xlwings - Make Excel Fly!

Release dev

Zoomer Analytics LLC

Nov 23, 2020
# Getting Started

## 1 Video course

## 2 Installation

2.1 Prerequisites .................................................. 3
2.2 Installation ....................................................... 3
2.3 Add-in ............................................................. 4
2.4 Dependencies .................................................... 4
2.5 How to activate xlwings PRO ................................ 4
2.6 Optional Dependencies ......................................... 5
2.7 Update ............................................................ 5
2.8 Uninstall .......................................................... 5

## 3 Quickstart

3.1 1. Scripting: Automate/interact with Excel from Python ................................. 7
3.2 2. Macros: Call Python from Excel ........................................ 8
3.3 3. UDFs: User Defined Functions (Windows only) ...................................... 9

## 4 Connect to a Book

4.1 Python to Excel .................................................. 11
4.2 Excel to Python (RunPython) ...................................... 12
4.3 User Defined Functions (UDFs) ...................................... 12

## 5 Syntax Overview

5.1 Active Objects .................................................. 13
5.2 Full qualification ................................................ 14
5.3 Range indexing/slicing .......................................... 14
5.4 Range Shortcuts ................................................ 14
5.5 Object Hierarchy ................................................ 15

## 6 Data Structures Tutorial

6.1 Single Cells ..................................................... 17
6.2 Lists ............................................................... 18
6.3 Range expanding ................................................ 19
6.4 NumPy arrays .................................................. 19
6.5 Pandas DataFrames ............................................ 19
6.6 Pandas Series ................................................... 20

7 Add-in & Settings .................................................. 21
7.1 Run main ......................................................... 21
7.2 Installation ....................................................... 22
7.3 User Settings ..................................................... 22
7.4 Environment Variables ......................................... 23
7.5 User Config: Ribbon/Config File ............................... 23
7.6 Workbook Directory Config: Config file ....................... 24
7.7 Workbook Config: xlwings.conf Sheet ......................... 24
7.8 Alternative: Standalone VBA module .......................... 24

8 RunPython .......................................................... 25
8.1 xlwings add-in .................................................... 25
8.2 Call Python with “RunPython” ................................. 25
8.3 Function Arguments and Return Values ....................... 26

9 User Defined Functions (UDFs) .................................. 27
9.1 One-time Excel preparations .................................... 27
9.2 Workbook preparation .......................................... 27
9.3 A simple UDF .................................................... 28
9.4 Array formulas: Get efficient .................................... 28
9.5 Array formulas with NumPy and Pandas ....................... 29
9.6 @xw.arg and @xw.ret decorators ............................... 30
9.7 Dynamic Array Formulas ....................................... 30
9.8 Docstrings ......................................................... 32
9.9 The “caller” argument ......................................... 32
9.10 The “vba” keyword ............................................ 32
9.11 Macros .......................................................... 32
9.12 Call UDFs from VBA ........................................... 33
9.13 Asynchronous UDFs ............................................ 33

10 Matplotlib & Plotly Charts ...................................... 35
10.1 Matplotlib ....................................................... 35
10.2 Plotly static charts ............................................ 38

11 Command Line Client (CLI) ..................................... 41

12 xlwings Reports .................................................. 43
12.1 Quickstart ....................................................... 43
12.2 Frames ........................................................ 44
12.3 Excel Tables .................................................... 45
12.4 Shape Text ...................................................... 47

13 Deployment ........................................................ 49
13.1 Zip files ........................................................ 49
13.2 RunFrozenPython .............................................. 49
Index
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 :)

CHAPTER 1

Video course
Installation

2.1 Prerequisites

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

Here are the last versions of xlwings to support:

- Python 3.5: 0.19.5
- Python 2.7: 0.16.6

2.2 Installation

xlwings comes pre-installed with Anaconda. If you are new to Python or have trouble installing xlwings, Anaconda is highly recommended. Otherwise you can also install it manually with pip:

```
pip install xlwings
```

or conda:

```
conda install xlwings
```

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

```
conda install -c conda-forge xlwings
```
**Note:** When you are on macOS and are installing xlwings with conda (or use the version that comes with Anaconda), you’ll need to run `$ xlwings runpython install` once to enable the RunPython calls from VBA. This is done automatically if you install the addin via `$ xlwings addin install`.

### 2.3 Add-in

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

```
xlwings addin install
```

To call Excel from Python, you don’t need an add-in. Also, you can use a single file VBA module instead of the add-in. For more details, see *Add-in & Settings*.

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

### 2.4 Dependencies

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

The dependencies are automatically installed via conda or pip.

### 2.5 How to activate xlwings PRO

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

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

To activate the license key, run the following command:

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

Make sure to replace `LICENSE_KEY` with your personal key. This will store the license key under your `xlwings.conf` file in your home folder. Alternatively, you can also store the license key under an environment variable with the name `XLWINGS_LICENSE_KEY`. 
xlwings PRO requires additionally the cryptography and Jinja2 packages which come pre-installed with Anaconda. Otherwise, install them via pip or conda.

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

### 2.6 Optional Dependencies

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

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

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

### 2.7 Update

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

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

or:

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

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

```
xlwings addin install
```

### 2.8 Uninstall

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

```
xlwings addin remove
```

Then

```
pip uninstall xlwings
```

or:

```
conda remove xlwings
```

Finally, manually remove the `.xlwings` directory in your home folder if it exists.
This guide assumes you have xlwings already installed. If that’s not the case, head over to Installation.

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

Establish a connection to a workbook:

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

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

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

Instantiate a sheet object:

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

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

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

There are many convenience features available, e.g. Range expanding:
Powerful converters handle most data types of interest, including Numpy arrays and Pandas DataFrames in both directions:

```python
>>> import pandas as pd
>>> df = pd.DataFrame([[1,2], [3,4]], columns=['a', 'b'])
>>> sht.range('A1').value = df
>>> sht.range('A1').options(pd.DataFrame, expand='table').value
   a  b
0  0.0  1.0  2.0
1  3.0  4.0
```

Matplotlib figures can be shown as pictures in Excel:

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

Shortcut for the active sheet: `xw.Range`

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

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

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

### 3.2 2. 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

Per default, RunPython expects hello.py in the same directory as the Excel file but you can change that via config. Refer to the calling Excel book by using xw.Book.caller:

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

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

To make this run, you’ll need to have the xlwings add-in installed. 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.

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

Writing a UDF in Python is as easy as:

```
import xlwings as xw

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

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

```
import xlwings as xw
import pandas as pd

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

Import this function into Excel by clicking the import button of the xlwings add-in: For further details, see User Defined Functions (UDFs).
Connect to a Book

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

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

### 4.1 Python to Excel

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

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

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

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

---

11
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.
CHAPTER

5

Syntax Overview

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

All code samples below depend on the following import:

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

### 5.1 Active Objects

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

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

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

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

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

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

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

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

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

5.3 Range indexing/slicing

Range objects support indexing and slicing, a few examples:

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

5.4 Range Shortcuts

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

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

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

```python
>>> rng = xw.apps[10559].books[0].sheets[0].range('A1')
>>> rng.sheet.book.app
<Excel App 10559>
```
This tutorial gives you a quick introduction to the most common use cases and default behaviour of xlwings when reading and writing values. For an in-depth documentation of how to control the behavior using the options method, have a look at *Converters and Options*.

All code samples below depend on the following import:

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

### 6.1 Single Cells

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

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

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

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

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

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

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

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

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

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

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

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

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

```python
>>> sht = xw.Book().sheets[0]
>>> sht.range('A1').value = [[1, 2], [3, 4]]
>>> rng1 = sht.range('A1').expand('table')  # or just .expand()
>>> rng2 = sht.range('A1').options(expand='table')
>>> rng1.value
[[1.0, 2.0], [3.0, 4.0]]
>>> rng2.value
[[1.0, 2.0], [3.0, 4.0]]
>>> sht.range('A3').value = [5, 6]
>>> rng1.value
[[1.0, 2.0], [3.0, 4.0]]
>>> rng2.value
[[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]]
```

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

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

6.4 NumPy arrays

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

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

6.5 Pandas DataFrames

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

(continues on next page)
6.6 Pandas Series

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

**Note:** You only need to specify the top left cell when writing a list, a NumPy array or a Pandas DataFrame to Excel, e.g.: `sht.range('A1').value = np.eye(10)`
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 from Python via xlwings.

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

**Note:** The add-in is password protected with the password xlwings. For debugging or to add new extensions, you need to unprotect it.

### 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\python.exe" or "/usr/local/bin/python3.9". Note that in the settings, this is stored as Interpreter_Win or Interpreter_Mac, respectively, see below!

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

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

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

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

- **Debug UDFs**: Check this box if you want to run the xlwings COM server manually for debugging, see Debugging.
- **RunPython**: Use UDF Server: Uses the same COM Server for RunPython as for UDFs. This will be faster, as the interpreter doesn’t shut down after each call.

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

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

### 7.3.1 Anaconda/Miniconda

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

### 7.4 Environment Variables

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

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

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

### 7.5 User Config: Ribbon/Config File

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

- **Windows**: `.xlwings\xlwings.conf` in your user folder

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

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

```
"INTERPRETER_WIN", "C:\path\to\python.exe"
"INTERPRETER_MAC", "/path/to/python"
"PYTHONPATH", ""
"CONDA_PATH", ""
"CONDA_ENV", ""
"UDF MODULES", ""
"DEBUG UDFS", ""
"USE UDF SERVER", ""
"SHOW CONSOLE", ""
"ONEDRIVE", ""
```

**Note:** The **ONEDRIVE** 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 vars on

---

**7.4. Environment Variables**
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 has to point to the root folder of your local OneDrive folder.

### 7.6 Workbook Directory Config: Config file

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

### 7.7 Workbook Config: xlwings.conf Sheet

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

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>PYTHONPATH</td>
</tr>
<tr>
<td>3</td>
<td>UDF Modules</td>
</tr>
<tr>
<td>4</td>
<td>Debug UDFs</td>
</tr>
<tr>
<td>5</td>
<td>Log File</td>
</tr>
<tr>
<td>6</td>
<td>Use UDF Server</td>
</tr>
</tbody>
</table>
```

### 7.8 Alternative: Standalone VBA module

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

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

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

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

```bash
$ xlwings quickstart myproject
```

8.2 Call Python with “RunPython”

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

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

This calls the following code in hello.py:

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

def world():
```

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

**Note:** Place `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.
This tutorial gets you quickly started on how to write User Defined Functions.

Note:

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

9.1 One-time Excel preparations

1) Enable Trust access to the VBA project object model under File > Options > Trust Center > Trust Center Settings > Macro Settings

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

9.2 Workbook preparation

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

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

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

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

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

```python
import xlwings as xw

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

- Now click on Import Python UDFs in the xlwings tab to pick up the changes made to myproject.py.
- Enter the formula =double_sum(1, 2) into a cell and you will see the correct result:

![Image of Excel cell with formula result]

- The docstring (in triple-quotes) will be shown as function description in Excel.

Note:

- You only need to re-import your functions if you change the function arguments or the function name.
- Code changes in the actual functions are picked up automatically (i.e. at the next calculation of the formula, e.g. triggered by Ctrl-Alt-F9), but changes in imported modules are not. This is the very behaviour of how Python imports work. If you want to make sure everything is in a fresh state, click Restart UDF Server.
- The @xw.func decorator is only used by xlwings when the function is being imported into Excel. It tells xlwings for which functions it should create a VBA wrapper function, otherwise it has no effect on how the functions behave in Python.

9.4 Array formulas: Get efficient

Calling one big array formula in Excel is much more efficient than calling many single-cell formulas, so it’s generally a good idea to use them, especially if you hit performance problems.
You can pass an Excel Range as a function argument, as opposed to a single cell and it will show up in Python as list of lists.

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

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

To use this formula in Excel,

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

![Excel screenshot showing array formula](image)

### 9.4.1 Number of array dimensions: ndim

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

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

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

### 9.5 Array formulas with NumPy and Pandas

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

To define a formula for matrix multiplication using numpy arrays, you would define the following function:
import xlwings as xw
import numpy as np

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

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

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

import xlwings as xw
import pandas as pd

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

9.6 @xw.arg and @xw.ret decorators

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

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

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

10.1.1 Getting started

The easiest sample boils down to:

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

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

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

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

10.1.2 Full integration with Excel

Calling the above code with `RunPython` and binding it e.g. to a button is straightforward and works cross-platform.
However, on Windows you can make things feel even more integrated by setting up a `UDF` along the following lines:

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

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

### 10.1.3 Properties

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

For example:

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

or:

```python
>>> plot = sht.pictures.add(fig, name='MyPlot', update=True)
>>> plot.height /= 2
>>> plot.width /= 2
```
xlwings - Make Excel Fly!, Release dev

10.1. Matplotlib
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()
```

10.2 Plotly static charts

This feature requires xlwings PRO.

10.2.1 Prerequisites

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

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

For alternative ways of installation, see: https://plotly.com/python/static-image-export/

10.2.2 How to use

It works the same as with Matplotlib, however, rendering a Plotly chart takes slightly longer. Here is a sample:
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()
wbsheets[0].pictures.add(fig, name='IrisScatterPlot', update=True)
xlwings comes with a command line client. On Windows, type the commands into a Command Prompt, on Mac, type them into a Terminal. To get an overview of all commands, simply type `xlwings` and hit Enter:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>addin</strong></td>
<td>Run &quot;xlwings addin install&quot; to install the Excel add-in (will be copied to the XLSTART folder). Instead of &quot;install&quot; you can also use &quot;update&quot;, &quot;remove&quot; or &quot;status&quot;. Note that this command may take a while. Use the &quot;--unprotected&quot; flag to install the add-in without password protection. (New in 0.6.0, the unprotected flag was added in 0.20.4)</td>
</tr>
<tr>
<td><strong>quickstart</strong></td>
<td>Run &quot;xlwings quickstart myproject&quot; to create a folder called &quot;myproject&quot; in the current directory with an Excel file and a Python file, ready to be used. Use the &quot;--standalone&quot; flag to embed all VBA code in the Excel file and make it work without the xlwings add-in. (New in 0.6.0, the unprotected flag was added in 0.20.4)</td>
</tr>
<tr>
<td><strong>runpython</strong></td>
<td>macOS only: run &quot;xlwings runpython install&quot; if you want to enable the RunPython calls without installing the add-in. This will create the following file: ~/Library/Application Scripts/com.microsoft.Excel/xlwings.applescript (new in 0.7.0)</td>
</tr>
<tr>
<td><strong>restapi</strong></td>
<td>Use &quot;xlwings restapi run&quot; to run the xlwings REST API via Flask dev server. Accepts &quot;--host&quot; and &quot;--port&quot; as optional arguments.</td>
</tr>
<tr>
<td><strong>license</strong></td>
<td>xlwings PRO: Use &quot;xlwings license update -k KEY&quot; where &quot;KEY&quot; is your personal (trial) license key. This will update ~/.xlwings/xlwings.conf with the LICENSE_KEY entry. If you have a paid license, you can run &quot;xlwings license deploy&quot; to create a deploy key. This (continues on next page)</td>
</tr>
<tr>
<td>config</td>
<td>Run &quot;xlwings config create&quot; 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 &quot;--force&quot; flag which will overwrite your current configuration. (New in 0.19.5)</td>
</tr>
<tr>
<td>code</td>
<td>Run &quot;xlwings code embed&quot; to embed all Python modules of the current dir in your active Excel file. Use the &quot;--file&quot; flag to only import a single file by providing its path. To run embedded code, you need an xlwings PRO license. (New in 0.20.2)</td>
</tr>
</tbody>
</table>
xlwings Reports

This feature requires xlwings PRO.
See also the Reports API reference.

12.1 Quickstart

xlwings Reports is part of xlwings PRO and a solution for template based Excel and PDF reporting. It allows business users without Python knowledge to create & maintain Excel templates without having to go back to a Python developer for every change: xlwings Reports separates the Python code (that gets and prepares all the data) from the Excel template (that defines which data goes where and how it should be formatted). See also the xlwings Reports homepage.

Start by creating the following Python script my_template.py:

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

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

Then create the following Excel file called my_template.xlsx:

Now run the Python script:

```bash
g: python my_template.py
```

This will copy the template and create the following output by replacing the variables in double curly braces with the value from the Python variable:
Apart from Strings and Pandas DataFrames, you can also use numbers, lists, simple dicts, NumPy arrays, Matplotlib figures and PIL Image objects that have a filename.

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

### 12.2 Frames

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

- Variables do not overwrite existing cell values as they do without Frames.
- Table formatting is applied to all data rows.

To use Frames, insert `<frame>` into **row 1** of your Excel template wherever you want a new dynamic column to start. Row 1 will be removed automatically when creating the report. Frames go from one `<frame>` to the next `<frame>` or the right border of the used range.

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

You can define and format tables by formatting exactly

- one header and
- one data row

as shown in the screenshot:

Running the following code:

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

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

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

create_report('my_template.xlsx',
              'my_report.xlsx',
              **data)

will generate this report:

12.3 Excel Tables

Using Excel tables is the recommended way to format tables as the styling can be applied dynamically across columns and rows. You can also use themes and apply alternating colors to rows/columns. On top of that, they are the easiest way to make the source of a chart dynamic. Go to Insert > Table and make sure that you activate My table has headers before clicking on OK. Add the placeholder as usual on the top-left of your Excel table:

Running the following script:

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

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

(continues on next page)
create_report('template.xlsx', 'output.xlsx', df=df.set_index('col 0'))

Will produce the following report:

Note:

• If you would like to exclude the DataFrame index, make sure to set the index to the first column e.g.: df.set_index('column_name').

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

• For Excel table support, you need at least version 0.21.0 and the index behavior was changed in 0.21.3
12.4 Shape Text

New in version 0.21.4.

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

```python
from xlwings.pro.reports import create_report
create_report('template.xlsx', 'output.xlsx', temperature=12.3)
```

This code turns this template:

![Template Image]

into this report:

![Report Image]
CHAPTER 13

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:

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

## 13.3 Embedded Code

This feature requires xlwings PRO.

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

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

```bash
$ xlwings code embed
```

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

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

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

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

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

### 13.4 One-Click Zero-Config Installer

This feature requires xlwings PRO.

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

13.4.1 End User Instructions

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

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

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

13.4.2 Build the Installer

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

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

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

13.4.3 Configuration

Excel file

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

• Add the standalone xlwings VBA module, e.g. via xlwings quickstart project --standalone
• Make sure that in the VBA editor (Alt-F11) under Tools > References xlwings is unchecked
• Rename the _xlwings.conf sheet into xlwings.conf
• In the `xlwings.conf` sheet, as Interpreter, set the following value: 
  `%LOCALAPPDATA%\project` while replacing `project` with the name of your project

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

**Source code**

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

**Dependencies**

Add your dependencies to `requirements.txt`. For example:

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

**Code signing (optional)**

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

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

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

**Project details**

Update the following under `.github/main.yml`:

```yaml
PROJECT:
APP_PUBLISHER:
```

**Python version**

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

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

### 13.5 Deployment Key

This feature requires `xlwings PRO`.

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

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

To create a deployment key, run the following command:

```
xlwings license deploy
```

Then paste the generated key into the xlwings config as `LICENSE_KEY`. For deployment purposes, usually the best place to do that is on a sheet called `xlwings.conf`, but you can also use an `xlwings.conf` file in either the same folder or in the `.xlwings` folder within the user’s home folder. To use an environment variable, use `XLWINGS_LICENSE_KEY`. See also *User Settings.*
CHAPTER 14

Troubleshooting

14.1 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

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

Solution:

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

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

Note: Don’t use quotes on the xlwings.conf sheet.
The purpose of xlwings PRO is to finance the continued maintenance and enhancement of xlwings. This will allow you to rely on the package without being left with the dreaded “this library currently has no active maintainers” message that happens to too many open-source packages after a couple of years.

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

Note: To get access to the additional functionality of xlwings PRO, you need a license key and at least xlwings v0.19.0. Everything under the xlwings.pro subpackage is distributed under a commercial license.

15.1 PRO Features

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

- Pricing: https://www.xlwings.org/pricing
- Trial license key: https://www.xlwings.org/trial
Introduction with v0.7.0, converters define how Excel ranges and their values are converted both during reading and writing operations. They also provide a consistent experience across xlwings.Range objects and User Defined Functions (UDFs).

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

All code samples below depend on the following import:

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

Syntax:

<table>
<thead>
<tr>
<th>Reading</th>
<th>xw.Range.options(convert=None, **kwargs).value</th>
<th>UDFs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>@arg('x', convert=None, **kwargs)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Writing</th>
<th>xw.Range.options(convert=None, **kwargs).value = myvalue</th>
<th>UDFs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>@ret(convert=None, **kwargs)</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

  ```python
  >>> sht.range('A1').value = 1
  >>> sht.range('A1').value
  1.0
  >>> sht.range('A1').options(numbers=int).value
  1
  ```

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

  Using this on UDFs looks like this:

  ```python
  @xw.func
  @xw.arg('x', numbers=int)
  def myfunction(x):
  ```

(continues on next page)
# all numbers in x arrive as int
return x

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

• dates

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

– Range:

```python
>>> import datetime as dt
>>> sht.range('A1').options(dates=dt.date).value
```

– UDFs: @xw.arg('x', dates=dt.date)

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

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

• empty

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

– Range: >>> sht.range('A1').options(empty='NA').value

– UDFs: @xw.arg('x', empty='NA')

• transpose

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

– Range: sht.range('A1').options(transpose=True).value = [1, 2, 3]

– UDFs:

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

• expand

This works the same as the Range properties table, vertical and horizontal but is only evaluated when getting the values of a Range:
>>> import xlwings as xw
>>> sht = xw.Book().sheets[0]
>>> sht.range('A1').value = [[1,2], [3,4]]
>>> rng1 = sht.range('A1').expand()
>>> rng2 = sht.range('A1').options(expand='table')
>>> rng1.value
[[1.0, 2.0], [3.0, 4.0]]
>>> rng2.value
[[1.0, 2.0], [3.0, 4.0]]
>>> sht.range('A3').value = [5, 6]
>>> rng1.value
[[1.0, 2.0], [3.0, 4.0]]
>>> rng2.value
[[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]]

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

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:

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

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

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:**

```python
>>> sht = xw.Book().sheets[0]
>>> s = sht.range('A1').options(pd.Series, expand='table').value
>>> s
date        2001-01-01   1
             2001-01-02   2
```
2001-01-03  3
2001-01-04  4
2001-01-05  5
2001-01-06  6
Name: series name, dtype: float64

```python
>>> sht.range('D1', header=False).value = s
```

### 16.2.4 Pandas DataFrame converter

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

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

**index:** `int` or `Boolean`

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

**header:** `int` or `Boolean`

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

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

**Example:**

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

# Writing back using the defaults:
>>> sht.range('A1').value = df
```
# Writing back and changing some of the options, e.g. getting rid of the index:

```python
>>> sht.range('B7').options(index=False).value = df
```

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

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

## 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:

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

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

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

Here are the steps to implement your own converter:

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

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

```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
sht = xw.Book().sheets[0]
df = pd.DataFrame([[1.,10.],[2.,np.nan],[3., 30.]]

• Default converter for DataFrames:
  # Write
  sht.range('A1').value = df

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

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

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

16.3. Custom Converter
• Register an alias (optional):

```python
DataFrameDropna.register('df_dropna')

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

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

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

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

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

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

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

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

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

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

The `Converter` class provides basic scaffolding to make the task of writing a new Converter easier. If you need more control you can subclass `Accessor` directly, but this part requires more work and is currently undocumented.
Since xlwings runs in every Python environment, you can use your preferred way of debugging.

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

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

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

![Error Message]

**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].range('A1').value = 1

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

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

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

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.
17.2. UDF debug server
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.
19.1 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
import queue
import xlwings as xw

num_threads = 4

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

q = queue.Queue()

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

    # THIS DOESN'T WORK - passing xlwings objects to threads will fail!
    rng = xw.Book('Book1.xlsx').sheets[0].range(cell)
    q.put(rng)

q.join()

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

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

num_threads = 4

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

q = Queue()

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

    q.put(cell)

q.join()

19.2 Multiprocessing

Note: Multiprocessing is only supported on Windows!

The same rules apply to multiprocessing as for threading, here’s a working example:
from multiprocessing import Pool
import xlwings as xw

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

if __name__ == '__main__':
    with Pool(4) as p:
If you’re missing a feature in xlwings, do the following:

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

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

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

This works accordingly for the other objects like `sheet.range('A1').api` etc.

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

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

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

# Mac
sheet.range('A1').api.wrap_text.set(True)
```
xlwings can also be used to call Python functions from VBA within Office apps other than Excel (like Outlook, Access etc.).

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

### 21.1 How To

1. As usual, write your Python function and import it into Excel (see *User Defined Functions (UDFs)*).
2. Press Alt-F11 to get into the VBA editor, then right-click on the xlwings_udfs VBA module and select Export File.... Save the xlwings_udfs.bas file somewhere.
3. Switch into the other Office app, e.g. Microsoft Access and click again Alt-F11 to get into the VBA editor. Right-click on the VBA Project and Import File..., then select the file that you exported in the previous step. Once imported, replace the app name in the first line to the one that you are using, i.e. Microsoft Access or Microsoft Outlook etc. so that the first line then reads: 

   #Const App = "Microsoft Access"

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

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

### 21.2 Config

The other Office apps will use the same global config file as you are editing via the Excel ribbon add-in. When it makes sense, you’ll be able to use the directory config file (e.g., you can put it next to your Access or Word file) or you can hardcode the path to the config file in the VBA standalone module, e.g., in the function `GetDirectoryConfigFilePath` (e.g., suggested when using Outlook that doesn’t really have the same concept of files like the other Office apps). NOTE: For Office apps without file concept, you need to make sure that the `PYTHONPATH` points to the directory with the Python source file. For details on the different config options, see `Config`. 
While xlwings is a pure Python package, there are cross-language packages that allow for a relatively straightforward use from/with other languages. This means, however, that you’ll always need to have Python with xlwings installed in addition to R or Julia. We recommend the Anaconda distribution, see also Installation.

22.1 R

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

Setup:

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

22.1.1 Simple functions with R

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

Python wrapper code:

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

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

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

### 22.1.2 Array functions with R

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

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

Python wrapper code:

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

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

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

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

### 22.2 Julia

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

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

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

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

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

```
julia> xw.Range("A1")[:value]
"Hello World"
```
23.1 Top-level functions

xlwings.view(obj, sheet=None)

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

Parameters

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

Examples

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

New in version 0.7.1.
23.2 Object model

23.2.1 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>])

    active
    Returns the active app.
    New in version 0.9.0.

    add()

    count
    Returns the number of apps.
    New in version 0.9.0.

    keys()
    Provides the PIDs of the Excel instances that act as keys in the Apps collection.
    New in version 0.13.0.

23.2.2 App

class xlwings.App(visibility=None, spec=None, add_book=True, impl=None)
    An app corresponds to an Excel instance. New Excel instances can be fired up like so:
    
    >>> import xlwings as xw
    >>> app1 = xw.App()
    >>> app2 = xw.App()

    An app object is a member of the apps collection:
    
    >>> xw.apps
    Apps([<Excel App 1668>, <Excel App 1644>])
    >>> xw.apps[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.

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

New in version 0.9.0.

**books**
A collection of all Book objects that are currently open.

New in version 0.9.0.

**calculate**()
Calculates all open books.

New in version 0.3.6.

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

**Examples**

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

Changed in version 0.9.0.

**display_alerts**
The default value is True. Set this property to False to suppress prompts and alert messages while code is running; when a message requires a response, Excel chooses the default response.
New in version 0.9.0.

**hwnd**

Returns the Window handle (Windows-only).

New in version 0.9.0.

**kill()**

Forces the Excel app to quit by killing its process.

New in version 0.9.0.

**macro**(name)

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

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

**Examples**

This VBA function:

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

can be accessed like this:

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

See also: `Book.macro()`

New in version 0.9.0.

**pid**

Returns the PID of the app.

New in version 0.9.0.

**quit()**

Quits the application without saving any workbooks.

New in version 0.3.3.

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

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

New in version 0.9.0.
**screen_updating**
Turn screen updating off to speed up your script. You won’t be able to see what the script is doing, but it will run faster. Remember to set the screen_updating property back to True when your script ends.

New in version 0.3.3.

**selection**
Returns the selected cells as Range.

New in version 0.9.0.

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

New in version 0.19.4.

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

New in version 0.20.0.

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

**Examples**

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

Changed in version 0.9.0.

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

New in version 0.3.3.

### 23.2.3 Books

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

```python
>>> import xlwings as xw
>>> xw.books  # active app
>>> xw.apps[10559].books  # specific app, get the PIDs via xw.apps.keys()
```
New in version 0.9.0.

**active**

Returns the active Book.

**add()**


**open**

```python
(open(fullname, update_links=None, read_only=None, format=None, password=None, write_res_password=None, ignore_read_only_recommended=None, origin=None, delimiter=None, editable=None, notify=None, converter=None, add_to_mru=None, local=None, corrupt_load=None))
```

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

**Parameters**

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

**Returns** Book

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

### 23.2.4 Book

**class xlwings.Book**

```python
class xlwings.Book(fullname=None, update_links=None, read_only=None, format=None, password=None, write_res_password=None, ignore_read_only_recommended=None, origin=None, delimiter=None, editable=None, notify=None, converter=None, add_to_mru=None, local=None, corrupt_load=None, impl=None)
```

A book object is a member of the books collection:

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

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

```python
>>> app = xw.App()  # or something like xw.apps[10559] for existing apps, get the PIDs via xw.apps.keys()
>>> app.books['Book1']
```
<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fullname</strong> <em>(str or path-like object, default None)</em></td>
<td>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.</td>
</tr>
<tr>
<td><strong>update_links</strong> <em>(bool, default None)</em></td>
<td>If this argument is omitted, the user is prompted to specify how links will be updated.</td>
</tr>
<tr>
<td><strong>read_only</strong> <em>(bool, default False)</em></td>
<td>True to open workbook in read-only mode.</td>
</tr>
<tr>
<td><strong>format</strong> <em>(str)</em></td>
<td>If opening a text file, this specifies the delimiter character.</td>
</tr>
<tr>
<td><strong>password</strong> <em>(str)</em></td>
<td>Password to open a protected workbook.</td>
</tr>
<tr>
<td><strong>write_res_password</strong> <em>(str)</em></td>
<td>Password to write to a write-reserved workbook.</td>
</tr>
<tr>
<td><strong>ignore_read_only_recommended</strong> <em>(bool, default False)</em></td>
<td>Set to True to mute the read-only recommended message.</td>
</tr>
<tr>
<td><strong>origin</strong> <em>(int)</em></td>
<td>For text files only. Specifies where it originated. Use XIPlat-form constants.</td>
</tr>
<tr>
<td><strong>delimiter</strong> <em>(str)</em></td>
<td>If format argument is 6, this specifies the delimiter.</td>
</tr>
<tr>
<td><strong>editable</strong> <em>(bool, default False)</em></td>
<td>This option is only for legacy Microsoft Excel 4.0 addins.</td>
</tr>
<tr>
<td><strong>notify</strong> <em>(bool, default False)</em></td>
<td>Notify the user when a file becomes available If the file cannot be opened in read/write mode.</td>
</tr>
<tr>
<td><strong>converter</strong> <em>(int)</em></td>
<td>The index of the first file converter to try when opening the file.</td>
</tr>
<tr>
<td><strong>add_to_mru</strong> <em>(bool, default False)</em></td>
<td>Add this workbook to the list of recently added workbooks.</td>
</tr>
<tr>
<td><strong>local</strong> <em>(bool, default False)</em></td>
<td>If True, saves files against the language of Excel, otherwise against the language of VBA. Not supported on macOS.</td>
</tr>
<tr>
<td><strong>corrupt_load</strong> <em>(int, default xlNormalLoad)</em></td>
<td>Can be one of xlNormalLoad, xlRepairFile or xlExtractData. Not supported on macOS.</td>
</tr>
</tbody>
</table>

**activate** *(steal_focus=False)*

Activates the book.
Parameters **steal_focus** *(bool, default False)* – If True, make front-most window and hand over focus from Python to Excel.

**api**

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

New in version 0.9.0.

**app**

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

New in version 0.9.0.

**classmethod caller()**

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

```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.

**fullname**

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

**macro**(name)

Runs a Sub or Function in Excel VBA.

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

**Examples**

This VBA function:

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

can be accessed like this:
See also: `App.macro()`

New in version 0.7.1.

**name**

Returns the name of the book as str.

**names**

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

Changed in version 0.9.0.

**save** *(path=None)*

Saves the Workbook. If a path is being provided, this works like SaveAs() in Excel. If no path is specified and if the file hasn’t been saved previously, it’s being saved in the current working directory with the current filename. Existing files are overwritten without prompting.

**Parameters**

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

**Example**

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

New in version 0.3.1.

**selection**

Returns the selected cells as Range.

New in version 0.9.0.

**set_mock_caller()**

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

**Examples**

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

(continues on next page)
import xlwings as xw

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

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

New in version 0.3.1.

sheets

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

New in version 0.9.0.

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

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

Parameters

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

Examples

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

New in version 0.21.1.

### 23.2.5 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.

**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.

### 23.2.6 Sheet

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

A sheet object is a member of the sheets collection:

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

Changed in version 0.9.0.

**activate()**

Activates the Sheet and returns it.

**api**

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

New in version 0.9.0.

**autofit**(axis=None)

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

**Parameters**

- **axis**(string, default None) –
• To autofit rows, use one of the following: rows or r
• To autofit columns, use one of the following: columns or c
• To autofit rows and columns, provide no arguments

Examples

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

New in version 0.2.3.

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

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

New in version 0.9.0.

**charts**
See Charts

New in version 0.9.0.

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

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

**delete()**
Deletes the Sheet.

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

**name**
Gets or sets the name of the Sheet.

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

New in version 0.9.0.

**pictures**
See Pictures

New in version 0.9.0.
range \((cell1, cell2=\text{None})\)

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

New in version 0.9.0.

select ()

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

New in version 0.9.0.

shapes

See `Shapes`

New in version 0.9.0.

tables

See `Tables`

New in version 0.21.0.

used_range

Used Range of Sheet.

Returns

Return type `xw.Range`

New in version 0.13.0.

visible

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

New in version 0.21.1.

### 23.2.7 Range

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

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

Parameters

- \textbf{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’

- \textbf{cell2} (`str or tuple or Range, default None`) – Name of the range in the lower-right corner in A1 notation or as index-tuple or as name or as `xw.Range` object.

**Examples**

Active Sheet:
import xlwings as xw

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

Specific Sheet:

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

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

Adds a hyperlink to the specified Range (single Cell)

Parameters

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

New in version 0.3.0.

address

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

New in version 0.9.0.

api

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

New in version 0.9.0.

autofit()

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

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

Changed in version 0.9.0.

clear()

Clears the content and the formatting of a Range.

clear_contents()

Clears the content of a Range but leaves the formatting.
color

Gets and sets the background color of the specified Range.

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

**Returns** RGB

**Return type** tuple

**Examples**

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

New in version 0.3.0.

column

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

**Returns**

**Return type** Integer

New in version 0.3.5.

column_width

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

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

column_width must be in the range: \(0 \leq \text{column_width} \leq 255\)

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

**Returns**

**Return type** float

New in version 0.4.0.

columns

Returns a \texttt{RangeColumns} object that represents the columns in the specified range.

New in version 0.9.0.
copy (destination=None)
Copy a range to a destination range or clipboard.

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

Returns
Return type None

count
Returns the number of cells.

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

Returns
Return type Range object

delete (shift=None)
Deletes a cell or range of cells.

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

Returns
Return type None

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

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

Examples

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

New in version 0.9.0.

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

102 Chapter 23. Python API
Parameters **mode** (*str*, *default* 'table') – One of 'table' (=down and right), 'down', 'right'.

Returns

Return type  *Range*

Examples

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

New in version 0.9.0.

**formula**

Gets or sets the formula for the given Range.

**formula2**

Gets or sets the formula2 for the given Range.

**formula_array**

Gets or sets an array formula for the given Range.

New in version 0.7.1.

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

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

Parameters

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

Returns

Return type  *str*
Examples

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

New in version 0.2.3.

**has_array**

Are we part of an Array formula?

**height**

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

Returns

**Return type**  float

New in version 0.4.0.

**hyperlink**

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

Examples

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

New in version 0.3.0.

**insert** *(shift=None, copy_origin='format_from_left_or_above')*

Insert a cell or range of cells into the sheet.

**Parameters**

- **shift** *(str, default None)* – Use right or down. If omitted, Excel decides based on the shape of the range.

- **copy_origin** *(str, default format_from_left_or_above)* – Use *format_from_left_or_above* or *format_from_right_or_below*. Note that this is not supported on macOS.
Returns
Return type  None

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

Returns
Return type  Range

Example

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

New in version 0.3.5.

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

Returns
Return type  float

New in version 0.6.0.

merge (across=False)
Creates a merged cell from the specified Range object.

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

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

merge_cells
Returns True if the Range contains merged cells, otherwise False

name
Sets or gets the name of a Range.

New in version 0.4.0.

number_format
Gets and sets the number_format of a Range.
Examples

```python
>>> import xlwings as xw
>>> wb = xw.Book()
>>> xw.Range('A1').number_format
'General'
>>> xw.Range('A1:C3').number_format = '0.00%'
>>> xw.Range('A1:C3').number_format
'0.00%'
```

New in version 0.2.3.

**offset** *(row_offset=0, column_offset=0)*

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

**Returns** Range object

**Return type** Range

New in version 0.3.0.

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

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

**Parameters**

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

**Keyword Arguments**

- **ndim** *(int, default None)* — number of dimensions
- **numbers** *(type, default None)* — type of numbers, e.g. int
- **dates** *(type, default None)* — e.g. datetime.date defaults to datetime.datetime
- **empty** *(object, default None)* — transformation of empty cells
- **transpose** *(Boolean, default False)* — transpose values
- **expand** *(str, default None)* — One of 'table', 'down', 'right'

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

**Returns**

**Return type** Range object

New in version 0.7.0.

**paste** *(paste=None, operation=None, skip_blanks=False, transpose=False)*

Pastes a range from the clipboard into the specified range.

**Parameters**
paste (str, default None) -- One of
all_merging_conditional_formats, all,
all_exceptBorders, all_using_source_theme,
column_widths, comments, formats, formulas,
formulas_and_number_formats, validation, values,
values_and_number_formats.

operation (str, default None) -- One of “add”, “divide”, “multiply”,
“subtract”.

skip_blanks (bool, default False) -- Set to True to skip over blank
cells

transpose (bool, default False) -- Set to True to transpose rows
and columns.

Returns
Return type None

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

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

Parameters

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

Returns Range object
Return type Range

New in version 0.3.0.

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

Returns
Return type Integer

New in version 0.3.5.

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

row_height must be in the range: 0 <= row_height <= 409.5
Note: If the Range is outside the used range of the Worksheet, and rows in the Range have different heights, returns the height of the first row.

**Returns**

**Return type** float

New in version 0.4.0.

**rows**

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

New in version 0.9.0.

**select()**

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

New in version 0.9.0.

**shape**

Tuple of Range dimensions.

New in version 0.3.0.

**sheet**

Returns the Sheet object to which the Range belongs.

New in version 0.9.0.

**size**

Number of elements in the Range.

New in version 0.3.0.

**table**

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

New in version 0.21.0.

**top**

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

**Returns**

**Return type** float

New in version 0.6.0.

**unmerge()**

Separates a merged area into individual cells.

**value**

Gets and sets the values for the given Range.

**Returns**

**Return type** object

returned object depends on the converter being used, see `xlwings`.  
*Range.options()*
width
Returns the width, in points, of a Range. Read-only.

Returns
Return type  float
New in version 0.4.0.

23.2.8 RangeRows

class xlwings.RangeRows(rng)
Represents the rows of a range. Do not construct this class directly, use Range.rows instead.

Example

```python
import xlwings as xw
rng = xw.Range('A1:C4')
assert len(rng.rows) == 4  # or rng.rows.count
rng.rows[0].value = 'a'
assert rng.rows(2) == xw.Range('A2:C2')
for r in rng.rows:
    print(r.address)
```

autofit()
Autofits the height of the rows.

count
Returns the number of rows.
New in version 0.9.0.

23.2.9 RangeColumns

class xlwings.RangeColumns(rng)
Represents the columns of a range. Do not construct this class directly, use Range.columns instead.

Example
import xlwings as xw
rng = xw.Range('A1:C4')
assert len(rng.columns) == 3  # or rng.columns.count
rng.columns[0].value = 'a'
assert rng.columns(2) == xw.Range('B1:B4')
for c in rng.columns:
    print(c.address)

autofit()
    Autofits the width of the columns.
count
    Returns the number of columns.
    New in version 0.9.0.

23.2.10 Shapes
class xlwings.main.Shapes(impl)
    A collection of all shape objects on the specified sheet:

>>> import xlwings as xw
>>> xw.books['Book1'].sheets[0].shapes
Shapes([<Shape 'Oval 1' in <Sheet [Book1]Sheet1>>, <Shape 'Rectangle 1' in <Sheet [Book1]Sheet1>>])
    New in version 0.9.0.

api
    Returns the native object (pywin32 or appscript obj) of the engine being used.
count
    Returns the number of objects in the collection.

23.2.11 Shape
class xlwings.Shape(*args, **options)
    The shape object is a member of the shapes collection:

>>> import xlwings as xw
>>> sht = xw.books['Book1'].sheets[0]
>>> sht.shapes[0]  # or sht.shapes['ShapeName']
<Shape 'Rectangle 1' in <Sheet [Book1]Sheet1>>
    Changed in version 0.9.0.
activate()  
Activates the shape.  
New in version 0.5.0.

api  
Returns the native object (pywin32 or appscript obj) of the engine being used.  
New in version 0.19.2.

delete()  
Deletes the shape.  
New in version 0.5.0.

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

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

name  
Returns or sets the name of the shape.  
New in version 0.5.0.

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

scale_height (factor, relative_to_original_size=False, scale='scale_from_top_left')  
factor [float] For example 1.5 to scale it up to 150%  
relative_to_original_size [bool, optional] If False, it scales relative to current height (default). For True must be a picture or OLE object.  
scale [str, optional] One of scale_from_top_left (default), scale_from_bottom_right, scale_from_middle  
New in version 0.19.2.

scale_width (factor, relative_to_original_size=False, scale='scale_from_top_left')  
factor [float] For example 1.5 to scale it up to 150%  
relative_to_original_size [bool, optional] If False, it scales relative to current width (default). For True must be a picture or OLE object.  
scale [str, optional] One of scale_from_top_left (default), scale_from_bottom_right, scale_from_middle  
New in version 0.19.2.
Returns or sets the text of a shape.
New in version 0.21.4.

**top**
Returns or sets the number of points that represent the vertical position of the shape.
New in version 0.5.0.

**type**
Returns the type of the shape.
New in version 0.9.0.

**width**
Returns or sets the number of points that represent the width of the shape.
New in version 0.5.0.

### 23.2.12 Charts

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

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

New in version 0.9.0.

**add** *(left=0, top=0, width=355, height=211)*
Creates a new chart on the specified sheet.

**Parameters**

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

**Returns**

**Return type** *Chart*

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

**api**

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

**count**

Returns the number of objects in the collection.

### 23.2.13 Chart

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

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

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

**api**

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

New in version 0.9.0.

**chart_type**

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

```text
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.

def delete()
  Deletes the chart.

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

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

def name
  Returns or sets the name of the chart.

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

def 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')

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

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

### 23.2.14 Pictures

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

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

New in version 0.9.0.

```python
add(image, link_to_file=False, save_with_document=True, left=0, top=0, width=None, height=None, name=None, update=False, scale=1)
  Adds a picture to the specified sheet.
```

Parameters


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

- **left** *(float, default 0)* – Left position in points.

- **top** *(float, default 0)* – Top position in points.

- **width** *(float, default None)* – Width in points. If PIL/Pillow is installed, it defaults to the width of the picture. Otherwise it defaults to 100 points.

- **height** *(float, default None)* – Height in points. If PIL/Pillow is installed, it defaults to the height of the picture. Otherwise it defaults to 100 points.

- **name** *(str, default None)* – Excel picture name. Defaults to Excel standard name if not provided, e.g. ‘Picture 1’.

- **update** *(bool, default False)* – Replace an existing picture with the same name. Requires name to be set.

**Returns**

**Return type** *Picture*

**Examples**

1. Picture

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

2. Matplotlib

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

**api**

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

**count**

Returns the number of objects in the collection.

### 23.2.15 Picture

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

The picture object is a member of the *pictures* collection:
```python
>>> import xlwings as xw
>>> sht = xw.books['Book1'].sheets[0]
>>> sht.pictures[0]  # or sht.charts['PictureName']
<Picture 'Picture 1' in <Sheet [Book1]Sheet1>>
```

Changed in version 0.9.0.

**api**

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

New in version 0.9.0.

**delete()**

Deletes the picture.

New in version 0.5.0.

**height**

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

New in version 0.5.0.

**left**

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

New in version 0.5.0.

**name**

Returns or sets the name of the picture.

New in version 0.5.0.

**parent**

Returns the parent of the picture.

New in version 0.9.0.

**top**

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

New in version 0.5.0.

**update(image)**

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

Parameters:

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

New in version 0.5.0.

**width**

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

New in version 0.5.0.
23.2.16 Names

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

New in version 0.9.0.

```python
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.

```python
api
```

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

New in version 0.9.0.

```python
count
```

Returns the number of objects in the collection.

23.2.17 Name

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

New in version 0.9.0.

```python
api
```

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

New in version 0.9.0.

23.2. Object model
delete()
    Deletes the name.
    New in version 0.9.0.

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

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

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

23.2.18 Tables

class xlwings.main.Tables(impl)

A collection of all table objects on the specified sheet:

```python
>>> import xlwings as xw
>>> xw.books['Book1'].sheets[0].tables
Tables([<Table 'Table1' in <Sheet [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**

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

### 23.2.19 Table

```python
class xlwings.main.Table(*args, **options)
```

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

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

New in version 0.21.0.

**api**

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

**data_body_range**

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

**display_name**

Returns or sets the display name for the specified Table object

**header_row_range**

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

**insert_row_range**

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

**name**

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

Returns the parent of the table.

**range**

Returns an xlwings range object of the table.

**show_autofilter**

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

**show_headers**

Show or hide the header (read/write)

**show_table_style_column_stripes**

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

**show_table_style_first_column**

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

**show_table_style_last_column**

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

**show_table_style_row_stripes**

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

**show_totals**

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

**table_style**

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

**totals_row_range**

Returns an xlwings range object representing the Total row

**update**(data)

This method requires xlwings PRO

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

Changed in version 0.21.3.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>(pandas DataFrame) – Currently restricted to pandas DataFrames. If you want to hide the index, set the first column as the index, e.g. df.set_index('column_name').</td>
</tr>
</tbody>
</table>

**Returns**

**Return type** 

*Table*

**Examples**

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

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

(continues on next page)
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
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)

## 23.3 UDF decorators

**xlwings.\textbf{func}** (category="xlwings", volatile=False, call_in_wizard=True)

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

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

New in version 0.10.3.

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

New in version 0.10.3.

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

New in version 0.10.3.

**xlwings.\textbf{sub}()**

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

**xlwings.\textbf{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:

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

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.
### 23.4 Reports

The `create_report` function can be used as follows:

```python
xlwings.pro.reports.create_report(template, output, book_settings=None, app=None, **data)
```

This feature requires xlwings PRO.

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

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

**Parameters**

- `template (str)` – Path to your Excel template, e.g. `r'C:\Path\to\my_template.xlsx'`
- `output (str)` – Path to your Report, e.g. `r'C:\Path\to\my_report.xlsx'`
- `app (xlwings App, default None)` – By passing in an `xlwings App` instance, you can control where your report runs and configure things like `visible=False`. For details see `xlwings.App`. By default, it creates the report in the currently active instance of Excel.
- `data (kwargs)` – All key/value pairs that are used in the template.

**Returns**

`wb`

**Return type**

`xlwings Book`

**Examples**

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

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

With many template variables it may be useful to collect the data first:

```python
>>> data = dict(title='MyTitle', df=df)
>>> wb = create_report('my_template.xlsx', 'my_report.xlsx', **data)
```

If you need to handle external links or a password, use it like so:
You can control the Excel instance by passing in an xlwings App instance. For example, to run the report in a separate and hidden instance of Excel, do the following:

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

xlwings offers an easy way to expose an Excel workbook via REST API both on Windows and macOS. This can be useful when you have a workbook running on a single computer and want to access it from another computer. Or you can build a Linux based web app that can interact with a legacy Excel application while you are in the progress of migrating the Excel functionality into your web app (if you need help with that, give us a shout).

You can run the REST API server from a command prompt or terminal as follows (this requires Flask>=1.0, so make sure to pip install Flask):

```
xlwings restapi run
```

Then perform a GET request e.g. via PowerShell on Windows or Terminal on Mac (while having an unsaved “Book1” open). Note that you need to run the server and the GET request from two separate terminals (or you can use something more convenient like Postman or Insomnia for testing the API):

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

(continues on next page)
In the command prompt where your server is running, press Ctrl-C to shut it down again.

The xlwings REST API is a thin wrapper around the Python API which makes it very easy if you have worked previously with xlwings. It also means that the REST API does require the Excel application to be up and running which makes it a great choice if the data in your Excel workbook is constantly changing as the REST API will always deliver the current state of the workbook without the need of saving it first.

**Note:** Currently, we only provide the GET methods to read the workbook. If you are also interested in the POST methods to edit the workbook, let us know via GitHub issues. Some other things will also need improvement, most notably exception handling.
24.2 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:

```python
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)
```

24.3 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`.

24.4 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.

24.5 Endpoint overview

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

24.6.1 /book

GET /book/<fullname_or_name>

Example response:

```json
{
    "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:

```json
{
    "names": [
        {
            "name": "Sheet1!mynamel",
            "refers_to": "=Sheet1!$B$2:$C$3"
        },
        {
            "name": "myname2",
            "refers_to": "=Sheet1!$A$1"
        }
    ]
}
```

GET /book/<fullname_or_name>/names/<name>

Example response:

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

GET /book/<fullname_or_name>/names/<name>/range

Example response:
GET /book/<fullname_or_name>/sheets

Example response:

```json
{
    "sheets": [
        {
            "charts": ["Chart 1"],
            "name": "Sheet1",
            "names": ["Sheet1!myname1"],
            "pictures": ["Picture 3"],
            "shapes": ["Chart 1", "Picture 3"],
            "used_range": "$A$1:$B$2"
        },
        {
            "charts": [],
            "name": "Sheet2",
            "names": []
        }
    ]
}
```

(continues on next page)
GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>

Example response:

```json
{
  "charts": [
    "Chart 1"
  ],
  "name": "Sheet1",
  "names": [
    "Sheet1!mynamel"
  ],
  "pictures": [
    "Picture 3"
  ],
  "shapes": [
    "Chart 1",
    "Picture 3"
  ],
  "used_range": "$A$1:$B$2"
}
```

GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>/charts

Example response:

```json
{
  "charts": [
    {
      "chart_type": "line",
      "height": 211.0,
      "left": 0.0,
      "name": "Chart 1",
      "top": 0.0,
      "width": 355.0
    }
  ]
}
```

GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>/charts/<chart_name_or_ix>

Example response:

```json
{
  "chart_type": "line",
  "height": 211.0,
  "left": 0.0,
  "name": "Chart 1",
  "top": 0.0,
  "width": 355.0
}
```
GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>/names

Example response:

```json
{
  "names": [
    {
      "name": "Sheet1!mynamel",
      "refers_to": "=Sheet1!$B$2:$C$3"
    }
  ]
}
```

GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>/names/<sheet_scope_name>

Example response:

```json
{
  "name": "Sheet1!mynamel",
  "refers_to": "=Sheet1!$B$2:$C$3"
}
```

GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>/names/<sheet_scope_name>/range

Example response:

```json
{
  "address": "$B$2:$C$3",
  "color": null,
  "column": 2,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    ["
      ",
    ],
    ["
      ",
    ],
    ["
      ",
    ],
    ["
      ",
    ]
  ]
}
```

(continues on next page)
GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>/pictures

Example response:

```json
{
  "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:

```json
{
  "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",
    (continues on next page)
```

24.6. Endpoint details

133
GET /book/<fullname_or_name>/sheets/<sheet_name_or_ix>/shapes

Example response:

```json
{
  "shapes": [
    {
      "height": 211.0,
      "left": 0.0,
      "name": "Chart 1",
      "shape": 2
    },
    {
      "top": 0.0,
      "value": [
        [2.1,
          "a string"
        ],
        [
          "Mon, 22 Oct 2018 00:09:18 GMT",
          null
        ]
      ]
    }
  ]
}
```
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",
```
GET /books/<book_name_or_ix>

Example response:

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

GET /books/<book_name_or_ix>/names

Example response:

```json
{
  "names": [
    {
      "name": "Sheet1!mynamel",
      "refers_to": "=Sheet1!$B$2:$C$3"
    }
  ]
}
```
```json
{
    "name": "mynname2",
    "refers_to": "=Sheet1!$A$1"
}
}

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

Example response:
```json
{
    "name": "mynname2",
    "refers_to": "=Sheet1!$A$1"
}
```

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

Example response:
```json
{
    "address": "$A$1",
    "color": null,
    "column": 1,
    "column_width": 8.47,
    "count": 1,
    "current_region": "$A$1:$B$2",
    "formula": "=1+1.1",
    "formula_array": "=1+1,1",
    "height": 14.25,
    "last_cell": "$A$1",
    "left": 0.0,
    "name": "mynname2",
    "number_format": "General",
    "row": 1,
    "row_height": 14.3,
    "shape": [1, 1],
    "size": 1,
    "top": 0.0,
    "value": 2.1,
    "width": 51.0
}
```

GET /books/<book_name_or_ix>/sheets

Example response:
```json
{
    "sheets": [
```
GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>

Example response:

```json
{
  "charts": [
    "Chart 1"
  ],
  "name": "Sheet1",
  "names": [
    "Sheet1!myname1"
  ],
  "pictures": [
    "Picture 3"
  ],
  "shapes": [
    "Chart 1",
    "Picture 3"
  ],
  "used_range": "$A$1:$B$2"
}
```

GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/charts

Example response:

```json
{
  "charts": [
    "Chart 1"
  ],
  "name": "Sheet1",
  "names": [
    "Sheet1!myname1"
  ],
  "pictures": [
    "Picture 3"
  ],
  "shapes": [
    "Chart 1",
    "Picture 3"
  ],
  "used_range": "$A$1:$B$2"
}
```
GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/charts/<chart_name_or_ix>

Example response:

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

GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/names

Example response:

```json
{
    "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:

```json
{
    "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:

```json
24.6. Endpoint details 139
```
GET /books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/pictures

Example response:

```json
{
  "pictures": [
    {
      "height": 100.0,
      "left": 0.0,
    }
  ]
}
(continues on next page)```
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$2",
  "column": 1,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "formula": [
    ["=1+1.1",
     "a string"
    ],
    ["43395.0064583333",
     ""
    ]
  ],
  "formula_array": null,
  "height": 28.5,
  "last_cell": "$B$2",
  "left": 0.0,
  "name": null,
  "number_format": null,
  "row": 1,
  "row_height": 14.3,
  "shape": [2, 2]
}
```

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

Example response:

```json
{
  "address": "$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
}
24.6.3 /apps

GET /apps

Example response:

```json
{
   "apps": [ {
      "books": [ 
         "Book1",
         "C:\Users\felix\DEV\xlwings\scripts\Book1.xlsx",
         "Book4"
      ],
      "calculation": "automatic",
      "display_alerts": true,
      "pid": 1104,
      "screen_updating": true,
      "selection": "[Book1.xlsx]Sheet2!$A$1",
      "version": "16.0",
      "visible": true
   },
   {
      "books": [ 
         "Book2",
         "Book5"
      ],
      "calculation": "automatic",
      "display_alerts": true,
      "pid": 7920,
      "screen_updating": true,
      "selection": "[Book5]Sheet2!$A$1",
      "version": "16.0",
      "visible": true
   }
   
}
```

GET /apps/<pid>

Example response:

```json
{
   "books": [ 
      "Book1",
      "C:\Users\felix\DEV\xlwings\scripts\Book1.xlsx",
      "Book4"
   ],
   "calculation": "automatic",
   "display_alerts": true,
   "pid": 1104,
   "screen_updating": true,
   "selection": "[Book1.xlsx]Sheet2!$A$1",
   "version": "16.0",
   "visible": true
}
```

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

Example response:

```json
{
  "books": [
    {
      "app": 1104,
      "fullname": "Book1",
      "name": "Book1",
      "names": [],
      "selection": "Sheet2!$A$1",
      "sheets": [
        "Sheet1"
      ]
    },
    {
      "app": 1104,
      "fullname": "C:\Users\felix\DEV\xlwings\scripts\Book1.xlsx",
      "name": "Book1.xlsx",
      "names": [
        "Sheet1!mynamel",
        "myname2"
      ],
      "selection": "Sheet2!$A$1",
      "sheets": [
        "Sheet1",
        "Sheet2"
      ]
    },
    {
      "app": 1104,
      "fullname": "Book4",
      "name": "Book4",
      "names": [],
      "selection": "Sheet2!$A$1",
      "sheets": [
        "Sheet1"
      ]
    }
  ]
}
```

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

Example response:

```json
{}
```


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

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

Example response:

```
{
    "names": [
        {
            "name": "Sheet1!mynamel",
            "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": "=i+1.1",
    (continues on next page)
```
GET /apps/<pid>/books/<book_name_or_ix>/sheets

Example response:

```json
{
    "sheets": [
    {
        "charts": [
            "Chart 1"
        ],
        "name": "Sheet1",
        "names": [
            "Sheet1!myname1"
        ],
        "pictures": [
            "Picture 3"
        ],
        "shapes": [
            "Chart 1",
            "Picture 3"
        ],
        "used_range": "$A$1:$B$2"
    },
    {
        "charts": [],
        "name": "Sheet2",
        "names": [],
        "pictures": [],
        "shapes": [],
        "used_range": "$A$1"
    }
    ]
}
```
GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>

Example response:

```json
{
    "charts": [
        "Chart 1",
        "name": "Sheet1",
        "names": [
            "Sheet1!mynamel"
        ],
        "pictures": [
            "Picture 3"
        ],
        "shapes": [
            "Chart 1",
            "Picture 3"
        ],
        "used_range": "$A$1:$B$2"
    ]
}
```

GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/charts

Example response:

```json
{
    "charts": [
        {
            "chart_type": "line",
            "height": 211.0,
            "left": 0.0,
            "name": "Chart 1",
            "top": 0.0,
            "width": 355.0
        }
    ]
}
```

GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/charts/<chart_name_or_ix>

Example response:

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

GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/names

Example response:

```json
{}
```json
{
    "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:

```json
{
    "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:

```json
{
    "address": "$B$2:$C$3",
    "color": null,
    "column": 2,
    "column_width": 8.47,
    "count": 4,
    "current_region": "$A$1:$B$2",
    "formula": [
        [
            "",
            ""
        ],
        [
            "",
            ""
        ],
        [
            "",
            ""
        ],
        [
            "",
            ""
        ]
    ],
    "formula_array": "",
    "height": 28.5,
    "last_cell": "$C$3",
    "left": 51.0,
    "name": "Sheet1!myname1",
    "number_format": "General",
    "row": 2,
    "row_height": 14.3,
    "shape": [2, 2],
    "size": 4,
    "top": 14.25,
}
```

(continues on next page)
"value": [
  [
    null,
    null
  ],
  [
    null,
    null
  ],
  "width": 102.0
}

GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/pictures

Example response:

{
  "pictures": [
    {
      "height": 100.0,
      "left": 0.0,
      "name": "Picture 3",
      "top": 0.0,
      "width": 100.0
    }
  ]
}

GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/pictures/<picture_name_or_ix>

Example response:

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

GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/range

Example response:

{
  "address": "$A$1:$B$2",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A$1:$B$2",
  "height": 100.0,
  "left": 0.0,
  "name": "Picture 3",
  "top": 0.0,
  "width": 100.0
}

(continues on next page)
"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": "$A1:$B$2",
  "color": null,
  "column": 1,
  "column_width": 8.47,
  "count": 4,
  "current_region": "$A1:$B$2",
  "formula": [
    ["=1+1.1",
     "a string"
  ],
  ["2.1",
   "a string"
],
  ["Mon, 22 Oct 2018 00:09:18 GMT",
   null
]
],
"width": 102.0
}

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

Example response:

```json
{
  "shapes": [
    {
      "height": 211.0,
      "left": 0.0,
      "name": "Chart 1",
      "top": 0.0,
      "type": "chart",
      "width": 355.0
    },
    {
      "height": 100.0,
      "left": 0.0,
      "name": "Picture 3",
      "top": 0.0,
      "width": 355.0
    }
  ]
}
```
GET /apps/<pid>/books/<book_name_or_ix>/sheets/<sheet_name_or_ix>/shapes/<shape_name_or_ix>

Example response:

```json
{
    "height": 211.0,
    "left": 0.0,
    "name": "Chart 1",
    "top": 0.0,
    "type": "chart",
    "width": 355.0
}
```
count (xlwings.RangeColumns attribute), 110
count (xlwings.RangeRows attribute), 109
create_report () (in module xlwings.pro.reports), 123
current_region (xlwings.Range attribute), 102

data_body_range (xlwings.main.Table attribute), 119
delete() (xlwings.Chart method), 114
delete() (xlwings.Name method), 117
delete() (xlwings.Picture method), 116
delete() (xlwings.Shape method), 111
delete() (xlwings.Sheet method), 98
display_alerts (xlwings.App attribute), 89
display_name (xlwings.main.Table attribute), 119

delete() (xlwings.Range method), 102
data_body_range (xlwings.main.Table attribute), 119
delete() (xlwings.Range method), 102
delete() (xlwings.Picture method), 116
delete() (xlwings.Shape method), 111
delete() (xlwings.Sheet method), 98
display_alerts (xlwings.App attribute), 89
display_name (xlwings.main.Table attribute), 119

delete() (xlwings.Range method), 102
data_body_range (xlwings.main.Table attribute), 119
delete() (xlwings.Range method), 102
delete() (xlwings.Picture method), 116
delete() (xlwings.Shape method), 111
delete() (xlwings.Sheet method), 98
display_alerts (xlwings.App attribute), 89
display_name (xlwings.main.Table attribute), 119

end() (xlwings.Range method), 102
expand() (xlwings.Range method), 102

formula (xlwings.Range attribute), 103
formula2 (xlwings.Range attribute), 103
formula_array (xlwings.Range attribute), 103
fullname (xlwings.Book attribute), 94

get_address() (xlwings.Range method), 103

has_array (xlwings.Range attribute), 104
header_row_range (xlwings.main.Table attribute), 119
height (xlwings.Chart attribute), 114
height (xlwings.Picture attribute), 116
height (xlwings.Range attribute), 104
height (xlwings.Shape attribute), 111
hwnd (xlwings.App attribute), 90
hyperlink (xlwings.Range attribute), 104

index (xlwings.Sheet attribute), 98
insert() (xlwings.Range method), 104
insert_row_range (xlwings.main.Table attribute), 119

K
keys() (xlwings.main.Apps method), 88
kill() (xlwings.App method), 90

L
last_cell (xlwings.Range attribute), 105
left (xlwings.Chart attribute), 114
left (xlwings.Picture attribute), 116
left (xlwings.Range attribute), 105
left (xlwings.Shape attribute), 111

M
macro() (xlwings.App method), 90
macro() (xlwings.Book method), 94
merge() (xlwings.Range method), 105
merge_area (xlwings.Range attribute), 105
merge_cells (xlwings.Range attribute), 105

N
Name (class in xlwings), 117
name (xlwings.Book attribute), 95
name (xlwings.Chart attribute), 114
name (xlwings.main.Table attribute), 119
name (xlwings.Name attribute), 118
name (xlwings.Picture attribute), 116
name (xlwings.Range attribute), 105
name (xlwings.Shape attribute), 111
name (xlwings.Sheet attribute), 98
Names (class in xlwings.main), 117
names (xlwings.Book attribute), 95
names (xlwings.Sheet attribute), 98
number_format (xlwings.Range attribute), 105

O
offset() (xlwings.Range method), 106
open() (xlwings.main.Books method), 92
options() (xlwings.Range method), 106

P
parent (xlwings.Chart attribute), 114
parent (xlwings.main.Table attribute), 120
parent (xlwings.Picture attribute), 116
parent (xlwings.Shape attribute), 111
paste() (xlwings.Range method), 106
Picture (class in xlwings), 115
Pictures (class in xlwings.main), 114
pictures (xlwings.Sheet attribute), 98
pid (xlwings.App attribute), 90
<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td><code>quit()</code> <em>(xlwings.App method)</em></td>
<td>90</td>
</tr>
<tr>
<td>R</td>
<td><code>Range</code> <em>(class in xlwings)</em></td>
<td>99</td>
</tr>
<tr>
<td></td>
<td><code>range</code> <em>(xlwings.main.Table attribute)</em></td>
<td>120</td>
</tr>
<tr>
<td></td>
<td><code>range()</code> <em>(xlwings.App method)</em></td>
<td>90</td>
</tr>
<tr>
<td></td>
<td><code>range()</code> <em>(xlwings.Sheet method)</em></td>
<td>98</td>
</tr>
<tr>
<td></td>
<td><code>RangeColumns</code> <em>(class in xlwings)</em></td>
<td>109</td>
</tr>
<tr>
<td></td>
<td><code>RangeRows</code> <em>(class in xlwings)</em></td>
<td>109</td>
</tr>
<tr>
<td></td>
<td><code>raw_value</code> <em>(xlwings.Range attribute)</em></td>
<td>107</td>
</tr>
<tr>
<td></td>
<td><code>refers_to</code> <em>(xlwings.Name attribute)</em></td>
<td>118</td>
</tr>
<tr>
<td></td>
<td><code>refers_to_range</code> <em>(xlwings.Name attribute)</em></td>
<td>118</td>
</tr>
<tr>
<td></td>
<td><code>resize()</code> <em>(xlwings.Range method)</em></td>
<td>107</td>
</tr>
<tr>
<td></td>
<td><code>row</code> <em>(xlwings.Range attribute)</em></td>
<td>107</td>
</tr>
<tr>
<td></td>
<td><code>row_height</code> <em>(xlwings.Range attribute)</em></td>
<td>107</td>
</tr>
<tr>
<td></td>
<td><code>rows</code> <em>(xlwings.Range attribute)</em></td>
<td>108</td>
</tr>
<tr>
<td>S</td>
<td><code>save()</code> <em>(xlwings.Book method)</em></td>
<td>95</td>
</tr>
<tr>
<td></td>
<td><code>scale_height()</code> <em>(xlwings.Shape method)</em></td>
<td>111</td>
</tr>
<tr>
<td></td>
<td><code>scale_width()</code> <em>(xlwings.Shape method)</em></td>
<td>111</td>
</tr>
<tr>
<td></td>
<td><code>screen_updating</code> <em>(xlwings.App attribute)</em></td>
<td>90</td>
</tr>
<tr>
<td></td>
<td><code>select()</code> <em>(xlwings.Range method)</em></td>
<td>108</td>
</tr>
<tr>
<td></td>
<td><code>select()</code> <em>(xlwings.Sheet method)</em></td>
<td>99</td>
</tr>
<tr>
<td></td>
<td><code>selection</code> <em>(xlwings.App attribute)</em></td>
<td>91</td>
</tr>
<tr>
<td></td>
<td><code>set_mock_caller()</code> <em>(xlwings.Book method)</em></td>
<td>95</td>
</tr>
<tr>
<td></td>
<td><code>set_source_data()</code> <em>(xlwings.Book method)</em></td>
<td>114</td>
</tr>
<tr>
<td></td>
<td><code>Shape</code> <em>(class in xlwings)</em></td>
<td>110</td>
</tr>
<tr>
<td></td>
<td><code>shape</code> <em>(xlwings.Range attribute)</em></td>
<td>108</td>
</tr>
<tr>
<td></td>
<td><code>Shapes</code> <em>(class in xlwings.main)</em></td>
<td>110</td>
</tr>
<tr>
<td></td>
<td><code>shapes</code> <em>(xlwings.Sheet attribute)</em></td>
<td>99</td>
</tr>
<tr>
<td></td>
<td><code>Sheet</code> <em>(class in xlwings)</em></td>
<td>97</td>
</tr>
<tr>
<td></td>
<td><code>sheet</code> <em>(xlwings.Range attribute)</em></td>
<td>108</td>
</tr>
<tr>
<td></td>
<td><code>Sheets</code> <em>(class in xlwings.main)</em></td>
<td>96</td>
</tr>
<tr>
<td></td>
<td><code>sheets</code> <em>(xlwings.Book attribute)</em></td>
<td>96</td>
</tr>
<tr>
<td></td>
<td><code>show Autofilter</code> <em>(xlwings.main.Table attribute)</em></td>
<td>120</td>
</tr>
<tr>
<td></td>
<td><code>show_headers</code> <em>(xlwings.main.Table attribute)</em></td>
<td>120</td>
</tr>
<tr>
<td></td>
<td><code>show_table_style_column_stripes</code> <em>(xlwings.main.Table attribute)</em></td>
<td>120</td>
</tr>
<tr>
<td></td>
<td><code>show_table_style_first_column</code> <em>(xlwings.main.Table attribute)</em></td>
<td>120</td>
</tr>
<tr>
<td></td>
<td><code>show_table_style_last_column</code> <em>(xlwings.main.Table attribute)</em></td>
<td>120</td>
</tr>
<tr>
<td></td>
<td><code>show_table_style_row_stripes</code> <em>(xlwings.main.Table attribute)</em></td>
<td>120</td>
</tr>
<tr>
<td></td>
<td><code>size</code> <em>(xlwings.Range attribute)</em></td>
<td>120</td>
</tr>
<tr>
<td></td>
<td><code>startup_path</code> <em>(xlwings.App attribute)</em></td>
<td>108</td>
</tr>
<tr>
<td></td>
<td><code>status_bar</code> <em>(xlwings.App attribute)</em></td>
<td>91</td>
</tr>
<tr>
<td>T</td>
<td><code>Table</code> <em>(class in xlwings.main)</em></td>
<td>119</td>
</tr>
<tr>
<td></td>
<td><code>table</code> <em>(xlwings.Range attribute)</em></td>
<td>108</td>
</tr>
<tr>
<td></td>
<td><code>table_style</code> <em>(xlwings.main.Table attribute)</em></td>
<td>120</td>
</tr>
<tr>
<td></td>
<td><code>Tables</code> <em>(class in xlwings.main)</em></td>
<td>118</td>
</tr>
<tr>
<td></td>
<td><code>tables</code> <em>(xlwings.Sheet attribute)</em></td>
<td>99</td>
</tr>
<tr>
<td></td>
<td><code>text</code> <em>(xlwings.Shape attribute)</em></td>
<td>111</td>
</tr>
<tr>
<td></td>
<td><code>to_pdf()</code> <em>(xlwings.Book method)</em></td>
<td>96</td>
</tr>
<tr>
<td></td>
<td><code>top</code> <em>(xlwings.Chart attribute)</em></td>
<td>114</td>
</tr>
<tr>
<td></td>
<td><code>top</code> <em>(xlwings.Picture attribute)</em></td>
<td>116</td>
</tr>
<tr>
<td></td>
<td><code>top</code> <em>(xlwings.Range attribute)</em></td>
<td>108</td>
</tr>
<tr>
<td></td>
<td><code>top</code> <em>(xlwings.Shape attribute)</em></td>
<td>112</td>
</tr>
<tr>
<td></td>
<td><code>totals_row_range</code> <em>(xlwings.main.Table attribute)</em></td>
<td>120</td>
</tr>
<tr>
<td></td>
<td><code>type</code> <em>(xlwings.Shape attribute)</em></td>
<td>112</td>
</tr>
<tr>
<td>U</td>
<td><code>unmerge()</code> <em>(xlwings.Range method)</em></td>
<td>108</td>
</tr>
<tr>
<td></td>
<td><code>update()</code> <em>(xlwings.main.Table method)</em></td>
<td>120</td>
</tr>
<tr>
<td></td>
<td><code>update()</code> <em>(xlwings.Picture method)</em></td>
<td>116</td>
</tr>
<tr>
<td></td>
<td><code>used_range</code> <em>(xlwings.Sheet attribute)</em></td>
<td>99</td>
</tr>
<tr>
<td>V</td>
<td><code>value</code> <em>(xlwings.Range attribute)</em></td>
<td>108</td>
</tr>
<tr>
<td></td>
<td><code>version</code> <em>(xlwings.App attribute)</em></td>
<td>91</td>
</tr>
<tr>
<td></td>
<td><code>view()</code> <em>(in module xlwings)</em></td>
<td>87</td>
</tr>
<tr>
<td></td>
<td><code>visible</code> <em>(xlwings.App attribute)</em></td>
<td>91</td>
</tr>
<tr>
<td></td>
<td><code>visible</code> <em>(xlwings.Sheet attribute)</em></td>
<td>99</td>
</tr>
<tr>
<td>W</td>
<td><code>width</code> <em>(xlwings.Chart attribute)</em></td>
<td>114</td>
</tr>
<tr>
<td></td>
<td><code>width</code> <em>(xlwings.Picture attribute)</em></td>
<td>116</td>
</tr>
<tr>
<td></td>
<td><code>width</code> <em>(xlwings.Range attribute)</em></td>
<td>108</td>
</tr>
<tr>
<td></td>
<td><code>width</code> <em>(xlwings.Shape attribute)</em></td>
<td>112</td>
</tr>
<tr>
<td>X</td>
<td><code>xlwings</code> <em>(module)</em></td>
<td>87</td>
</tr>
<tr>
<td></td>
<td><code>xlwings.arg()</code> <em>(in module xlwings)</em></td>
<td>121</td>
</tr>
<tr>
<td></td>
<td><code>xlwings.func()</code> <em>(in module xlwings)</em></td>
<td>121</td>
</tr>
</tbody>
</table>
xlwings.pro.reports (module), 123
xlwings.ret() (in module xlwings), 122
xlwings.sub() (in module xlwings), 121