet1",
      "Sheet2"
    ]
  },
  {
    "app": 1104,
    "fullname": "Book4",
    "name": "Book4",
    "names": [],
    "selection": "Sheet2!$A$1",
    "sheets": [
      "Sheet1"
    ]
  }
  ]
}
```

GET /books/<book_name_or_ix>

Example response:

```
{
  "app": 1104,
  "fullname": "C:\\Users\\felix\\DEV\\xlwings\\scripts\\Book1.xlsx",
  "name": "Book1.xlsx",
  "names": [
    "Sheet1!myname1",
    "myname2"
  ],
  "selection": "Sheet2!$A$1",
  "sheets": [
    "Sheet1",
    "Sheet2"
  ]
}
```

GET /books/<book_name_or_ix>/names

Example response:

```
{
  "names": [
    {
      "name": "Sheet1!myname1",
```

```
      "refers_to": "=Sheet1!$B$2:$C$3"
    },
    {
      "name": "myname2",
      "refers_to": "=Sheet1!$A$1"
    }
  ]
}
```

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

Example response:

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

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

Example response:

```
{
  "address": "$A$1",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 1,
  "current_region": "$A$1:$B$2",
  "formula": "=1+1.1",
  "formula_array": "=1+1,1",
  "height": 14.25,
  "last_cell": "$A$1",
  "left": 0.0,
  "name": "myname2",
  "number_format": "General",
  "row": 1,
  "row_height": 14.3,
  "shape": [
    1,
    1
  ],
  "size": 1,
  "top": 0.0,
  "value": 2.1,
  "width": 51.0
}
```

`GET /books/<book_name_or_ix>/sheets`

Example response:

```
{
  "sheets": [
    {
      "charts": [
        "Chart 1"
      ],
      "name": "Sheet1",
      "names": [
        "Sheet1!myname1"
      ],
      "pictures": [
        "Picture 3"
      ],
      "shapes": [
        "Chart 1",
        "Picture 3"
      ],
      "used_range": "$A$1:$B$2"
    },
    {
      "charts": [],
      "name": "Sheet2",
      "names": [],
      "pictures": [],
      "shapes": [],
      "used_range": "$A$1"
    }
  ]
}
```

`GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>`

Example response:

```
{
  "charts": [
    "Chart 1"
  ],
  "name": "Sheet1",
  "names": [
    "Sheet1!myname1"
  ],
  "pictures": [
    "Picture 3"
```

(continues on next page)

```
  ],
  "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_type": "line",
      "height": 211.0,
      "left": 0.0,
      "name": "Chart 1",
      "top": 0.0,
      "width": 355.0
    }
  ]
}
```

GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/charts/<chart_name_or_ix>

Example response:

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

GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/names

Example response:

```
{
  "names": [
    {
      "name": "Sheet1!myname1",
      "refers_to": "=Sheet1!$B$2:$C$3"
```

```
    }
  ]
}
```

GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/names/<sheet_scope_name>

Example response:

```
{
  "name": "Sheet1!myname1",
  "refers_to": "=Sheet1!$B$2:$C$3"
}
```

GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/names/<sheet_scope_name>/
range

Example response:

```
{
  "address": "$B$2:$C$3",
  "color": null,
  "column": 2,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    [
      "",
      ""
    ],
    [
      "",
      ""
    ]
  ],
  "formula_array": "",
  "height": 28.5,
  "last_cell": "$C$3",
  "left": 51.0,
  "name": "Sheet1!myname1",
  "number_format": "General",
  "row": 2,
  "row_height": 14.3,
  "shape": [
    2,
    2
  ],
```

```
    "size": 4,
    "top": 14.25,
    "value": [
      [
        null,
        null
      ],
      [
        null,
        null
      ]
    ],
    "width": 102.0
}
```

GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/pictures

Example response:

```
{
  "pictures": [
    {
      "height": 100.0,
      "left": 0.0,
      "name": "Picture 3",
      "top": 0.0,
      "width": 100.0
    }
  ]
}
```

GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/pictures/
<picture_name_or_ix>

Example response:

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

GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/range

Example response:

```
{
  "address": "$A$1:$B$2",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    [
      "=1+1.1",
      "a string"
    ],
    [
      "43395.0064583333",
      ""
    ]
  ],
  "formula_array": null,
  "height": 28.5,
  "last_cell": "$B$2",
  "left": 0.0,
  "name": null,
  "number_format": null,
  "row": 1,
  "row_height": 14.3,
  "shape": [
    2,
    2
  ],
  "size": 4,
  "top": 0.0,
  "value": [
    [
      2.1,
      "a string"
    ],
    [
      "Mon, 22 Oct 2018 00:09:18 GMT",
      null
    ]
  ],
  "width": 102.0
}
```

GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/range/<address>

Example response:

```
{
  "address": "$A$1:$B$2",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    [
      "=1+1.1",
      "a string"
    ],
    [
      "43395.0064583333",
      ""
    ]
  ],
  "formula_array": null,
  "height": 28.5,
  "last_cell": "$B$2",
  "left": 0.0,
  "name": null,
  "number_format": null,
  "row": 1,
  "row_height": 14.3,
  "shape": [
    2,
    2
  ],
  "size": 4,
  "top": 0.0,
  "value": [
    [
      2.1,
      "a string"
    ],
    [
      "Mon, 22 Oct 2018 00:09:18 GMT",
      null
    ]
  ],
  "width": 102.0
}
```

GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/shapes

Example response:

```
{
  "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
}
```

35.6.3 /apps

GET /apps

Example response:

```
{
  "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:

```
{
  "books": [
    "Book1",
    "C:\\Users\\felix\\DEV\\xlwings\\scripts\\Book1.xlsx",
    "Book4"
  ],
  "calculation": "automatic",
  "display_alerts": true,
  "pid": 1104,
  "screen_updating": true,
  "selection": "[Book1.xlsx]Sheet2!$A$1",
  "version": "16.0",
  "visible": true
}
```

`GET /apps/<pid>/books`

Example response:

```
{
  "books": [
    {
      "app": 1104,
      "fullname": "Book1",
      "name": "Book1",
      "names": [],
      "selection": "Sheet2!$A$1",
      "sheets": [
        "Sheet1"
      ]
    },
    {
      "app": 1104,
      "fullname": "C:\\Users\\felix\\DEV\\xlwings\\scripts\\Book1.xlsx",
      "name": "Book1.xlsx",
      "names": [
        "Sheet1!myname1",
        "myname2"
      ],
      "selection": "Sheet2!$A$1",
      "sheets": [
        "Sheet1",
        "Sheet2"
      ]
    },
    {
      "app": 1104,
      "fullname": "Book4",
      "name": "Book4",
      "names": [],
      "selection": "Sheet2!$A$1",
      "sheets": [
        "Sheet1"
      ]
    }
  ]
}
```

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

Example response:

```
{
  "app": 1104,
  "fullname": "C:\\Users\\felix\\DEV\\xlwings\\scripts\\Book1.xlsx",
```

```
    "name": "Book1.xlsx",
    "names": [
      "Sheet1!myname1",
      "myname2"
    ],
    "selection": "Sheet2!$A$1",
    "sheets": [
      "Sheet1",
      "Sheet2"
    ]
}
```

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

Example response:

```
{
  "names": [
    {
      "name": "Sheet1!myname1",
      "refers_to": "=Sheet1!$B$2:$C$3"
    },
    {
      "name": "myname2",
      "refers_to": "=Sheet1!$A$1"
    }
  ]
}
```

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

Example response:

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

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

Example response:

```
{
  "address": "$A$1",
  "color": null,
  "column": 1,
  "column_width": 8.47,
```

```
  "count": 1,
  "current_region": "$A$1:$B$2",
  "formula": "=1+1.1",
  "formula_array": "=1+1,1",
  "height": 14.25,
  "last_cell": "$A$1",
  "left": 0.0,
  "name": "myname2",
  "number_format": "General",
  "row": 1,
  "row_height": 14.3,
  "shape": [
    1,
    1
  ],
  "size": 1,
  "top": 0.0,
  "value": 2.1,
  "width": 51.0
}
```

GET /apps/<pid>/books/<book_name_or_ix>/sheets

Example response:

```
{
  "sheets": [
    {
      "charts": [
        "Chart 1"
      ],
      "name": "Sheet1",
      "names": [
        "Sheet1!myname1"
      ],
      "pictures": [
        "Picture 3"
      ],
      "shapes": [
        "Chart 1",
        "Picture 3"
      ],
      "used_range": "$A$1:$B$2"
    },
    {
      "charts": [],
```

```
        "name": "Sheet2",
        "names": [],
        "pictures": [],
        "shapes": [],
        "used_range": "$A$1"
    }
  ]
}
```

GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>

Example response:

```
{
  "charts": [
    "Chart 1"
  ],
  "name": "Sheet1",
  "names": [
    "Sheet1!myname1"
  ],
  "pictures": [
    "Picture 3"
  ],
  "shapes": [
    "Chart 1",
    "Picture 3"
  ],
  "used_range": "$A$1:$B$2"
}
```

GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/charts

Example response:

```
{
  "charts": [
    {
      "chart_type": "line",
      "height": 211.0,
      "left": 0.0,
      "name": "Chart 1",
      "top": 0.0,
      "width": 355.0
    }
  ]
}
```

GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/charts/
<chart_name_or_ix>

Example response:

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

GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/names

Example response:

```
{
  "names": [
    {
      "name": "Sheet1!myname1",
      "refers_to": "=Sheet1!$B$2:$C$3"
    }
  ]
}
```

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

Example response:

```
{
  "name": "Sheet1!myname1",
  "refers_to": "=Sheet1!$B$2:$C$3"
}
```

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

Example response:

```
{
  "address": "$B$2:$C$3",
  "color": null,
  "column": 2,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
```

(continues on next page)

```
  "formula": [
    [
      "",
      ""
    ],
    [
      "",
      ""
    ]
  ],
  "formula_array": "",
  "height": 28.5,
  "last_cell": "$C$3",
  "left": 51.0,
  "name": "Sheet1!myname1",
  "number_format": "General",
  "row": 2,
  "row_height": 14.3,
  "shape": [
    2,
    2
  ],
  "size": 4,
  "top": 14.25,
  "value": [
    [
      null,
      null
    ],
    [
      null,
      null
    ]
  ],
  "width": 102.0
}
```

GET /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$2",
  "formula": [
    [
      "=1+1.1",
      "a string"
    ],
    [
      "43395.0064583333",
      ""
    ]
  ],
  "formula_array": null,
  "height": 28.5,
  "last_cell": "$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 /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/range/<address>

Example response:

```
{
  "address": "$A$1:$B$2",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    [
      "=1+1.1",
      "a string"
    ],
    [
      "43395.0064583333",
      ""
    ]
  ],
  "formula_array": null,
  "height": 28.5,
  "last_cell": "$B$2",
  "left": 0.0,
```

```
    "name": null,
    "number_format": null,
    "row": 1,
    "row_height": 14.3,
    "shape": [
      2,
      2
    ],
    "size": 4,
    "top": 0.0,
    "value": [
      [
        2.1,
        "a string"
      ],
      [
        "Mon, 22 Oct 2018 00:09:18 GMT",
        null
      ]
    ],
    "width": 102.0
}
```

GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/shapes

Example response:

```
{
  "shapes": [
    {
      "height": 211.0,
      "left": 0.0,
      "name": "Chart 1",
      "top": 0.0,
      "type": "chart",
      "width": 355.0
    },
    {
      "height": 100.0,
      "left": 0.0,
      "name": "Picture 3",
      "top": 0.0,
      "type": "picture",
      "width": 100.0
    }
  ]
```

```
}
```

GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/shapes/
<shape_name_or_ix>

Example response:

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