PyXLL User Guide

Release 5.13.0

PyXLL Ltd.

Oct 20, 2026

CONTENTS

1 Getting Started with PyXLL **1**
- 1.1 Install PyXLL . . . 1
- 1.2 Write your first Excel function . . . 1
- 1.3 Explore the built-in examples . . . 3
- 1.4 Troubleshooting . . . 3
- 1.5 Next Steps . . . 4

2 What's new in PyXLL 5 **5**
- 2.1 New Features and Improvements . . . 5
- 2.2 Important notes for upgrading from previous versions . . . 7

3 User Guide **9**
- 3.1 Installing PyXLL . . . 9
- 3.2 Configuring PyXLL . . . 17
- 3.3 Worksheet Functions . . . 49
- 3.4 Macro Functions . . . 100
- 3.5 Real Time Data . . . 108
- 3.6 Using Type Hints . . . 117
- 3.7 Cell Formatting . . . 121
- 3.8 Charts and Plotting . . . 128
- 3.9 Excel Application Events . . . 144
- 3.10 Custom Task Panes . . . 149
- 3.11 ActiveX Controls . . . 161
- 3.12 Using Pandas in Excel . . . 176
- 3.13 Customizing the Ribbon . . . 187
- 3.14 Context Menu Functions . . . 191
- 3.15 Working with Tables . . . 195
- 3.16 Python as a VBA Replacement . . . 200
- 3.17 Menu Functions . . . 207
- 3.18 Reloading and Rebinding . . . 210
- 3.19 Error Handling . . . 213
- 3.20 Deploying your add-in . . . 219
- 3.21 Workbook Metadata . . . 226

4 API Reference **227**
- 4.1 Worksheet Functions . . . 227
- 4.2 Real Time Data . . . 230
- 4.3 Macro Functions . . . 233
- 4.4 Type Conversion . . . 239
- 4.5 Ribbon Functions . . . 242
- 4.6 Menu Functions . . . 245
- 4.7 Plotting . . . 246
- 4.8 Custom Task Panes . . . 248
- 4.9 ActiveX Controls . . . 252

4.10 Cell Formatting . . . 255
4.11 Tables . . . 259
4.12 Errors and Exceptions . . . 262
4.13 Utility Functions . . . 264
4.14 LRU Cache . . . 268
4.15 Excel Application Events . . . 269
4.16 PyXLL Add-in Events . . . 280
4.17 Excel C API Functions . . . 282

Index **285**

CHAPTER

ONE

GETTING STARTED WITH PYXLL

PyXLL is an Excel add-in that lets you write Excel worksheet functions, macros, and more using Python. This guide takes you from a fresh install to your first working Python function in Excel.

1.1 Install PyXLL

Open a command prompt, activate your Python virtual environment if necessary, and run:

```
pip install pyxll
```

Followed by:

```
pyxll install
```

and follow the on-screen instructions.

- When asked whether you have already downloaded PyXLL, enter `n` and the installer will download the right version for you.
- You do **not** need a license key. When asked if you have a license key, enter `n` and the free 30 day trial will begin automatically when the add-in loads.

For other installation options (including manual installation) see the *installation guide*.

> **Tip**
>
> pyxll command not recognized?
>
> `pip install pyxll` installs the `pyxll` command into your Python scripts folder, but if the Python scripts folder isn't on your path then the command won't be found.
>
> Instead, run `python -m pyxll install`. This will do the same thing, without needing the `pyxll` command to be installed on your system path.

1.2 Write your first Excel function

Once PyXLL is installed you can expose a Python function to Excel in four steps.

1.2.1 Step 1: Create a Python module

Create a new file, for example `my_functions.py`, containing a function decorated with *`@xl_func`*:

```
from pyxll import xl_func

@xl_func
def hello(name):
```

(continues on next page)

(continued from previous page)

```
    """Say hello to someone."""
    return "Hello, " + name
```

1.2.2 Step 2: Tell PyXLL to load your module

Open the PyXLL config file. The config file is the file named `pyxll.cfg` in the folder where you installed the PyXLL add-in. You can edit this with any text editor, such as notepad, or your Python code editor.

If you are not sure where the file is located, you can run `pyxll status` and that will tell you, or you can run `pyxll config` to open the config file using the associated editor.

The file is split into multiple sections. Look for the `[PYTHON]` section and in there you will find the `pythonpath` setting. This is a list of folders where Python will look for modules. Add the folder where you saved your new `my_functions.py` file so that it can be imported.

Next, look for the `[PYXLL]` section. In that section you will find the `modules` list. This is the list of Python modules (or packages) that will be loaded by the PyXLL add-in. Add your module name (`my_functions`) to this list. Note that a Python module name does not include the `.py` file extension.

```
[PYTHON]
pythonpath = C:\path\to\folder\containing\my_functions

[PYXLL]
modules = my_functions
```

1.2.3 Step 3: Reload PyXLL in Excel

In Excel, open the **PyXLL** menu and select **Reload**. This picks up your new module without restarting Excel.

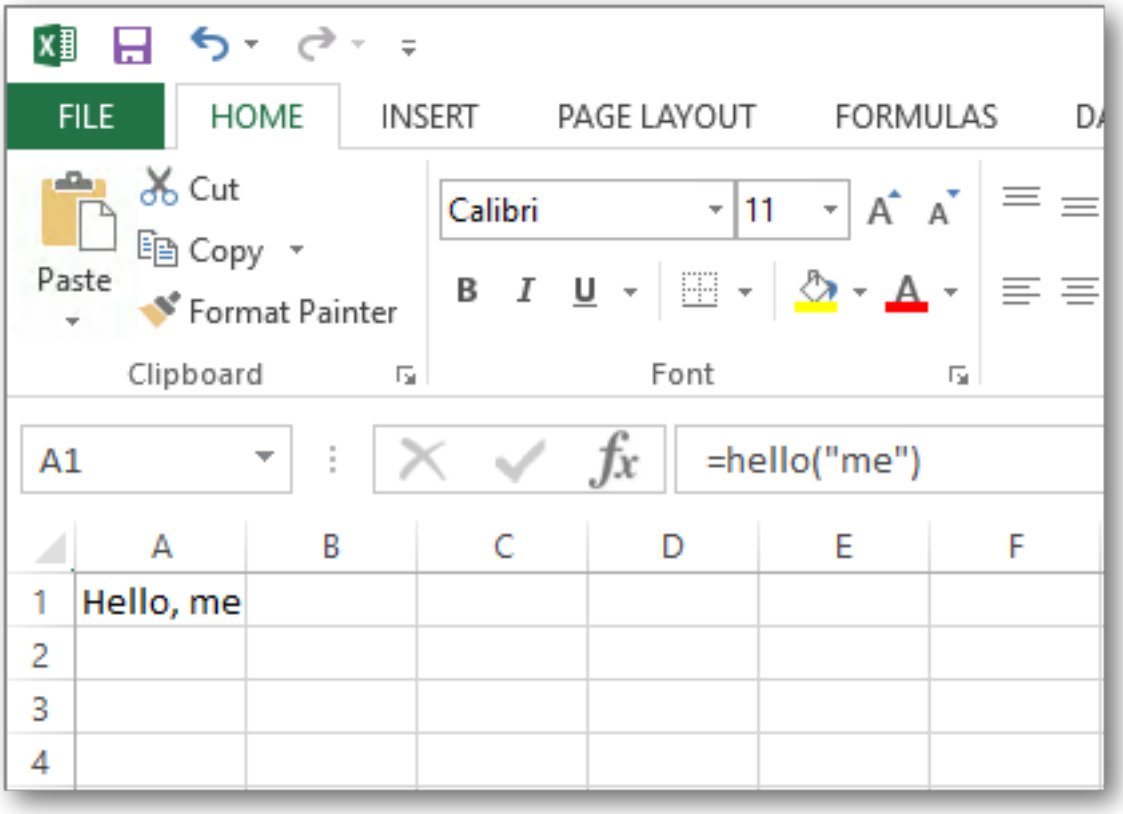
Each time your open Excel, your configured modules are loaded automatically.

> **Tip**
>
> If there are any errors at this stage, check the PyXLL log file
>
> By default it is in the `logs` folder inside your PyXLL installation directory. You can run `pyxll status` to find where PyXLL is installed if you are not sure where that is.

1.2.4 Step 4: Call your function from a cell

In any worksheet cell type:

```
=hello("World")
```



The *Worksheet Functions* section of the user guide explains the full range of what you can do with *`@xl_func`*, including type hints, array functions, and optional arguments.

1.3 Explore the built-in examples

The PyXLL installation includes an `examples` folder with an Excel workbook and Python source files demonstrating all major features — worksheet functions, macros, real-time data, plotting, ribbon toolbars, and more.

Open `examples.xlsx` to see them in action, and read the corresponding `.py` files to understand how each one works. Use `pyxll status` if you cannot remember where PyXLL was installed.

1.4 Troubleshooting

- **pip or the download fails due to a corporate firewall or proxy**

  Follow the *manual installation instructions* instead.

- **"The 'pyxll' command is not recognised" after pip install**

  Your Python `Scripts` folder may not be on your system path. Use `python -m pyxll install` instead - it works identically.

- **"pythonXXX.dll not found" when starting Excel**

  The PyXLL add-in was built for a different Python version. Let the command-line installer download the correct version by entering `n` when asked if you have already downloaded PyXLL.

- **Other issues**

  Check the PyXLL log file for the full error and Python stack trace. Search the FAQ or contact us for further help.

1.5 Next Steps

- *User Guide* — complete documentation of all PyXLL features
- videos — video tutorials covering installation, worksheet functions, and more
- *API Reference* — full Python API reference

CHAPTER

TWO

WHAT'S NEW IN PYXLL 5

Looking for an earlier version?

See 4.x/whatsnew for a detailed overview of the features added in PyXLL 4.

- *New Features and Improvements*
  - *Excel Tables*
  - *RTD Generators*
  - *Polars Types*
  - *Easier Installation*
  - *Custom Task Panes*
  - *Plotting Integrations*
  - *Serialized Cached Objects*
  - *Entry Points*
  - *Composite Ribbon Toolbars*
  - *Auto-Rebinding*
  - *Improved Cell Formatting*
  - *Log Rolling*
  - *LRU Cache*
- *Important notes for upgrading from previous versions*
  - *Updated Software License Agreement*
  - *Deep reloading is now enabled by default*
  - *RTD functions no longer recalculate on open by default*
  - *async_func has been replaced with schedule_call*

2.1 New Features and Improvements

2.1.1 Excel Tables

New in PyXLL 5.8

Read and write Excel tables from Python.

See *Working with Tables* for details.

2.1.2 RTD Generators

New in PyXLL 5.6

Writing an Excel RTD (Real Time Data) function is now as simple as writing a Python generator.

See *RTD Generators* for more details.

2.1.3 Polars Types

New in PyXLL 5.6

Polars DataFrames can be used as argument and return types, in the same way as pandas DataFrames.

See *Polars DataFrames* for more details.

2.1.4 Easier Installation

The PyXLL Excel add-in can now be installed and uninstalled via a new *command line tool*.

To install the PyXLL Excel add-in first use *pip install* to install the PyXLL wheel, eg

```
> pip install "pyxll >= 5.0.0"
```

Once the PyXLL wheel is installed the new *pyxll command line tool* can be used to install, configure and uninstall the PyXLL Excel add-in, eg

```
> pyxll install
```

See *PyXLL Command Line Tool*.

2.1.5 Custom Task Panes

Task Panes are Excel windows that can be floating or docked as part of the Excel user interface.

PyXLL 5 adds the capability to write custom task panes in Python using any of the following Python UI toolkits:

- PySide2 and PySide6
- PyQt5 and PyQt5
- wxWindows
- tkinter

See *Custom Task Panes*.

2.1.6 Plotting Integrations

PyXLL 5 adds integration with the following Python plotting and charting packages:

- matplotlib
- plotly
- bokeh
- altair

See *Charts and Plotting*.

2.1.7 Serialized Cached Objects

Cached objects can be serialized and saved as part of the Excel workbook. When a workbook containing saved objects is opened they are deserialized and loaded into PyXLL's object cache.

To specify that an object should be saved use the `save` parameter to the `object` return type.

See *Saving Objects in the Workbook*.

2.1.8 Entry Points

Python packages can now be loaded by PyXLL via setuptools' `entry-points`.

This allows package developers to distribute functionality to other PyXLL users more easily as no additional PyXLL configuration is required when installing a package with PyXLL entry points.

See *Setuptools Entry Points*.

2.1.9 Composite Ribbon Toolbars

The ribbon toolbar can now be composed of multiple xml files instead of a single file.

The `ribbon` setting can now be a list of files, which PyXLL will merge into a single ribbon.

This can be used by package authors who want to add a ribbon to their package via an entry point without needing changes to be made to the main PyXLL configuration or ribbon xml file.

Images specified in the ribbon xml can now also be package resources as well as files.

2.1.10 Auto-Rebinding

When using the `@xl_func`, `@xl_macro` or `@xl_menu` decorators outside of the usual module imports as PyXLL is starting, PyXLL will automatically reflect these functions in Excel without needing to call `rebind`.

This simplifies working with adhoc worksheet functions from an interactive Python prompt in Excel, such as a Jupyter notebook.

2.1.11 Improved Cell Formatting

- Cell formatting can now be applied to RTD functions as well as standard worksheet functions.
- The `DataFrameFormatter` can now do conditional formatting based on the values in the returned DataFrame.

See *Conditional Formatting*.

2.1.12 Log Rolling

New in PyXLL 5.2

PyXLL can now automatically roll its log file when it reaches a certain size or after a specific interval has elapsed. This avoids long running Excel processes from generating huge log files. Old log files can be kept for a while and then later automatically cleaned up to avoid using excessive disk space.

See *Logging* for more details.

2.1.13 LRU Cache

New in PyXLL 5.11

The LRU, or *Least Recently Used*, cache avoids calling your Python functions when Excel recalculates but the arguments haven't changed.

This is useful in many situations where Excel is recalculating an expensive function unnecessarily (for example, if one of the inputs is volatile).

See *Cached Functions* for details.

2.2 Important notes for upgrading from previous versions

PyXLL 5.0 contains some changes that may require you to make changes to your code and/or config before upgrading from previous versions.

2.2.1 Updated Software License Agreement

The PyXLL software license agreement has been updated.

See terms-and-conditions or the software license agreement PDF file included in the PyXLL download for details.

2.2.2 Deep reloading is now enabled by default

This can be disabled for backwards compatibility

Deep reloading is now enabled by default. See *Reloading and Rebinding* for details about how PyXLL reloads modules.

To disable deep reloading set the following in your PyXLL config file.

```
[PYXLL]
deep_reload = 0
```

2.2.3 RTD functions no longer recalculate on open by default

This can be disabled for backwards compatibility

In previous versions of PyXLL RTD functions were implicitly marked as needed to be recalculated when opening a workbook. This was done to be consistent with earlier behaviour where RTD functions were registered as volatile.

As of PyXLL 5 RTD and standard functions behave in the same consistent way. That is, unless the `recalc_on_open=True` is passed to `@xl_func`, or defaulted via the config, RTD functions will not recalculate and start ticking when a workbook is opened automatically.

To enable recalculation on open as the default for all RTD functions you may set the following in your config file.

```
[PYXLL]
recalc_rtd_on_open = 1
```

2.2.4 async_func has been replaced with schedule_call

If your code uses `async_call` you should replace it with the new `schedule_call`. The old `async_call` is still available but has been deprecated and will log a warning if used.

CHAPTER

THREE

USER GUIDE

New to PyXLL? Start with our *Quickstart Guide*!

3.1 Installing PyXLL

Before you start you will need to have Microsoft Excel for Windows installed, as well as a compatible version of Python.

PyXLL works with any Python distribution, including *Anaconda*.

3.1.1 First Time Users

This page has moved. See Getting Started with PyXLL.

3.1.2 PyXLL Command Line Tool

The PyXLL command line tool automates tasks around installing, updating and switching between different versions of PyXLL.

In order to use the PyXLL command line tool you first need to install it using pip:

```
>> pip install pyxll
```

If you are using a conda or virtual env you should activate the environment you want to use first.

To get the latest version of the PyXLL command line too you should update the package using pip:

```
>> pip install --upgrade pyxll
```

Many commands take a `--user` switch, which can be used if installing or managing the Excel add-in on behalf of a different user[3].

The PyXLL wheel file is also included in the PyXLL download and may be installed from there.

After installing, the following commands are available:

- *pyxll install*
- *pyxll configure*
- *pyxll status*
- *pyxll update*
- *pyxll activate*
- *pyxll install-certificate*

[3] The `--user` option was added in PyXLL 5.12.0.

- *pyxll uninstall*

pyxll install

The *install* command installs the PyXLL Excel add-in into Excel. It is necessary to either perform this step or to install PyXLL manually before the PyXLL Excel add-in can be used.

```
>> pyxll install [OPTIONS] [PATH]
```

Options:	
`--version`	Version of PyXLL to install.
`--user`	Install the Excel add-in for a different user.[Page 9, 3]
`--debug` / `-d`	Output more information when running the command.
`--install-first`	Install the PyXLL add-in as the first add-in to be loaded.[2]

- Can be run with or without *PATH*.
  - If *PATH* is not specified then either the latest version of PyXLL or the version specified will be downloaded. You will be prompted for some details in order to complete the download.
  - If *PATH* is specified it can be a zip file downloaded from the download page, or a folder containing the extracted downloaded zip file.
- If you already have PyXLL installed you will be warned but may continue.
  - Any existing files that will be over-written will be backed up.
  - You will be given the choice to change the location of the installation, allowing you to maintain multiple copies of PyXLL.
  - If installing in the same folder as your existing installation, your existing config file will be backed up and a new one will be created with the default configuration.
- PyXLL will be configured automatically to use the active Python environment.

  Further configuration can be performed by editing the pyxll.cfg file included in the installation or by using the *pyxll configure* command.

pyxll configure

The *configure* command opens the pyxll.cfg configuration file for the currently active PyXLL addin.

```
>> pyxll configure [OPTIONS]
```

Options:	
`--user`	Use the Excel add-in installed for a different user.[Page 9, 3]
`--debug` / `-d`	Output more information when running the command.

- The default editor for the file type *.cfg* will be used to open the config file.
- PyXLL can have been installed using *pyxll install* or manually.

pyxll status

The *status* command checks the status of the active PyXLL installation and reports information about it.

[2] The `--install-first` option was added in PyXLL 5.10.0.

```
>> pyxll status [OPTIONS]
```

Options:	
`--user`	Use the Excel add-in installed for a different user.[Page 9, 3]
`--debug` / `-d`	Output more information when running the command.

If there are any issues with your current PyXLL installation this command may help identify what the problem is.

pyxll update

The *update* command updates your active PyXLL installation to the latest version of PyXLL.

```
>> pyxll update [OPTIONS] [PATH]
```

Options:	
`--version`	Version of PyXLL to update to.
`--user`	Update the Excel add-in for a different user.[Page 9, 3]
`--force`	For the update, even if the installed version is newer.
`--debug` / `-d`	Output more information when running the command.

- Your existing *pyxll.cfg* file will not be modified.
- The previous *pyxll.xll* file will be backed up.
- Can be run with or without *PATH*.
  - If *PATH* is not specified then either the latest version of PyXLL or the version specified will be downloaded. You will be prompted for some details in order to complete the download.
  - If *PATH* is specified it can be a zip file downloaded from the download page, or a folder containing the extracted downloaded zip file.
- If you want to try out a new version of PyXLL *before* upgrading use the *pyxll install* command and specify a different folder to install it to. You can use the *pyxll activate* command to switch between installs easily.

pyxll activate

The *activate* command switches between different PyXLL installations quickly.

```
>> pyxll activate [OPTIONS] [PATH]
```

Options:	
`--user`	Activate the Excel add-in for a different user.[Page 9, 3]
`--debug` / `-d`	Output more information when running the command.
`--non-interactive` / `-ni`	Don't prompt the user for any input.[1]
`--install-first`	Install the PyXLL add-in as the first add-in to be loaded.[Page 10, 2]

- You can maintain multiple versions of PyXLL at the same time by installing PyXLL into different folders.
- This command selects which PyXLL add-in is active in Excel and does not change any files or configuration.
- Can be run with or without *PATH*.
  - If *PATH* is not specified then it will look for *pyxll.xll* in the current working directory and activate that, or prompt for a path if *pyxll.xll* is not found.

[1] The `--non-interactive` option is new in PyXLL 5.3.0 and enables `pyxll activate` to be used from a script more easily for automated deployment of PyXLL environments.

- If *PATH* is specified it should be a folder containing the PyXLL add-in to be activated.

pyxll install-certificate

Installs the PyXLL certificate into the 'Trusted Publishers' certificate store.

```
>> pyxll install-certificate [OPTIONS]
```

Options:	
`--debug` / `-d`	Output more information when running the command.

Installing the certificate is done as part of installing PyXLL but can also be done using this command (for example, if installing the certificate failed during the initial install).

If the certificate can't be installed then Excel may prompt the user that the add-in is unsafe or prevent it from loading, depending on Excel's Trust Center Settings.

pyxll uninstall

The *uninstall* command uninstalls the PyXLL Add-In from Excel.

```
>> pyxll uninstall
```

Options:	
`--user`	Uninstall the Excel add-in for a different user.[Page 9, 3]
`--force`	Uninstall without any confirmation.
`--dry-run`	Log what would happen without actually uninstalling.
`--debug` / `-d`	Output more information when running the command.

- This command only uninstalls the PyXLL add-in from Excel.
- No files will be deleted.
- To reinstall the same PyXLL add-in run *pyxll activate*.

3.1.3 Manual Installation

Before you start you will need to have Microsoft Excel for Windows installed, as well as a compatible version of Python.

PyXLL works with any Python distribution, including Anaconda. For specific instructions about installing with Anaconda or Miniconda see *Using PyXLL with Anaconda*.

> **Warning**
>
> These instructions are for **manually** installing the PyXLL Excel Add-In.
>
> You may find it more convenient to use our *command line tool* for installing or upgrading PyXLL.

1. Download the PyXLL Zipfile

PyXLL comes as a zipfile you download from the download page. Select and download the correct version depending on the versions of Python and Excel you want to use and agree to the terms and conditions.

Warning

Excel, Python and PyXLL all come in 64-bit and 32-bit versions.

The three products must be **all** *32-bit or* **all** *64-bit.*

2. Unpack the Zipfile

PyXLL is packaged as a zip file. Unpack the zip file where you want PyXLL to be installed.

There is no installer to run; you complete the installation in Excel after any necessary configuration changes.

3. Edit the Config File

You configure PyXLL by editing the *pyxll.cfg* file. Any text editor will do.

Set the *executable* setting in the *PYTHON* section of your config file to the full path to your Python executable.

pythonw.exe or python.exe

You may have noticed we've used *pythonw.exe* instead of *python.exe*.

The only difference between the two is that *pythonw.exe* doesn't open a console window and so using that means that we don't see a console window is a Python subprocess is started (e.g. if using the *subprocess* or *multiprocessing* Python packages).

If you prefer to use *python.exe* then that will work fine too.

```
[PYTHON]
executable = <path to your pythonw.exe>
```

PyXLL uses this setting to determine where the Python runtime libraries and Python packages are located.

You can determine where the executable for an installed Python interpreter with the command:

```
pythonw -c "import sys; print(sys.executable)"
```

While you have the *pyxll.cfg* file open take look through and see what other options are available.

You can find documentation for all available options in the *Configuring PyXLL* section of the user guide.

One important section of the config file is the *LOG* section. In there you can set where PyXLL should log to and the logging level. If you are having trouble, set the log verbosity to *debug* to get more detailed logging.

```
[LOG]
verbosity = debug
```

Warning

The ";" character is used to comment out lines in the config file.

If a line starts with ";" then it will not be read by PyXLL.

4. Install the Add-In in Excel

DLL not found

If you get an error saying that Python is not installed or the Python dll can't be found you may need to set the Python executable in the config.

If setting the executable doesn't resolve the problem then it's possible your Python dll is in a non-standard location. You can set the dll location in the config to tell PyXLL where to find it.

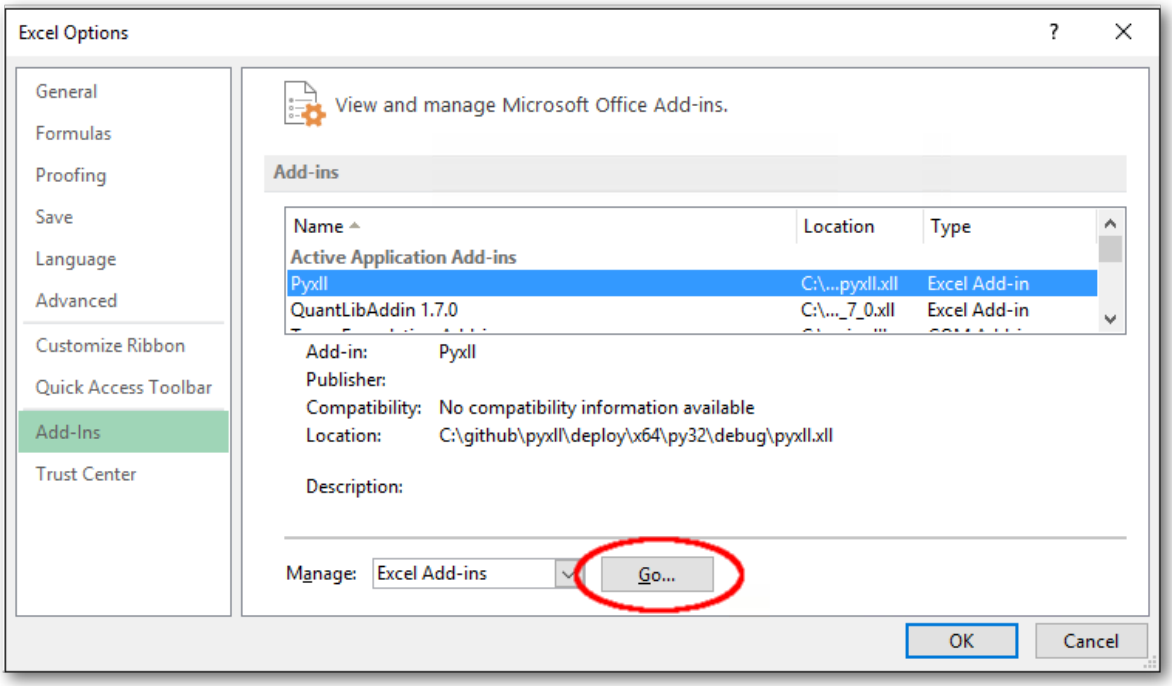
Once you're happy with the configuration you can install the add-in in Excel by following the instructions below.

- **Excel 2010 - 2019 / Office 365**
  Select the File menu in Excel and go to *Options -> Add-Ins -> Manage Excel Addins* and browse for the folder you unpacked PyXLL to and select pyxll.xll.
- **Excel 2007**
  Click the large circle at the top left of Excel and go to *Options -> Add-Ins -> Manage Excel Addins* and browse for the folder you unpacked PyXLL to and select pyxll.xll.
- **Excel 97 - 2003**
  Go to *Tools -> Add-Ins -> Browse* and locate pyxll.xll in the folder you unpacked the zip file to.

Warning

If Excel prompts you to ask if you want to copy the add-in to your local add-ins folder then select **No**.

When PyXLL loads it expects its config file to be in the same folder as the add-in, and if Excel copies it to your local add-ins folder then it won't be able to find its config file.



5. Install the PyXLL Stubs Package *(Optional)*

If you are using a Python IDE that provides autocompletion or code checking or if you want to execute your code outside Excel, say for testing purposes, you will need to install the pyxll module to avoid your code raising `ImportError` exceptions.

In the downloaded zip file you will find a *.whl* file whose exact filename depends on the version of PyXLL. That's a Python Wheel containing a dummy pyxll module that you can import when testing without PyXLL. You can then use code that depends on the pyxll module outside of Excel (e.g. when unit testing).

To install the wheel run the following command (substituting the actual wheel filename) from a command line:

```
> cd C:\Path\Where\You\Unpacked\PyXLL
> pip install "pyxll-wheel-filename.whl"
```

The real pyxll module is compiled into the pyxll.xll addin, and so is always available when your code is running inside Excel.

If you are using a version of Python that doesn't support pip you can instead unzip the *.whl* file into your Python site-packages folder (the wheel file is simply a zip file with a different file extension).

Next Steps

Now you have PyXLL installed you can start adding your own Python code to Excel.

See *Worksheet Functions* for details of how you can expose your own Python functions to Excel as worksheet functions, or browse the *User Guide* for information about the other features of PyXLL.

3.1.4 Using PyXLL with Anaconda

- *What is Anaconda*
- *Which Anaconda Distribution to Choose*
- *Creating a Virtual Environment (optional)*
- *Installing PyXLL with Anaconda*
- *Switching Virtual Environments*

What is Anaconda

Anaconda is an open source Python distribution that aims to simplify Package management and distribution.

The Anaconda distribution includes over a thousand Python packages as well as its own package and virtual environment manager, Conda.

For users wanting just the package and virtual environment manager, Conda, without the large download and install size of the full Anaconda distribution, there is also Miniconda.

Both Anaconda and Miniconda work well with PyXLL.

Which Anaconda Distribution to Choose

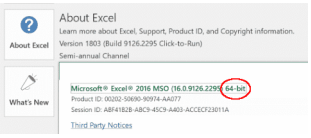
PyXLL will work fine with any Anaconda or Miniconda distribution for Windows. Note that PyXLL only supports Microsoft Windows and will not work on macOS.

When downloading Anaconda you are given the choice between Python 2 and Python 3. All current Python versions are supported by PyXLL, and so you are free to choose whichever version is right for you.

The Anaconda download page also offers the choice between a 64 bit installer and a 32 bit installed. The 64 bit installer is the default selection, but which one you need depends on the version of Excel you are using.

It is not possible to use a 64 bit Python environment with the 32 bit version of Excel.

To determine which version of Excel you are using, in Excel go to File -> Account -> About.



If your Excel version does not inclide "64-bit" as shown above, you are using the 32 bit version of Excel and will need to download the 32 bit version of Anaconda or Miniconda.

Creating a Virtual Environment (optional)

When using Anaconda or Miniconda it's recommended to work within a virtual environment.

A virtual environment is a Python environment where you can install and update packages without modifying the base Python install. You can have multiple environments at any time, so you could have a virtual environment dedicated to everything you do in Excel with PyXLL without having to change any other environments you might have for other tasks.

Virtual environments are created using the "conda create" command.

For example, to create a Python 3.7 environment for use with PyXLL named "pyxll", start an Anaconda command prompt and run the following:

```
>> conda create -n pyxll python=3.7
```

This will create a new Python 3.7 environment called "pyxll" (the name can be anything, it doesn't have to be pyxll).

You then have to activate that environment and install the packages you want:

```
>> activate env
>> conda install pandas
```

To see what environments you have, use "conda info –envs". That will give you the path to where the new pyxll environment has been created.

Installing PyXLL with Anaconda

PyXLL can be used with the Anaconda and Miniconda distributions. Use of either a virtual env or the base Python environment is supported.

Follow the *installation instructions* to install PyXLL.

If you are using the *PyXLL Command Line Tool* then be sure to activate your conda environment first.

```
(base)  >> activate env
(env)   >> pip install pyxll
(env)   >> pyxll install
```

If you are installing the PyXLL add-in manually then edit your *pyxll.cfg* file so that the *executable* setting references the Python executable from your conda environment:

```
[PYTHON]
executable = C:\Program Files\Anaconda\envs\pyxll\pythonw.exe
```

To determine what Python executable to use, open an Anaconda Command prompt and activate the virtual environment you want to use and type "where pythonw":

```
(base)  >> activate env
(env)   >> where pythonw
C:/Program Files/Anaconda/envs/env/pythonw.exe
```

Switching Virtual Environments

To change the virtual environment that PyXLL uses from the one your originally configured, simply update your *pyxll.cfg* config file to use the new virtual env and restart Excel.

Don't forget that you may also need to install the pyxll stubs package in the new virtual environment if you require code completion in your IDE, or if you are importing pyxll outside of Excel for any other reason.

3.2 Configuring PyXLL

Finding the config file

In PyXLL's *About* dialog it displays the full path to the config file in use. Clicking on the path will open the config file in your default editor.

The PyXLL config is available to your addin code at run-time via `get_config`.

If you add your own sections to the config file they will be ignored by PyXLL but accessible to your code via the config object.

If you've not installed the PyXLL addin yet, see *Installing PyXLL*.

The config file is a plain text file that should be kept in the same folder as the PyXLL addin .xll file, and should have the same name as the addin but with a `.cfg` extension. In most cases it will simply be `pyxll.cfg`.

You can load the config file from an alternative location by setting the environment variable `PYXLL_CONFIG_FILE` to the full path of the config file you wish to load before starting Excel.

Paths used in the config file may be absolute or relative. The latter (those not beginning with a slash) are interpreted relative to the directory containing the config file.

There are various *functions* that can be used in config settings, as well as *variable substitions*.

Warning

Lines beginning with a semicolon are ignored as comments.

When setting a value in the configuration file, make sure there is no leading semicolon or your changes will have no effect.

```
THIS WILL HAVE NO EFFECT
;setting = value

SETTING IS EFFECTIVE WITH NO SEMICOLON
setting = value
```

3.2.1 Python Settings

```
[PYTHON]
;
; Python settings
;
pythonpath = semi-colon or new line delimited list of directories
executable = full path to the Python executable (python.exe)
dll = full path to the Python dynamic link library (pythonXX.dll)
pythonhome = location of the standard Python libraries
ignore_environment = ignore environment variables when initializing Python
inspect = see sys.flags.inspect (default is 1)
optimize = see sys.flags.optimize (default is 0)
debug = see sys.flags.debug (default is 0)
verbose = see sys.flags.verbose (default is 0)
dont_write_bytecode = see sys.flags.dont_write_bytecode (default is 0)
no_user_site = see sys.flags.no_user_site (default is 0)
no_site = see sys.flags.no_site (default is 0)
xoptions = list of Python *X* options (requires Python >= 3.11)
```

The Python settings determine which Python interpreter will be used, and some Python settings. Generally speak-

ing, when your system responds to the `python` command by running the correct interpreter there is usally no need to alter this part of your configuration.

Sometimes you may want to specify options that differ from your system default; for example, when using a Python virtual environment or if the Python you want to use is not installed as your system default Python.

- **`pythonpath`**

  The pythonpath is a list of directories that Python will search in when importing modules.

  When writing your own code to be used with PyXLL you will need to change this to include the directories where that code can be imported from.

  ```
  [PYTHON]
  pythonpath =
      c:\path\to\your\code
      c:\path\to\some\more\of\your\code
      .\relative\path\relative\to\config\file
  ```

- **`executable`**

  If you want to use a different version of Python than your system default Python then setting this option will allow you to do that.

  Note that the Python version (e.g. 2.7 or 3.5) must still match whichever Python version you selected when downloading PyXLL, but this allows you to switch between different virtual environments or different Python distributions.

  PyXLL does not actually use the executable for anything, but this setting tells PyXLL where it can expect to find the other files it needs as they will be installed relative to this file (e.g. the Python dll and standard libraries).

  ```
  [PYTHON]
  executable = c:\path\to\your\python\installation\pythonw.exe
  ```

  If you wish to set the executable globally outside of the config file, the environment variable `PYXLL_PYTHON_EXECUTABLE` can be used. The value set in the config file is used in preference over this environment variable.

- **`dll`**

  PyXLL can usually locate the necessary Python dll without further help, but if your installation is non-standard or you wish to use a specific dll for any reason then you can use this setting to indform PyXLL of its location..

  ```
  [PYTHON]
  dll = c:\path\to\your\python\installation\pythonXX.dll
  ```

  If you wish to set the dll globally outside of the config file, the environment variable `PYXLL_PYTHON_DLL` can be used. The value set in the config file is used in preference over this environment variable.

- **`pythonhome`**

  The location of the standard libraries is usually determined by the location of the Python executable.

  If for any reason the standard libraries are not installed relative to the chosen or default executable then setting this option will tell PyXLL where to find them.

  Usually if this setting is set at all it should be set to whatever `sys.prefix` evaluates to in a Python prompt from the relevant interpreter.

  ```
  [PYTHON]
  pythonhome = c:\path\to\your\python\installation
  ```

  If you wish to set the pythonhome globally outside of the config file, the environment variable `PYXLL_PYTHONHOME` can be used. The value set in the config file is used in preference over this environment variable.

- **`ignore_environment`**
  New in PyXLL 3.5

  When this option is set to any value, any standard Python environment variables such as `PYTHONPATH` are ignored when initializing Python.

  This is advisable so that any global environment variables that might conflict with the settings in the pyll.cfg file do not affect how Python is initialized.

  This must be set if using FINCAD, as FINCAD sets PYTHONPATH to it's own internal Python distribution.

- **`inspect`**
  New in PyXLL 4.5

  This should typically left unset, and has a default value of 1.

  Setting to 1 is equivalent to starting Python with the "-i" switch.

  If not set, raising a `SystemExit` exception will cause the Excel process to exit.

  See `sys.flags.inspect` in the Python documentation for more information.

- **`optimize`**
  New in PyXLL 4.5

  Setting to 1 is equivalent to starting Python with the "-O" switch.

  This should typically left unset, and has a default value of 0.

  See `sys.flags.optimize` in the Python documentation for more information.

- **`debug`**
  New in PyXLL 4.5

  Setting to 1 is equivalent to starting Python with the "-d" switch.

  This should typically left unset, and has a default value of 0.

  See `sys.flags.debug` in the Python documentation for more information.

- **`verbose`**
  New in PyXLL 4.5

  Setting to 1 is equivalent to starting Python with the "-v" switch.

  This should typically left unset, and has a default value of 0.

  See `sys.flags.verbose` in the Python documentation for more information.

- **`dont_write_bytecode`**
  New in PyXLL 4.5

  Setting to 1 is equivalent to starting Python with the "-B" switch.

  This should typically left unset, and has a default value of 0.

  See `sys.flags.dont_write_bytecode` in the Python documentation for more information.

- **`no_user_site`**
  New in PyXLL 4.5

  Setting to 1 is equivalent to starting Python with the "-s" switch.

  This can be set if you do not want Python to import user installed site packages.

  See `sys.flags.no_user_site` in the Python documentation for more information.

- **`no_site`**
  New in PyXLL 4.5

  Setting to 1 is equivalent to starting Python with the "-S" switch.

  This should typically left unset, and has a default value of 0.

See `sys.flags.no_site` in the Python documentation for more information.

- **`xoptions`**

  New in PyXLL 5.11

  Requires Python >= 3.11

  List of Python *X* options, equivalent to starting Python with the "-X" switch.

3.2.2 PyXLL Settings

- *Common Settings*
- *Reload Settings*
- *Abort Settings*
- *Array Settings*
- *Object Cache Settings*
- *Plotting Settings*
- *NaN Return Settings*
- *AsyncIO Settings*
- *win32com Settings*
- *Error Handling*
- *RTD Settings*
- *CTP Settings*
- *Metadata*
- *Web Control Settings*
- *Other Settings*


```
[PYXLL]
;
modules = comma or new line delimited list of python modules
ribbon = filename (or list of filenames) of a ribbon xml documents
developer_mode = 1 or 0 indicating whether or not to use the developer mode
name = name of the addin visible in Excel
category = default category for functions registered with :py:deco:`xl_func`
docstring_styles = comma or new line delimited list of docstring styles to auto-
↪detect (rst, google, numpy)
external_config = paths or URLs of additional config files to load
optional_external_config = paths or URLs of additional config files to load
;
; reload settings
;
auto_reload = 1 or 0 to enable or disable automatic reloading (off by default)
auto_rebind = 1 or 0 to enable or disable automatic rebinding (on by default)
deep_reload = 1 or 0 to activate or deactivate the deep reload feature
deep_reload_include = modules and packages to include when reloading (only when deep_
↪reload is set)
deep_reload_exclude = modules and packages to exclude when reloading (only when deep_
↪reload is set)
deep_reload_include_site_packages = 1 or 0 to include site-packages when deep␣
```

(continues on next page)

(continued from previous page)

```
↪reloading
deep_reload_disable = 1 or 0 to disable all deep reloading functionality
;
; allow abort settings
;
allow_abort = 1 or 0 to set the default value for the allow_abort kwarg
abort_throttle_time = minimum time in seconds between checking abort status
abort_throttle_count = minimum number of calls to trace function between checking␣
↪abort status
;
; array settings
;
auto_resize_arrays = 1 or 0 to enable automatic resizing of all array functions
always_use_2d_arrays = disable 1d array types and use ``[]`` to mean a 2d array
allow_auto_resizing_with_dynamic_arrays = Resize CSE array formulas even when dynamic␣
↪arrays are available
disable_array_formula_check = Don't check whether an array formula is a CSE array␣
↪formula or not
;
; object cache settings
;
get_cached_object_id = function to get the id to use for cached objects
clear_object_cache_on_reload = clear the object cache when reloading PyXLL
recalc_cached_objects_on_open = recalculate cached object functions when opening␣
↪workbooks (default=0)
recalc_cached_objects_on_reload = recalculate cached object functions when reloading␣
↪PyXLL (default=0)
disable_loading_objects = disable loading cached objects saved in the workbook␣
↪(default=0)
;
; plotting settings
;
plot_allow_html = 1 or 0 to disable or enable html plots by default (see␣
↪:py:func:`plot`)
plot_allow_svg = 1 or 0 to disable or enable svg plots by default (see␣
↪:py:func:`plot`)
plot_allow_resize = 1 or 0 to disable or enable resizing of plot images (see␣
↪:py:func:`plot`)
plot_temp_path = path of a directory to use to save temporary images when plotting
plot_alt_text = Default alt text to use for plot images
webview2_userdata_folder = Folder to use for the web control for html plots
;
; Number conversion settings
;
nan_value = value to use if NaN is returned by a Python function
posinf_value = value to use if +Inf is returned by a Python function
neginf_value = value to use if -Inf is returned by a Python function
decimals_to_float = 1 or 0 to automatically convert returned Python ``Decimal``␣
↪objects to floating point numbers.
;
; asyncio event loop settings
;
stop_event_loop_on_reload = 1 or 0 to stop the event loop when reloading PyXLL
start_event_loop = fully qualified function name if providing your own event loop
stop_event_loop = fully qualified function name to stop the event loop
;
```

(continues on next page)

(continued from previous page)

```
; win32com settings
;
win32com_gen_path = path to use for win32com's __gen_path__ for generated wrapper␣
↪classes
win32com_delete_gen_path = 1 or 0. If set, win32com's __gen_path__ folder will be␣
↪deleted when starting
win32com_no_dynamic_dispatch = 1 or 0. If set, don't use win32com's dynamic wrappers
win32com_mutex_disable= 1 or 0. If set, don't use a global mutex to prevent␣
↪concurrent access to gen_py wrappers.
win32com_mutex_timeout = Timeout in seconds for global mutex. Use -1 for an infinite␣
↪timeout.
win32com_mutex_name = Name of the global mutex to prevent concurrent access to gen_py␣
↪wrappers.
;
; error handling
;
error_handler = function for handling uncaught exceptions
error_cache_size = maximum number of exceptions to cache for failed function calls
disable_error_cache = 1 or 0 to disable caching exceptions raised by worksheet␣
↪functions
;
; RTD settings
;
recalc_rtd_on_open = recalculate RTD functions when opening workbooks (default=1)
recalc_rtd_on_reload = recalculate RTD functions when reloading PyXLL (default=0)
rtd_volatile_default = make RTD functions volatile by default (default=0)
;
; CTP settings
;
ctp_timer_interval = time in seconds between calls to CTPBridge.on_timer (default=0.1)
;
; metadata
;
metadata_custom_xml_namespace = namespace to use instead of the default for saved␣
↪CustomXMLPart metadata
disable_saving_metadata = disable saving any metadata with the workbook
;
; other settings
;
disable_com_addin = 1 or 0 to disable the COM addin component of PyXLL
disable_com_macro_wrapper = 1 or 0 to disable calling macro functions via a COM␣
↪wrapper
disable_recalc_on_open = 1 or 0 to disable recalculating any cells on the opening of␣
↪a workbook.
disable_recalc_on_reload = 1 or 0 to disable recalculating any cells on reloading␣
↪PyXLL.
disable_function_wizard_calc = 1 or 0 to disable calculating in the function wizard.
disable_replace_calc = 1 or 0 to disable calculating in the find and replace dialog.
disable_load_library_hook = 1 or 0 to disable a fix for loading problematic DLLs.
ignore_entry_points = 1 or 0 to ignore entry points
quiet = 1 or 0 to disable all start up messages
```

Common Settings

- **`modules`**
  When PyXLL starts or is reloaded this list of modules will be imported automatically.

  Any code that is to be exposed to Excel should be added to this list, or imported from modules in this list.

  The interpreter will look for the modules usings its standard import mechanism. By adding folders using the *pythonpath* setting, which can be set in the *[PYTHON]* config section, you can cause it to look in specific folders where your software can be found.

- **`ribbon`**
  If set, the *ribbon* setting should be the file name (or list of files) of custom ribbon user interface XML file. The file names may be absolute paths or relative to the config file.

  The XML files should conform to the Microsoft CustomUI XML schema (*customUI.xsd*) which may be downloaded from Microsoft here https://www.microsoft.com/en-gb/download/details.aspx?id=1574.

  If a list of files is given then all of those files will be loaded. Any tabs or groups with the same ids found in the files will be merged.

  See the *Customizing the Ribbon* chapter for more details.

- **`developer_mode`**
  When the developer mode is active a PyXLL menu with a *Reload* menu item will be added to the Addins toolbar in Excel.

  If the developer mode is inactive then no menu items will be automatically created so the only ones visible will be the ones declared in the imported user modules.

  This setting defaults to off (0) if not set.

- **`name`**
  The *name* setting, if set, changes the name of the addin as it appears in Excel.

  When using this setting the addin in Excel is indistinguishable from any other addin, and there is no reference to the fact it was written using PyXLL. If there are any menu items in the default menu, that menu will take the name of the addin instead of the default 'PyXLL'.

- **`category`**
  The *category* setting changes the default category used when registering worksheet functions with `@xl_func`.

- **`docstring_styles`**
  New in PyXLL 5.13

  By default PyXLL auto-detects function docstrings written in reST/Sphinx, Google or NumPy style (see *Function Documentation*) and uses whichever style it finds first in the docstring.

  This setting can be used to restrict which of those styles are considered, in case detection ever misfires for one of your functions. It should be a comma or newline separated list containing any of `rst`, `google` and `numpy`, e.g.:

  ```
  [PYXLL]
  ; Only detect reST/Sphinx style docstrings
  docstring_styles = rst
  ```

- **`external_config`**
  This setting may be used to reference another config file (or files) located elsewhere, either as a relative or absolute path or as a URL.

  For example, if you want to have the main pyxll.cfg installed on users' local PCs but want to control the configuration via a shared file on the network you can use this to reference that external config file.

  Multiple external config files can be used by setting this value to a list of file names (comma or newline separated) or file patterns.

Values in external config files override what's in the parent config file, apart from *pythonpath*, *modules* and *external_config* which get appended to.

In addition to setting this in the config file, the environment variable *PYXLL_EXTERNAL_CONFIG_FILE* can be used. Any external configs set by this environment variable will be added to those specified in the config.

- **`optional_external_config`**
  This setting is identical to `external_config` except that if a file does not exist or cannot be read then a warning will be logged rather than an error.

  This can be useful if specifying a user config in a standard location and your users may or may not have that file.

Reload Settings

- **`auto_reload`**
  When set PyXLL will detect when any Python modules, config or ribbon files have been modified and automatically trigger a reload.

  This setting defaults to off (0) if not set.

- **`auto_rebind`**
  If any of the decorators `@xl_func`, `@xl_macro` or `@xl_menu` are called after PyXLL has started PyXLL can automatically re-create the function bindings in Excel. This is useful if dynamically importing modules after PyXLL has started.

  This setting defaults to on (1) if not set.

- **`deep_reload`**
  Reloading PyXLL reloads all the modules listed in the *modules* config setting. When working on more complex projects often you need to make changes not just to those modules, but also to modules imported by those modules.

  PyXLL keeps track of anything imported by the modules listed in the *modules* config setting (both imported directly and indirectly) and when the *deep_reload* feature is enabled it will automatically reload the module dependencies prior to reloading the main modules.

  Standard Python modules and any packages containing C extensions are never reloaded.

  It should be set to 1 to enable deep reloading 0 (the default) to disable it.

- **`deep_reload_include`**
  Optional list of modules or packages to restrict reloading to when deep reloading is enabled.

  If not set, everything excluding the standard Python library and packages with C extensions will be considered for reloading.

  This can be useful when working with code in only a few packages, and you don't want to reload everything each time you reload. For example, you might have a package like:

  ```
  my_package \
      - __init__.py
      - business_logic.py
      - data_objects.py
      - pyxll_functions.py
  ```

  In your config you would add *my_package.pyxll_function* to the modules to import, but when reloading you would like to reload everything in *my_package* but not any other modules or packages that it might also import (either directly or indirectly). By adding *my_package* to *deep_reload_include* the deep reloading is restricted to only reload modules in that package (in this case, *my_package.business_logic* and *my_package.data_objects*).

```
[PYXLL]
modules = my_package
deep_reload = 1
deep_reload_include = my_package
```

- **`deep_reload_exclude`**
  Optional list of modules or packages to exclude from deep reloading when *deep_reload* is set.

  If not set, only modules in the standard Python library and modules with C extensions will be ignored when doing a deep reload.

  Reloading Python modules and packages doesn't work for all modules. For example, if a module modifies the global state in another module when its imported, or if it contains a circular dependency, then it can be problematic trying to reload it.

  Because the deep_reload feature will attempt to reload all modules that have been imported, if you have a module that cannot be reloaded and is causing problems you can add it to this list to be ignored.

  Excluding a package (or sub-package) has the effect of also excluding anything within that package or sub-package. For example, if there are modules `a.b.x` and `a.b.y` then excluding `a.b` will also exclude `a.b.x` and `a.b.y`.

  deep_reload_exclude can be set when *deep_reload_include* is set to restrict the set of modules that will be reloaded. For example, if there are modules 'a.b and 'a.b.c', and everything in 'a' should be reloaded except for 'a.b.c' then 'a' would be added to *deep_reload_include* and 'a.b.c' would be added to *deep_reload_exclude*.

- **`deep_reload_include_site_packages`**
  When *deep_reload* is set, any modules inside the *site-packages* folder will be ignored unless this option is enabled.

  This setting defaults to off (0) if not set.

- **`deep_reload_disable`**
  Deep reloading works by installing an import hook that tracks the dependencies between imported modules. Even when *deep_reload* is turned off this import hook is enabled, as it is sometimes convenient to be able to turn it on to do a deep reload without restarting Excel.

  When *deep_reload_disable* is set to 1 then this import hook is not enabled and setting *deep_reload* will have no effect. .. warning:: *Changing this setting requires Excel to be restarted.*

Abort Settings

- **`allow_abort` (defaults to 0)**
  The *allow_abort* setting is optional and sets the default value for the *allow_abort* keyword argument to the decorators `@xl_func`, `@xl_macro` and `@xl_menu`.

  It should be set to 1 for True or 0 for False. If unset the default is 0.

  Using this feature enables a Python trace function which will impact the performance of Python code while running a UDF. The exact performance impact will depend on what code is being run.

- **`abort_throttle_time`**
  When a UDF has been registered as abort-able, a trace function is used that gets called frequently as the Python code is run by the Python interpreter.

  To reduce the impact of the trace function Excel can be queried less often to see if the user has aborted the function.

  abort_throttle_time is the minimum time in seconds between checking Excel for the abort status.

- **`abort_throttle_count`**
  When a UDF has been registered as abort-able, a trace function is used that gets called frequently as the Python code is run by the Python interpreter.

To reduce the impact of the trace function Excel can be queried less often to see if the user has aborted the function.

abort_throttle_count is the minimum number of call to the trace function between checking Excel for the abort status.

Array Settings

- **`auto_resize_arrays` (defaults to 0)**
  The *auto_resize_arrays* setting can be used to enable automatic resizing of array formulas for all array function. It is equivalent to the *auto_resize* keyword argument to `@xl_func` and applies to all array functions that don't explicitly set *auto_resize*.

  It should be set to 1 for True or 0 for False (the default).

- **`always_use_2d_arrays` (defaults to 0)**
  Before PyXLL 4.0, all array arguments and return types were 2d arrays (list of lists). The type suffix `[]` was used to mean a 2d array type (e.g. a `float[]` argument would receive a list of lists).

  Since PyXLL 4.0, 1d arrays have been added and `[][]` should now be used when a 2d array is required. To make upgrading easier, this setting disables 1d arrays and any array types specified with `[]` will be 2d arrays as they were prior to version 4.

- **`allow_auto_resizing_with_dynamic_arrays` (defaults to 1)**
  In 2019 Excel added a new "Dynamic Arrays" feature to Excel. This replaces the need for auto resized arrays in PyXLL.

  It is still possible to enter old-style Ctrl+Shift+Enter (CSE) arrays however, and these will continue to be resized automatically by PyXLL if `auto_resize` is set for the function.

  PyXLL's auto-resizing can be disabled completely if Excel has the new dynamic arrays feature by setting this option to 0.

  New in PyXLL 4.4.

- **`disable_array_formula_check` (defaults to 0)**
  PyXLL checks the formula of array functions to determine whether the function is an old style Ctrl+Shift+Enter (CSE) formula or a new style dynamic array.

  It uses this to determine whether or not to use its own auto-resizing for the the array function.

  This check can be disabled by setting this to 1.

  New in PyXLL 4.4.

Object Cache Settings

- **`get_cached_object_id`**
  When Python objects are returned from an Excel worksheet function and no suitable converter is found (or the return type `object` is specified) the object is added to an internal object cache and a handle to that cached object is returned.

  The format of the cached object handle can be customized by setting *get_cached_object_id* to a custom function, e,g

```
[PYXLL]
get_cached_object_id = module_name.get_custom_object_id
```

```
def get_custom_object_id(obj):
    return "[Cached %s <0x%x>]" % (type(obj), id(obj))
```

  The computed id must be unique as it's used when passing these objects to other functions, which retrieves them from the cache by the id.

- **clear_object_cache_on_reload**
  Clear the object cache when reloading the PyXLL add-in.

  Defaults to 1, but if using cached objects that are instances of classes that aren't reloaded then this can be set to 0 to avoid having to recreate them when reloading.

- **recalc_cached_objects_on_open**
  If set, default all functions that return cached objects as needing to be recalculated when opening a workbook.

  This is the equivalent to setting `recalc_on_open=True` in the *@xl_func* decorator. Disabling it does not prevent cells that have already been saved with this flag set from be calculated when a workbook opens. For that, set `disable_recalc_on_open=1` in your config.

  This setting can be overridden on specific functions by setting `recalc_on_open` in the *@xl_func* decorator.

  Defaults to 0.

  See *Recalculating On Open*.

- **recalc_cached_objects_on_reload**
  If set, default all functions that return cached objects as needing to be recalculated when reloading the PyXLL add-in.

  This is the equivalent to setting `recalc_on_reload=True` in the *@xl_func* decorator. Disabling it does not prevent cells that have already been saved with this flag set from being calculated when reloading. For that, set `disable_recalc_on_reload=1` in your config.

  This setting can be overridden on specific functions by setting `recalc_on_reload` in the *@xl_func* decorator.

  Defaults to 0.

  See *Recalculating On Reload*.

- **disable_loading_objects**
  If set, any cached objects saved as part of a workbook will be ignored when opening the workbook.

  Defaults to 0.

  See *Saving Objects in the Workbook*.

Plotting Settings

- **plot_allow_html**
  New in Python 5.9

  For plotting libraries that support html plots, PyXLL will try to use an interactive web control to display the plot.

  Setting `plot_allow_html = 0` changes the default behaviour of the *plot* function so that if `allow_html` is not specified, by default html plots will be disabled and a static image format will be used instead (if available).

- **plot_allow_svg**
  For plotting libraries that support exporting SVG files, PyXLL will use that instead of a bitmap format.

  Setting `plot_allow_svg = 0` changes the default behaviour of the *plot* function so that if `allow_svg` is not specified, by default SVG plots will be disabled and a bitmap format will be used instead (if available).

- **plot_allow_resize**
  For plots displayed in Excel as static images, when resizing the image PyXLL can re-plot the figure and update the image to the new size automatically.

Setting `plot_allow_resize = 0` changes the default behaviour of the *plot* function so that if `allow_resize` is not specified, by default if the image in Excel is resized the figure will not be re-plotted to match the new size.

- **`plot_temp_path`**
  When exporting plots as images they are exported to a temporary file. This option can be used to change where the temporary files will be saved.
- **`plot_alt_text`**
  Sets the default alternative text (description) used for the picture object created in Excel by the *plot* function.

  Setting `plot_alt_text` changes the default behaviour of the *plot* function so that if `alt_text` is not specified, this value will be used instead.
- **`webview2_userdata_folder`**
  New in Python 5.9

  For HTML plots displayed using a web control, the web control needs a folder to store its user data. The same folder is also used for temporarily writing the exported html for the plots.

  This setting can be used to change where the web control should store its data.

  By default, a folder named `.webview2_userdata` will be used in the same location as the PyXLL add-in.

NaN Return Settings

New in PyXLL 5.5.

Values to use when `NaN`, `+Inf` and `-Inf` are returned from Python to Excel.

These can be overridden for individual functions when using *@xl_func*, *@xl_macro* or *XLCell.options*, but if not specified the values set in the config file will be used instead.

Valid values for these settings are:

`#NULL!`
`#DIV/0!`
`#VALUE!`
`#REF!`
`#NAME!`
`#NUM!`
`#N/A`
`INF`
`None`

Or any numeric or string value can also be used.

- **`nan_value`**
  Value to use if `NaN` is returned from Python to Excel. Defaults to `#NUM!`
- **`posinf_value`**
  Value to use if `+Inf` is returned from Python to Excel. Defaults to `+INF`.
- **`neginf_value`**
  Value to use if `-Inf` is returned from Python to Excel. Defaults to `-INF`.
- **`none_value`**
  Value to use if `None` is returned from Python to Excel. Defaults to `None`.

AsyncIO Settings

- **stop_event_loop_on_reload**
  If set to '1', the asyncio Event Loop used for async user defined functions and RTD methods will be stopped when PyXLL is reloaded.

  See *Asynchronous Functions*.

  New in PyXLL 4.2.0.

- **start_event_loop**
  Used to provide an alternative implementation of the asyncio event loop used by PyXLL.

  May be set to the fully qualified name of a function that takes no arguments and returns a started asyncio.AbstractEventLoop.

  If this option is set then *stop_event_loop* should also be set.

  See *Asynchronous Functions*.

  New in PyXLL 4.2.0.

- **stop_event_loop**
  Used to provide an alternative implementation of the asyncio event loop used by PyXLL.

  May be set to the fully qualified name of a function that stops the event loop started by the function specified by the option *start_event_loop*.

  If this option is set then *start_event_loop* should also be set.

  See *Asynchronous Functions*.

  New in PyXLL 4.2.0.

win32com Settings

- **win32com_gen_path**
  This sets the `win32com.__gen_path__` path used for win32com's generated wrapper classes.

  By default win32com uses the user's Temp folder, but this is shared between all Python sessions, not just PyXLL. If this becomes corrupted or updated by an external Python script then it can stop the win32com package from functioning correctly, and setting it to a folder specifically for PyXLL can avoid that problem.

- **win32com_delete_gen_path**
  If set the `win32com.__gen_path__` folder used for generated wrapper classes will be deleted when PyXLL starts.

  This is not usually necessary as setting `win32com_gen_path` will ensure that no other Python code will use the same generated wrapper classes, however it can be set if you are experiencing problems with the wrapper classes becoming corrupted or invalid.

  If using this option you will also want to set `win32com_gen_path` so the wrapper classes are created somewhere other than the default location. The folder referenced by `win32com_gen_path` is the one that will be deleted.

  Care should be taken to ensure that there is nothing in the folder you do not want to be deleted before setting this option, although the folder can be recovered from the recycle bin.

- **win32com_no_dynamic_dispatch**
  When returning a COM object using the win32com package, PyXLL will attempt to use a static wrapper generated by win32com. If that fails and this setting is not set then it will fallback to using a dynamic dispatch wrapper.

  Dynamic wrappers are suitable in most cases and behave in the same way as the static wrappers, but the win32com.client.constants set of constants only contains constants included by static wrappers, and so falling back to dynamic dispatch can result in missing constants.

- **win32com_mutex_disable**
  PyXLL uses a global mutex to prevent multiple Excel sessions from attempting to create the win32com wrapper modules at the same time when calling `xl_app`.

  This is to prevent multiple Excel sessions from creating the wrappers at the same time and overwriting each other, leading to corrupt wrapper files.

  This can be disabled by setting this setting to 1 but if you think you need to disable this then please contact PyXLL support before doing so.

  New in PyXLL 5.1.

- **win32com_mutex_timeout**
  This setting is only provided as a precaution and you should contact PyXLL support if you think you need to change it.

  New in PyXLL 5.1.

- **win32com_mutex_name**
  This setting is only provided as a precaution and you should contact PyXLL support if you think you need to change it.

  New in PyXLL 5.1.

Error Handling

- **error_handler**
  If a function raises an uncaught exception, the error handler specified here will be called and the result of the error handler is returned to Excel.

  If not set, uncaught exceptions are returned to Excel as error codes.

  See *Error Handling*.

- **error_cache_size**
  If a worksheet function raises an uncaught exception it is cached for retrieval via the `get_last_error` function.

  This setting sets the maximum number of exceptions that will be cached. The least recently raised exceptions are removed from the cache when the number of cached exceptions exceeds this limit.

  The default is 500.

- **disable_error_cache**
  New in PyXLL 5.13

  Set to `1` to disable the *error cache* entirely, so that exceptions raised by worksheet functions are not cached and `get_last_error` will not return anything for them.

  Off (`0`) by default.

RTD Settings

- **recalc_rtd_on_open**
  Default all RTD functions as needing to be recalculated when opening a workbook.

  This is the equivalent to setting `recalc_on_open=True` in the `@xl_func` decorator. Disabling it does not prevent cells that have already been saved with this flag set from be calculated when a workbook opens. For that, set `disable_recalc_on_open=1` in your config.

  This setting can be overridden on specific functions by setting `recalc_on_open` in the `@xl_func` decorator.

  Defaults to 1.

  See *Recalculating On Open*.

- **recalc_rtd_on_reload**
  Default all RTD functions as needing to be recalculated when reloading the PyXLL add-in.

  This is the equivalent to setting `recalc_on_reload=True` in the *@xl_func* decorator. Disabling it does not prevent cells that have already been saved with this flag set from being calculated when reloading. For that, set `disable_recalc_on_reload=1` in your config.

  This setting can be overridden on specific functions by setting `recalc_on_reload` in the *@xl_func* decorator.

  Defaults to 0.

  See *Recalculating On Reload*.

- **rtd_volatile_default**
  Make all RTD functions volatile by default. This restores the behaviour prior to PyXLL 4.5.0.

  When enabled RTD functions are volatile so they will be calculated when opening a workbook, but the wrapped Python function will only be called if the arguments to the function are actually changed.

  Usually this should be left disabled as RTD functions are now calculated when the workbook opens using the *Recalculating On Open* feature of PyXLL instead.

  Defaults to 0.

CTP Settings

- **ctp_timer_interval**
  Time in seconds between calls to *CTPBridgeBase.on_timer*.

  The CTP bridge classes are what integrate the Python UI toolkit with the Excel Windows message loop. They use on_timer to poll their own message queues. If you are finding the panel is not responsive enough you can reduce the timer interval with this setting.

  This can also be set for each CTP by passing `timer_interval` to :py:func`create_ctp`.

  New in PyXLL 5.1

Metadata

- **metadata_custom_xml_namespace**
  Custom metadata is saved in order to support certain features of PyXLL such as recalculating cells when a workbook opens.

  This is saved in the workbook as a CustomXMLPart using an XML namespace specific to the PyXLL add-in so as not to conflict with data saved by other add-ins. If you have specified a name for your add-in using the `name` setting that will be used to avoid conflict with any other PyXLL add-ins you may have loaded.

  If you prefer to specify the namespace to use instead of having PyXLL use it's own namespace you can do so by setting this option.

  ```
  [PYXLL]
  metadata_custom_xml_namespace = urn:your_name:metadata
  ```

- **disable_saving_metadata**
  Set this option to disable writing any metadata.

  Note that this will affect all PyXLL features that require metadata such as *recalculating on open*, as well as *formatting dynamic arrays*.

  The default is 0 (not disabled).

Web Control Settings

New in PyXLL 5.5

PyXLL can make web requests to fetch files from web servers instead of using a file path. For example, the license *file*, `external_config` and `startup_script` settings can all use a URL instead of a path to download a file. PyXLL will also attempt to ping a license server and download the latest license information, if possible.

The web control used to make these requests can be configured using the options in this section.

Usually these options should not be set as PyXLL will detect the correct settings automatically.

- **`webclient_disable_autoproxy`**
  PyXLL will attempt to detect your proxy settings automatically if you have AutoProxy configured.

  This can be disabled by setting this option to 1.

- **`webclient_autoproxy_logon_if_challenged`**
  If your proxy server requires authentication PyXLL will attempt to auto-logon if challenged.

  This can be disabled by setting this option to 0.

- **`webclient_proxy`**
  If you connect to the internet through a manually configured proxy server, and PyXLL is not detecting that proxy server automatically, this can be used to set the proxy server.

- **`webclient_proxy_bypass`**
  If you connect to the internet through a proxy, and you are manually specifying the proxy server using the `webclient_proxy` setting above, you can use this setting to configure some sites to bypass the proxy server.

  The proxy server will not be used for addresses beginning with entries in this list. Use semicolons (;) to separate entries.

- **`webclient_disable_autologon`**
  If connecting to an Intranet site or trusted URL that supports auto-logon PyXLL can try to authenticate automatically.

  By default, auto-logon is only used for Intranet requests or Trusted Sites. This can be overridden using the `webclient_autologon_security_level` setting.

  Auto-logon can be disabled by setting this option to 1.

- **`webclient_autologon_security_level`**
  This can be set to *low*, *medium* or *high* and determines what type of sites can be sent credentials for authentication.

  - *low*: Default credentials may be sent to all servers.
  - *medium*: Default credentials may be sent for Intranet requests.
  - *high*: Default credentials cannot be used for authentication.

  By default, auto-logon is used for Intranet requests and Trusted Sites and it is recommended in most cases that this setting is not set.

- **`webclient_retry_count`**
  New in PyXLL 5.11

  Retry a number of times if a request fails.

- **`webclient_retry_delay`**
  New in PyXLL 5.11

  Delay in milliseconds between each retry after a request has failed.

Other Settings

- **`startup_script`**
  Path or URL of a batch or Powershell script to run when Excel starts.

  This script will be run when Excel starts, but before Python is initialized. This is so that the script can install anything required by the add-in on demand when Excel runs.

  See *Startup Script*.

- **`disable_com_addin`**
  PyXLL is packaged as a single Excel addin (the pyxll.xll file), but it actually implements both a standard XLL addin and COM addin in the same file.

  Setting *disable_com_addin* to 1 stops the COM addin from being used.

  The COM addin is used for ribbon customizations and RTD functions and if disabled these features will not be available.

- **`disable_com_macro_wrapper`**
  Macro functions (see `@xl_macro`) are called via a COM wrapper, which allows both COM and Excel commands (such as `xlcAlert`) to be used safely from within the same macro.

  Setting this to 1 disables that COM wrapper, reverting to calling macro functions directly.

  New in PyXLL 5.13

- **`disable_recalc_on_open`**
  Disable any automatic recalculations when a workbook is opened that would otherwise be caused by the *Recalculating On Open* feature.

  This does not stop Excel from calculating anything else, such as volatile functions or other dirty cells in the saved workbook.

  See *Recalculating On Open*.

- **`disable_recalc_on_reload`**
  Disable any automatic recalculations when the PyXLL add-in is reloaded that would otherwise be caused by the *Recalculating On Reload* feature.

  This does not stop Excel from calculating anything else, such as volatile functions or other dirty cells.

  See *Recalculating On Reload*.

- **`disable_function_wizard_calc`**
  If set, the default behaviour of functions registered with `@xl_func` is for them to not calculate when the Excel function wizard is active.

  This can be overriden using the `disable_function_wizard_calc` argument to `@xl_func`.

  New in PyXLL 5.8.0

- **`disable_replace_calc`**
  If set, the default behaviour of functions registered with `@xl_func` is for them to not calculate when the Excel find and replace dialog is active.

  This can be overriden using the `disable_replace_calc` argument to `@xl_func`.

  New in PyXLL 5.8.0

- **`disable_load_library_hook`**
  There can be issues loading certain DLLs, such as some OpenSSL DLLs used by the packages *ssl*, *cryptography*, and others with some Python distributions (Anaconda).

  PyXLL attempts to work around these issues by modifying how Python loads DLLs. This option disables that work around.

  New in PyXLL 5.10.0

- **ignore_entry_points**
  If your Python packages are on a network drive it can be slow to look for entry points, which may result in slow start times for Excel.

  This setting stops PyXLL from looking for entry points.

  See *Setuptools Entry Points*.

- **decimals_to_float**
  If 1 (default) then any Python `Decimal` objects will automatically be converted to a floating point number before being returned to Excel.

  Excel has no support for decimals and so if this is disabled any returned `Decimal` objects will be returned as a Python object handle.

  New in PyXLL 5.11.0

- **quiet**
  The *quiet* setting is for use in enterprise settings where the end user has no knowledge that the functions they're provided with are via a PyXLL addin.

  When set PyXLL won't raise any message boxes when starting up, even if errors occur and the addin can't load correctly. Instead, all errors are written to the log file.

3.2.3 License Key

```
[LICENSE]
key = license key
file = path to shared license key file
```

If you have a PyXLL license key you should set it in `[LICENSE]` section of the config file.

The license key may be embedded in the config as a plain text string, or it can be referenced as an external file containing the license key. This can be useful for group licenses so that the license key can be managed centrally without having to update each user's configuration when it is renewed.

- **key**
  Plain text license key as provided when you purchased PyXLL.

  This does not need to be set if you are setting *file*.

  The environment variable *PYXLL_LICENSE_KEY* can be used instead of setting this in the config file.

- **file**
  Path or URL of a plain text file containing the license key as provided when you purchased PyXLL.

  The file may contain comment lines starting with *#*.

  This does not need to be set if you are setting *key*.

  The environment variable *PYXLL_LICENSE_FILE* can be used instead of setting this in the config file.

3.2.4 Logging

PyXLL redirects all stdout and stderr to a log file. All logging is done using the standard logging python module.

The `[LOG]` section of the config file determines where logging information is redirected to, and the verbosity of the information logged.

The *Config Variables* are useful when configuring the log file as they allow including the current date, process id, and other variables in the log file name and/or path.

```
[LOG]
path = directory of where to write the log file
file = filename of the log file
verbosity = logging level (debug, info, warning, error or critical)
```

(continues on next page)

(continued from previous page)

```
format = log format template
debug_format = optional format template (defaults to format)
info_format = optional format template (defaults to same as debug_format)
warning_format = optional format template (defaults to same as info_format)
error_format = optional format template (defaults to same as warning_format)
critical_format = optional format template (defaults to same as error_format)
max_size = maximum size the log file can get to before rolling to a new file.
roll_interval = period before the log file will be rolled and a new log will be␣
↪started.
backup_count = number of old log files to keep.
encoding = encoding to use when writing the logfile (defaults to 'utf-8')
redirect_stdout = 1 or 0 to enable or disable redirection of native (C) stdout to log␣
↪file.
redirect_stderr = 1 or 0 to enable or disable redirection of native (C) stderr to log␣
↪file.
```

- **path**

  Path where the log file will be written to.

  This may include substitution variables as in the section *Config Variables*, e.g.

  ```
  [LOG]
  path = C:/Temp/pyxll-logs-%(date)s
  ```

- **file**

  Filename of the log file.

  This may include substitution variables as in the section *Config Variables*, e.g.

  ```
  [LOG]
  file = pyxll-log-%(pid)s-%(xlversion)s-%(date)s.log
  ```

- **verbosity**

  The logging verbosity can be used to filter out or show warning and errors. It sets the log level for the root logger in the `logging` module, as well as setting PyXLL's internal log level.

  It may be set to any of the following

  - debug (most verbose level, show all log messages including debugging messages)
  - info
  - warning
  - error
  - critical (least verbose level, only show the most critical errors)

  If you are having any problems with PyXLL it's recommended to set the log verbosity to *debug* as that will give a lot more information about what PyXLL is doing.

- **format**

  The format string is used by the logging module to format any log messages. An example format string is:

  ```
  [LOG]
  format = "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
  ```

  In addition to the standard `LogRecord` attrbitues provided by the `logging` module, the following may also be used:

Attribute Name	Format	Description	PyXLL Version
address	`%(address)s`	Full address of the calling cell, if known.	>= 5.10.0
sheet	`%(sheet)s`	Sheet name of the calling cell, if known.	>= 5.10.0
cell	`%(cell)s`	Cell reference of the calling cell, if known.	>= 5.10.0

For more information about log formatting, please see the `logging` module documentation from the Python standard library.

- **`debug_format`**[3]
  Log format to use for log messages of level `DEBUG`.

  Defaults to the `format` option if not set (default).

- **`info_format`**[3]
  Log format to use for log messages of level `INFO`.

  Defaults to the same as `debug_format` if not set (default).

- **`warning_format`**[3]
  Log format to use for log messages of level `WARNING`.

  Defaults to the same as `info_format` if not set (default).

- **`error_format`**[3]
  Log format to use for log messages of level `ERROR`.

  Defaults to the same as `warning_format` if not set (default).

- **`critical_format`**[3]
  Log format to use for log messages of level `CRITICAL`.

  Defaults to the same as `error_format` if not set (default).

- **`max_size`**[1]
  Maximum size the log file is allowed to grow to.

  Once the log file goes over this size it will be renamed to add a timestamp to the file and a new log file will be started.

  The size can be in Kb, Mb or Gb, for example to set it to 100Mb use `max_size = 100Mb`.

  If zero, the log file will be allowed to grow indefinitely.

- **`roll_interval`**[1]
  If set the log file will be rolled periodically.

  This setting can be used alongside `max_size` and if both are set the log will be rolled either either the roll period is reached or the file size goes over the maximum allowed size.

  The interval can be any of:

  - a number of days, hours, minutes or seconds using the form `Nd` for days (eg `7d`), `Nm`, and `Ns` respectively.
  - `midnight` to indicate the log should be rolled after midnight.
  - `W0-6` to roll on a specific day of the week, eg `W0` for Sunday and `W6` for Saturday.

- **`roll_backoff_interval`**[2]
  If rolling the log file fails a retry won't be attempted for a short period of time. The default time between retries is 5 minutes.

  The interval can be number of days, hours, minutes or seconds using the form `Nd` for days (eg `7d`), `Nm`, and `Ns` respectively.

[3] New in PyXLL 5.10
[1] Log rolling is new in PyXLL 5.2.
[2] New in PyXLL 5.6.

- **backup_count**[Page 36, 1]
  The number of backup log files to keep after rolling the log.

  If set, only the last *N* rolled log files will be kept.

  Instead of setting a fixed number a period can be specified, eg `7d` to keep log files for 7 days.
- **encoding**
  Encoding to use when writing the log file.

  Defaults to 'utf-8'.

  New in PyXLL 4.2.0.
- **redirect_stdout**
  Enables or disabled redirecting the native (C runtime) stdout to the PyXLL log file.

  Some Python extensions, and the Python interpreter itself, sometimes log to the native C stdout instead of to the redirected Python stdout or to a log file.

  When this option is set the native stdout is redirected to the PyXLL log file.

  Defaults to 1.

  New in PyXLL 5.11.0.
- **redirect_stderr**
  Enables or disabled redirecting the native (C runtime) stderr to the PyXLL log file.

  Some Python extensions, and the Python interpreter itself, sometimes log to the native C stderr instead of to the redirected Python stderr or to a log file.

  When this option is set the native stderr is redirected to the PyXLL log file.

  Defaults to 1.

  New in PyXLL 5.11.0.

3.2.5 Warnings

The Python `warnings` package is part of the Python standard library and used to alert the user of a problem.

Typically these warnings are logged to the log file, but the warnings module can be configured to elevate these warnings to exceptions or to ignore them completely.

For full details of the `warnings` package please see the Python documentation here https://docs.python.org/3/library/warnings.html.

The `warnings` package can be configured in the `[LOG]` section of the PyXLL config file with the following options.

- **capture_warnings**
  Write warnings to the log file.

  If this is set to 0 then warnings will not be captured and written to the log file.

  Defaults to 1.
- **warnings_filters**
  Warnings filters to control whether warnings are ignored, dispayed, or turned into errors.

  Multiple warning filters can be configured on multiple lines.

  Individual warnings filters are specified as a sequence of fields separated by colons:

  ```
  action:message:category:module:line
  ```

  Example:

```
[LOG]
warnings_filters =
    default::DeprecationWarning:__main__
    ignore::DeprecationWarning
    ignore::PendingDeprecationWarning
    ignore::ImportWarning
    ignore::ResourceWarning
```

3.2.6 Config Variables

PyXLL creates some configuration substitution variables that can be used in any config values.

Variable substitution in the config file follows the same format as Python's `configparser` module, which is `%(name)s`, where `name` is the variable name.

For example, including today's date in the log file name would be specified as:

```
[LOG]
file = pyxll-log-%(date)s.txt
```

The following sustitution variables are available:

Variable Name	Format	Description	PyXLL Version
pid	`%(pid)s`	Process id	All
date	`%(date)s`	Current date in YYYYMMDD format	All
time	`%(time)s`	Current time in HHMMSS format	>= 5.6
xl_version[1]	`%(xl_version)s`	Excel version	All
py_version	`%(py_version)s`	Python version	>= 5.2
pyxll_version	`%(pyxll_version)s`	PyXLL version	>= 5.2
xll_path	`%(xll_path)s`	Full path of the pyxll.xll add-in	>= 5.6
xll_dir	`%(xll_dir)s`	Directory containing the pyxll.xll add-in	>= 5.6
cfg_path	`%(cfg_path)s`	Full path of the config file being processed	>= 5.6
cfg_dir	`%(cfg_dir)s`	Directory containing the config file being processed	>= 5.6
basecfg_path[2]	`%(basecfg_path)s`	Full path of the base config file	>= 5.6
basecfg_dir[2]	`%(basecfg_dir)s`	Directory containing the base config file	>= 5.6

In addition to these standard substition variables, environment variables can also be used as substitution variables in the config.

See *Environment Variables* for more details about using environment variables in the config file.

See also *Config Functions*.

3.2.7 Config Functions

New in PyXLL 5.11

A few pre-defined functions can be used within the config to simplify some scenarios.

Function calls are made using the same syntax as variable substition, but with arguments added.

A function call is specified using the form `%(function arg1, arg2, ...)s`. Note the trailing `s` which is the same as for substituted variables.

The available functions are:

[1] `xlversion` was renamed to `xl_version` in PyXLL 5.2 but both forms will continue to work.

[2] The base config is the first config file loaded by the add-in. This can be different from the config file being processed if the base config specifies additional config files using the `external_config` option.

- *abspath*
- *dirname*
- *basename*
- *filename*

Functions can be nested, and substitution and environment variables can be used as arguments. For example:

```
[SECTION]
; Get the directory of the absulte path of some file.
; The environment variable 'USER_OVERRIDE_PATH' can be set, otherwise ./default.txt is␣
↪used.
abs_dir = %(dirname %(abspath %(USER_OVERRIDE_PATH:./default.txt)s)s)s
```

abspath

Converts a relative path (relative to the config file being parsed) to an absolute path.

Example:

```
[SECTION]
option = %(abspath ./file.txt)s
```

dirname

Gets the directory component of a path.

Example:

```
[SECTION]
; option will be set to `C:\path\to`
option = %(dirname C:\path\to\file.txt)s
```

basename

Gets the base name of a path. The base name is the full file name, including any extension.

Combine with `filename` if the file extension is not required.

Example:

```
[SECTION]
; option will be set to 'file.txt'
option = %(basename C:\path\to\file.txt)s
```

filename

Strips any file extension from a path and returns only the filename component (including any directory).

Combine with `basename` if only the file name without any path component is required.

Example:

```
[SECTION]
; option will be set to 'file'
option = %(filename %(basename C:\path\to\file.txt)s)s
```

3.2.8 Environment Variables

Config values may include references to environment variables. To substitute an environment variable into your value use

`%(ENVVAR_NAME)s`

When the variable has not been set, (since PyXLL 4.1) you can set a default value using the following format

`%(ENVVAR_NAME:default_value)s`

For example:

```
[LOG]
path = %(TEMP:./logs)s
file = %(LOG_FILE:pyxll.log)s
```

It's possible to set environment variables in the `[ENVIRONMENT]` section of the config file.

```
[ENVIRONMENT]
NAME = VALUE
```

For each environment variable you would like set, add a line to the `[ENVIRONMENT]` section.

3.2.9 Startup Script

New in PyXLL 4.4.0

- *Introduction*
- *Example*
- *Script Commands*

Introduction

The `startup_script` option can be used to run a batch or Powershell script when Excel starts, and again each time PyXLL is reloaded.

This can be useful for ensuring the Python environment is installed correctly and any Python packages are up to date, or for any other tasks you need to perform when starting Excel.

The script runs before Python is initialized, and can therefore be used to set up a Python environment if one doesn't already exist. The PyXLL config can be manipulated from the startup script so any settings such as the `modules` list, `pythonpath` or even the Python `executable` can be set on startup rather than being fixed in the pyxll.cfg file.

The startup script can be a local file, a file on a network drive, or even a URL. Using a network drive or a URL can be a good option when deploying PyXLL to multiple users where you want to have control over what's run on startup without having to update each PC.

Batch files (.bat or .cmd) and Powershell files (.ps1) are supported. Script files must use one of these file extensions.

The script is run with the current working directory (CWD) set to the same folder as the PyXLL add-in itself, and so relative paths can be used relative to the xll file.

If successful the script should exit with exit code 0. Any other exit code will be interpreted as the script not having been run successfully by PyXLL.

See also *Using a startup script to install and update Python code*.

Example

A startup script could be used to download a Python environment and configure PyXLL.

```
REM startup-script.bat
@ECHO OFF

REM If the Python env already exists no need to download it
IF EXIST ./python-env-xx GOTO SKIPDOWNLOAD

REM Download and unpack a Python environment to ./python-env-xx/
wget https://intranet/python/python-env-xx.tar.gz
tar -xzf python-env-xx.tar.gz --directory python-env-xx
:SKIPDOWNLOAD

REM Update the PyXLL settings with the executable
ECHO pyxll-set-option PYTHON executable ./python-venv-xx/pythonw.exe
```

The script is configured in the pyxll.cfg file, and could be on a remote network drive or web server.

```
[PYXLL]
startup_script = https://intranet/pyxll/startup-script.bat
```

Script Commands

When PyXLL runs the startup script (either a batch or Powershell script) it monitors the stdout of the script for special commands. These commands can be used by your script to get information from PyXLL, update settings, and give the user information.

To call one of the commands from your script you echo it to the stdout. For example, the command `pyxll-set-option` can be used to set one of PyXLL's configuration options. In a batch file, to set the `LOG/verbosity` setting to `debug` it would be called as follows:

```
ECHO pyxll-set-option LOG verbosity debug
```

Calling the command from Powershell is the same:

```
Echo "pyxll-set-option LOG verbosity debug"
```

Some commands return results back to the script. They do this by writing the result to the script's stdin. To read the result from a command that returns something you need to read it from the stdin into a variable. The command `pyxll-get-command` is one that returns a result and can be used from a batch file as follows:

```
ECHO pyxll-get-option PYTHON executable
SET /p EXECUTABLE=
REM The PYTHON executable setting is now in the variable %EXECUTABLE%
```

Or in Powershell it would look like:

```
Echo "pyxll-get-option PYTHON executable"
$executable = Read-Host
```

Below is a list of the available commands.

- *pyxll-get-option*
- *pyxll-set-option*
- *pyxll-unset-option*

- *pyxll-set-progress*
- *pyxll-show-progress*
- *pyxll-set-progress-status*
- *pyxll-set-progress-title*
- *pyxll-set-progress-caption*
- *pyxll-get-version*
- *pyxll-get-python-version*
- *pyxll-get-arch*
- *pyxll-get-pid*
- *pyxll-reload-config*
- *pyxll-restart-excel*
- *pyxll-set-error-message*

pyxll-get-option

Gets the value of any option from the config.

Takes two arguments, SECTION and OPTION, and returns the option's value.

- Batch File

```
ECHO pyxll-get-option SECTION OPTION
SET /p VALUE=
```

- Powershell

```
Echo "pyxll-get-option SECTION OPTION"
$value = Read-Host
```

If used on a multi-line option (e.g. PYTHON/modules and PYTHON/pythonpath) the value returned will be a list of value delimited by the separator documented for the setting.

pyxll-set-option

Sets a config option.

Takes three arguments, SECTION, OPTION and VALUE. Doesn't return a value.

- Batch File

```
ECHO pyxll-set-option SECTION OPTION VALUE
```

- Powershell

```
Echo "pyxll-set-option SECTION OPTION VALUE"
```

When used with multi-line options (e.g. PYTHON/modules and PYTHON/pythonpath) this command appends to the list of values. Use `pyxll-unset-option` to clear the list first if you want to overwrite any current value.

pyxll-unset-option

Unsets the specified option.

Takes two arguments, SECTION and OPTION. Doesn't return value.

- Batch File

```
ECHO pyxll-unset-option SECTION OPTION
```

- Powershell

```
Echo "pyxll-unset-option SECTION OPTION"
```

pyxll-set-progress

Display or update a progress indicator dialog to inform the user of the current progress.

This is useful for potentially long running start up scripts, such as when downloading files from a network location or installing a large number of files.

Takes one argument, the current progress as a number between 0 and 100. Doesn't return a value.

- Batch File

```
ECHO pyxll-set-progress PERCENT_COMPLETE
```

- Powershell

```
Echo "pyxll-set-progress PERCENT_COMPLETE"
```

pyxll-show-progress

Displays the progress indicator without setting the current progress.

This shows the progress indicator in 'marquee' style where it animates continuously rather than showing any specific progress.

If the progress indicator is already shown this command does nothing.

Takes no arguments and doesn't return a value.

- Batch File

```
ECHO pyxll-show-progress
```

- Powershell

```
Echo "pyxll-show-progress"
```

pyxll-set-progress-status

Sets the status text of the progress indicator dialog.

This does not show the progress indicator if it is not already shown. Use `pyxll-show-progress` or `pyxll-set-progress` to show the progress indicator.

Takes one argument, STATUS, and doesn't return a value.

- Batch File

```
ECHO pyxll-set-progress-status STATUS
```

- Powershell

```
Echo "pyxll-set-progress-status STATUS"
```

pyxll-set-progress-title

Sets the title of the progress indicator dialog.

This does not show the progress indicator if it is not already shown. Use `pyxll-show-progress` or `pyxll-set-progress` to show the progress indicator.

Takes one argument, TITLE, and doesn't return a value.

- Batch File

```
ECHO pyxll-set-progress-title TITLE
```

- Powershell

```
Echo "pyxll-set-progress-title TITLE"
```

pyxll-set-progress-caption

Sets the caption text of the progress indicator dialog.

This does not show the progress indicator if it is not already shown. Use `pyxll-show-progress` or `pyxll-set-progress` to show the progress indicator.

Takes one argument, CAPTION, and doesn't return a value.

- Batch File

```
ECHO pyxll-set-progress-caption CAPTION
```

- Powershell

```
Echo "pyxll-set-progress-caption CAPTION"
```

pyxll-get-version

Gets the version of the installed PyXLL add-in.

Takes no arguments and returns the version.

- Batch File

```
ECHO pyxll-get-version
SET /p VERSION=
```

- Powershell

```
Echo "pyxll-get-version"
$version = Read-Host
```

pyxll-get-python-version

Gets the version of Python the installed PyXLL add-in is compatible with in the form *PY_MAJOR_VERSION.PY_MINOR_VERSION*.

Takes no arguments and returns the Python version.

- Batch File

```
ECHO pyxll-get-python-version
SET /p VERSION=
```

- Powershell

```
Echo "pyxll-get-python-version"
$version = Read-Host
```

pyxll-get-arch

Gets the machine architecture of the Excel process and PyXLL add-in.

Takes no arguments and returns either 'x86' for 32 bit or 'x64' for a 64 bit.

- Batch File

```
ECHO pyxll-get-arch
SET /p ARCH=
```

- Powershell

```
Echo "pyxll-get-arch"
$arch = Read-Host
```

pyxll-get-pid

Gets the process id of the Excel process.

Takes no arguments and the process id.

- Batch File

```
ECHO pyxll-get-pid
SET /p PID=
```

- Powershell

```
Echo "pyxll-get-pid"
$pid = Read-Host
```

pyxll-reload-config

Stops the script and reloads the `pyxll.cfg` config file from scratch. The startup script will be re-run if the config file contains a startup script.

This is useful when applying updates to an environment that include changes to the config file.

Care should be taken so that this command is only used if the config has changed, as otherwise it can result in a loop where the config is continually reloaded and the script is re-run. The number of reloads is limited to avoid an infinite loop, but it will result in an error.

This command takes no arguments and doesn't return a value.

- Batch File

```
ECHO pyxll-reload-config
```

- Powershell

```
Echo "pyxll-reload-config"
```

pyxll-restart-excel

Displays a message box to the user informing them Excel needs to restart. If the user selects 'Ok' then Excel will restart. The user can cancel this and if they do so the script will be terminated.

This can be used if your script needs to install something that would require Excel to be restarted. When Excel restarts your script will be run again and so you should ensure that it doesn't repeatedly request to restart Excel.

One possible use case is if you want to upgrade the PyXLL add-in itself. You can rename the existing one (it can't be deleted while Excel is using it, but it can be renamed) and copy a new one in its place and then request to restart Excel.

Takes one optional argument, MESSAGE, which will be disaplyed to the user. Doesn't return a result.

- Batch File

```
ECHO pyxll-restart-excel MESSAGE
```

- Powershell

```
Echo "pyxll-restart-excel MESSAGE"
```

pyxll-set-error-message

New in PyXLL 5.6

Sets the error message to be displayed to the user if the script fails.

This can be used to customize what the user sees in the message box if the script exits with a non-zero exit code.

- Batch File

```
ECHO pyxll-set-error-message MESSAGE
```

- Powershell

```
Echo "pyxll-set-error-message MESSAGE"
```

3.2.10 Menu Ordering

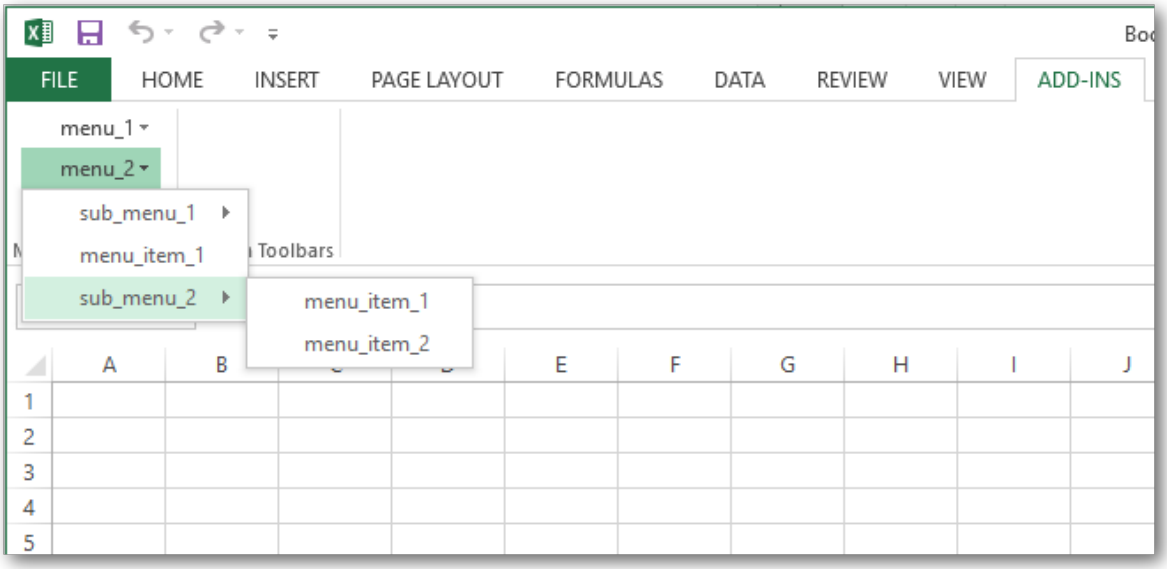
Menu items added via the `@xl_menu` decorator can specify what order they should appear in the menus. This can be also be set, or overridden, in the config file.

To specify the order of sub-menus and items within the sub-menus use a "." between the menu name, sub-menu name and item name.

The example config below shows how to order menus with menu items and sub-menus.

```
[MENUS]
menu_1 = 1  # order of the top level menu menu_1
menu_1.menu_item_1 = 1  # order of the items within menu_1
menu_1.menu_item_2 = 2
menu_1.menu_item_3 = 3
menu_2 = 2  # order of the top level menu menu_2
menu_2.sub_menu_1 = 1  # order of the sub-menu sub_menu_1 within menu_2
menu_2.sub_menu_1.menu_item_1 = 1  # order of the items within sub_menu_1
menu_2.sub_menu_1.menu_item_2 = 2
menu_2.menu_item_1 = 2 # order of item within menu_2
menu_2.sub_menu_2 = 3
menu_2.sub_menu_2.menu_item_1 = 1
menu_2.sub_menu_2.menu_item_2 = 2
```

Here's how the menus appear in Excel:



3.2.11 Shortcuts

Macros can have keyboard shortcuts assigned to them by using the *shortcut* keyword argument to `@xl_macro`. Alternatively, these keyboard shortcuts can be assigned, or overridden, in the config file.

Shortcuts should be one or more modifier key names (*Ctrl*, *Shift* or *Alt*) and a key, separated by the '+' symbol. For example, 'Ctrl+Shift+R'. If the same key combination is already in use by Excel it may not be possible to assign a macro to that combination.

The PyXLL developer macros (reload and rebind) can also have shortcuts assigned to them.

```
[SHORTCUTS]
pyxll.reload = Ctrl+Shift+R
module.macro_function = Alt+F3
```

See *Keyboard Shortcuts* for more details.

3.2.12 Default Decorator Parameters

New in PyXLL 5.10

The decorators `@xl_func`, `@xl_macro`, and `@xl_menu` take various parameters to control their behaviour.

The default parameters for these decorators can be modified through the following configuration options:

```
[PYXLL]
xl_func_defaults = ...
xl_macro_defaults = ...
xl_menu_defaults = ...
```

Each can be set to a function that returns the defaults to be used. The function is specified using its fully qualified function name, including the module or package containing the function.

For example, to specify that the function *my_xl_func_defaults* from the Python module *custom_defaults.py* should be used to fetch default values for the `@xl_func` decorator the following config would be used:

```
[PYXLL]
xl_func_defaults = custom_defaults.my_xl_func_defaults
```

The specified function is called with the decorated function, and should return a dictionary of default parameter values.

Example Use

Suppose we wanted to change the default value for the *name* parameter the `@xl_func`. We might want to do this in order to use an all uppercase name for the Excel function, instead of simply using the function name is it is in Excel.

The following funcition gets the function name in uppercase and returns a dictionary of the defaults to use for the `@xl_func` decorator:

```
def my_xl_func_defaults(func)
    # 'func' is the Python function that will be
    # called from Excel.
    name = func.__name__

    # We want to use the funtion name in all uppercase
    # in Excel.
    xl_name = name.uppercase()

    # Here we return a dictionary of default values
    # for the @xl_func parameters.
    return {
        "name": xl_name
    }
```

We configure PyXLL to use this function for the `@xl_func` decorator default parameters as above.

With this function configured, any function using decorated with the `@xl_func` decorator will now have the *name* parameter defaulted to the upper case function name.

The following Python function would appear in Excel with the name *A_PYTHON_FUNCTION*.

```
@xl_func
def a_python_function():
    return "In Excel!"
```

Different Defaults For Different Dackages

Usually you will not want to apply the same defaults to all modules and packages. You may want to only apply your defaults to your own packages if you are using some other third-party packages that also use PyXLL, or you may want to use different defaults for your own different packages.

The defaults can be configured on a per-module or per-package basis to accomodate this.

Suppose your had two Python modules *my_math_functions* and *my_string_functions* and you wanted all the functions in *my_math_functions* to have the category *Math* and all functions in *my_string_functions* to have the category *String*.

We can write a couple of user default functions that return the default *category* parameter for `@xl_func` as follows:

```
def math_xl_func_defaults(func)
    return {
        "category": "Math"
    }

def string_xl_func_defaults(func)
    return {
        "category": "String"
    }
```

We use the following config to configure those for the two modules *my_math_functions* and *my_string_functions*:

```
[PYXLL]
xl_func_defaults =
    my_math_functions: custom_defaults.math_xl_func_defaults
    my_string_functions: custom_defaults.string_xl_func_defaults
```

If you are using packages and want to include all modules and sub-packages within a package you use * as a wildcard in the module specification in the config:

```
[PYXLL]
xl_func_defaults =
    my_math_package.*: custom_defaults.math_xl_func_defaults
    my_string_package.*: custom_defaults.string_xl_func_defaults
```

3.3 Worksheet Functions

3.3.1 Introduction

Writing an Excel Worksheet Function in Python

If you've not installed the PyXLL addin yet, see *Installing PyXLL*.

PyXLL user defined functions (UDFs) written in Python are exactly the same as any other Excel worksheet function. They are called from formulas in an Excel worksheet in the same way, and appear in Excel's function wizard just like Excel's native functions (see *Function Documentation*).

To tell the PyXLL add-in to expose a Python function so that we can call it from Excel, all that is needed is to add the `@xl_func` decorator to a Python function:

```
from pyxll import xl_func

@xl_func
def hello(name):
    return "Hello, %s" % name
```

This function takes just a single argument, `name`, which can be passed in when we call the function from Excel.

PyXLL supports passing arguments and returning values of many different types, which is covered in detail in the *next section*.

Configuring PyXLL with your Python Module

Once you have saved that code you need to ensure the interpreter can find it by modifying the following settings in your `pyxll.cfg` config file:

- `[PYXLL]` / `modules`

  The list of Python modules that PyXLL will import.

- `[PYTHON]` / `pythonpath`

  The list of folders that Python will look for modules in.

If you saved the above code into a new file called *my_module.py* in a folder *C:\Users\pyxll\modules* you would add the Python module *my_module* to the `modules` list, and *C:\Users\pyxll\modules* to the `pythonpath`.

Note that Python module *file names* end in *.py*, but the Python *module names* do not.

```
[PYXLL]
;
; Make sure that PyXLL imports the module when loaded.
```

(continues on next page)

(continued from previous page)

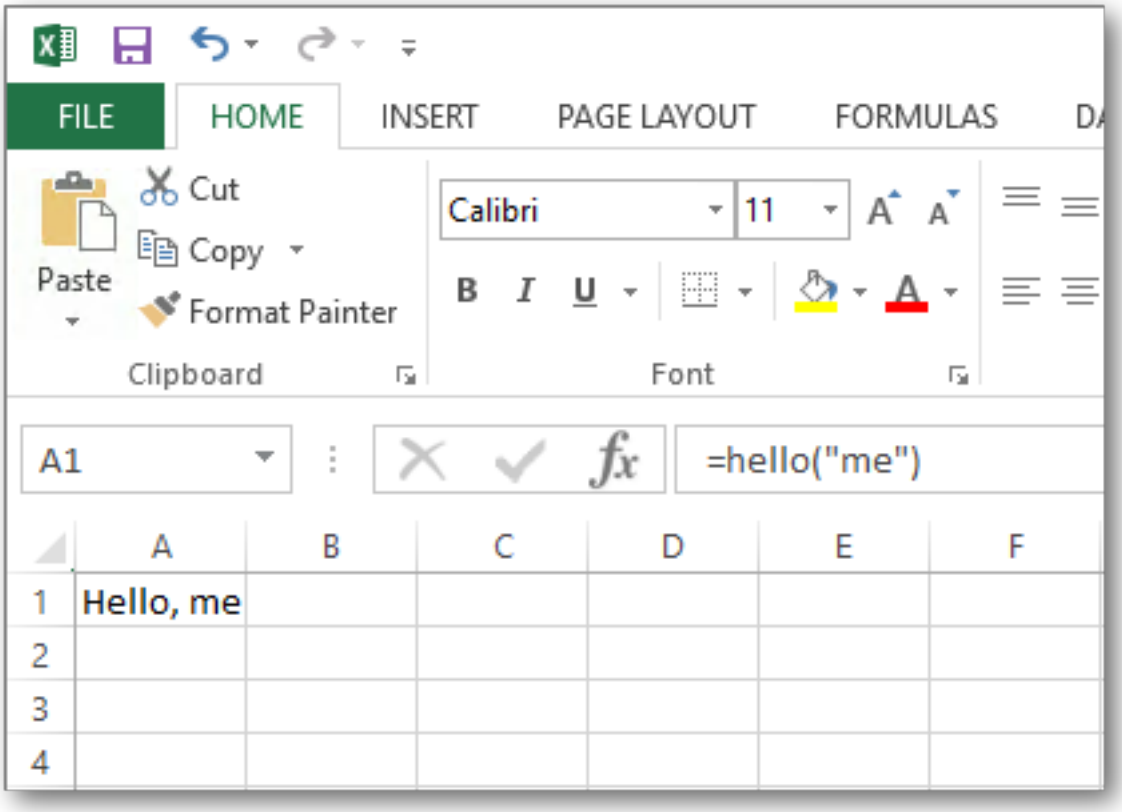
```
;
; We use the module name here, not the file name,
; and so the ".py" file extension is omitted.
;
modules = my_module

[PYTHON]
;
; Ensure that PyXLL can find the module.
; Multiple modules can come from a single directory.
;
pythonpath = C:\Users\pyxll\modules
```

Calling your Python Function from Excel

> **Tip: No Need to Restart Excel!**
>
> Use the 'Reload' menu item under the PyXLL menu to reload your Python code without restarting Excel - this causes all Python modules to be reloaded, making updated code available without the need to restart Excel itself.

After making these changes reload the PyXLL addin, or restart Excel. You can use the PyXLL function you have just added in formulas in any Excel worksheet, because the function was decorated with `@xl_func`.

```
=hello("me")
```



Tip

If your function does not appear in Excel or you get an error message, check the PyXLL log file. By default, the log file will be in the *logs* folder next to the PyXLL add-in.

Different Argument and Return Types

Worksheet functions can take simple values, as in the example above, or more complex arguments including Pandas DataFrames and Numpy arrays.

This is covered in detail in the *next section*.

3.3.2 Argument and Return Types

- *Specifying the Argument and Return Types*
  - *@xl_func Function Signature*
  - *Python Type Hints*
  - *@xl_arg and @xl_return Decorators*
  - *Type Parameters*
- *Standard Types*
  - *Array Types*
  - *The 'var' Type*
  - *Numpy Types*
  - *Pandas Types*
  - *Polars Types*
  - *Dictionary Types*
  - *Dataclass Types*
  - *Union Types*
  - *Optional Types*
  - *Function Type*
  - *Error Types*
- *Using Python Objects Directly*
- *Custom Types*
- *Manual Type Conversion*

Specifying the Argument and Return Types

When you started using PyXLL you probably discovered how easy it is to register a Python function in Excel. To improve efficiency and reduce the chance of errors, you can also specify what types the arguments to that function are expected to be, and what the return type is. This information is commonly known as a function's *signature*. There are three common ways to add a signature to a function, described in the following sections.

Also see *Using Type Hints*.

@xl_func Function Signature

The most common way to provide the signature is to provide a function *signature* as the first argument to `@xl_func`:

```
from pyxll import xl_func
from datetime import date, timedelta

@xl_func("date d, int i: date")
def add_days(d, i):
    return d + timedelta(days=i)
```

When adding a function signature string it is written as a comma separated list of each argument type followed by the argument name, ending with a colon followed by the return type. The signature above specifies that the function takes two arguments called `d`, a date, and `i`, and integer, and returns a value of type date. You may omit the return type; PyXLL automatically converts it into the most appropriate Excel type.

Adding type information is useful as it means that any necessary type conversion is done automatically, before your function is called.

Python Type Hints

Type information can also be provided using type annotations, or hints, in Python 3.

See *Using Type Hints* for more detailed information about using type hints.

This example shows how you pass dates to Python functions from Excel using type annotations:

```
from pyxll import xl_func
from datetime import date, timedelta

@xl_func
def add_days(d: date, i: int) -> date:
    return d + timedelta(days=i)
```

Internally, an Excel date is just a floating-point a number. If you pass a date to a Python function with no type information then that argument will just be a Python float when it is passed to your Python function. Adding a signature removes the need to convert from a float to a date in every function that expects a date. The annotation on your Python function (or the signature argument to `@xl_func`) tells PyXLL and Excel what type you expect, and the the conversion is done automatically.

@xl_arg and @xl_return Decorators

The final way type information can be added to a function is by using specific argument and return type decorators. These are particularly useful for more complex types that require parameters, such as *NumPy arrays* and *Pandas types*. Parameterized types can be specified as part of the function signature, or using `@xl_arg` and `@xl_return`.

For example, the following function takes two 1-dimensional NumPy arrays, using a function signature:

```
from pyxll import xl_func
import numpy as np

@xl_func("numpy_array<ndim=1> a, numpy_array<ndim=1> b: var")
def add_days(a, b):
    return np.correlate(a, b)
```

But this could be re-written using `@xl_arg` as follows:

```
from pyxll import xl_func, xl_arg
import numpy as np
```

(continues on next page)

(continued from previous page)

```
@xl_func
@xl_arg("a", "numpy_array", ndim=1)
@xl_arg("b", "numpy_array", ndim=1)
def add_days(a, b):
    return np.correlate(a, b)
```

Type Parameters

Many types can be parameterised to further control the type conversion between Excel and Python. An example of this is in the section above where we see the `numpy_array` type accepts a type parameter `ndim`.

Type parameters can be specified when using a function signature, or when using the *@xl_arg* and *@xl_return* decorators.

For details of the type parameters available see the specific documentation for the type you are interested in. Type parameters can be different depending on whether it is the argument conversion or return conversion that is being specified.

Standard Types

Several standard types may be used in the signature specified when exposing a Python worksheet function. These types have a straightforward conversion between PyXLL's Excel-oriented types and Python types. Arrays and more complex objects are discussed later.

Below is a list of these basic types. Any of these can be specified as an argument type or return type in a function signature. For some types, Python type hints or annotations can be used.

PyXLL Type	Python Type	Python Type Hint
float	`float`	`float`
int	`int`	`int`
str	`str`	`str`
unicode	`unicode`[4]	N/A
bool	`bool`	`bool`
datetime	`datetime.datetime`[1]	`datetime.datetime`
date	`datetime.date`	`datetime.date`
time	`datetime.time`	`datetime.time`
var	`object`[5]	`typing.Any`
object	`object`[2]	`object`
rtd	*RTD*[3]	*RTD*
xl_cell	*XLCell*[6]	*XLCell*
range	Excel Range COM Wrapper[7]	N/A
function	`function`[8]	`typing.Callable`

[4] Unicode was only introduced in Excel 2007 and is not available in earlier versions. Use *xl_version* to check what version of Excel is being used if in doubt.

[1] Excel represents dates and times as numbers. PyXLL will convert dates and times to and from Excel's number representation, but in Excel they will look like numbers unless formatted. When returning a date or time from a Python function you will need to change the Excel cell formatting to a date or time format.

[5] The `var` type can be used when the argument or return type isn't fixed. Using the more a specific type has the advantage that arguments passed from Excel will get coerced correctly. For example if your function takes an int you'll always get an `int` and there's no need to do type checking in your function. If you use a `var`, you may get a `float` if a number is passed to your function, and if the user passes a non-numeric value your function will still get called so you need to check the type and raise an exception yourself.

If no type information is provided for a function it will be assumed that all arguments and the return type are the `var` type. PyXLL will do its best to perform the necessary conversions, but providing specific information about typing is the best way to ensure that type conversions are correct.

[2] The `object` type in PyXLL lets you pass Python objects between functions as object handles that reference the real objects in an *internal object cache*. You can store object references in spreadsheet cells and use those cell references as function arguments.

For Python's primitive types, use the `var` type instead.

[3] `rtd` is for functions that return *Real Time Data*.

[6] Specifying `xl_cell` as an argument type passes an *XLCell* instance to your function instead of the value of the cell. This is useful if you

Array Types

See *Array Functions* for more details about array functions.

Ranges of cells can be passed from Excel to Python as a 1d or 2d array.

Any type can be used as an array type by appending `[]` for a 1d array or `[][]` for a 2d array:

```
from pyxll import xl_func

@xl_func("float[][] array: float")
def py_sum(array):
    """return the sum of a range of cells"""
    total = 0.0

    # 2d array is a list of lists of floats
    for row in array:
        for cell_value in row:
            total += cell_value

    return total
```

A 1d array is represented in Python as a simple list, and when a simple list is returned to Excel it will be returned as a column of data. A 2d array is a list of lists (list of rows) in Python. To return a single row of data, return it as a 2d list of lists with only a single row.

When returning a 2d array remember that it* must* be a list of lists. This is why you woud return a single a row of data as `[[1, 2, 3, 4]]`, for example. To enter an array formula in Excel you select the cells, enter the formula and then press *Ctrl+Shift+Enter*.

Any type can be used as an array type, but `float[]` and `float[][]` require the least marshalling between Excel and python and are therefore the fastest of the array types.

If you a function argument has no type specified or is using the `var` type, if it is passed a range of data that will be converted into a 2d list of lists of values and passed to the Python function.

Type hints such as `List[int]` can also be used. See *Array Type Hints* for details.

See *NumPy Array Types* and *Pandas Types* for details of how to pass `numpy` and `pandas` types between Excel and Python functions.

The 'var' Type

The `var` type can be used when your function accepts any type. It is also the default type used if no other type is specified.

When an argument is passed from Excel to Python using the `var` type the most appropriate conversion is chosen automatically from the primiative types natively supported by Excel.

need to know the location or some other data about the cell used as an argument as well as its value.

[7] **New in PyXLL 4.4**

The `range` argument type is the same as `xl_cell` except that instead of passing an *`XLCell`* instance a `Range` COM object is used instead.

The default Python COM package used is `win32com`, but this can be changed via an argument to the range type. For example, to use `xlwings` instead of `win32com` you would use `range<xlwings>`.

[8] **New in PyXLL 5.4**

The *function* argument type can be used to pass other *`@xl_func`* functions to Python functions in Excel. This can be useful for functions that require a callback function and is cleaner than specifying the function name as a string and then having to look up the Python function.

The following examples all use the `var` type:

```
from pyxll import xl_func

@xl_func
def my_function(x):
    # As no type was specified, both 'x' and return type type will
    # default to 'var'
    return str(type(x))  # return type is also 'var' as unspecified
```

```
from pyxll import xl_func

@xl_func("x var: str")
def my_function(x):
    # x can be of any type as 'var' was specified as the argument
    # type in the function signature above.
    return str(type(x))  # return type is 'str' from the signature
```

```
from pyxll import xl_func
import typing

@xl_func
def my_function(x: typing.Any) -> str:
    # x will use the 'var' type because the 'Any' type hint was used
    return str(type(x))   # return type is 'str' from the type hint
```

```
from pyxll import xl_func

@xl_func
@xl_arg("x", "var")
def my_function(x):
    # x was specified to use the 'var' type using @xl_arg above
    return str(type(x))  # return type is unspecified as so 'var' is assumed
```

When using see *cached objects* the `var` will, by default, look up the cached object from the object handle passed to the function and pass the object to the function. This can be disabled using the `no_object_lookup` type parameter, for example `var<no_object_lookup=True>`.

> **Note**
>
> The `no_object_lookup` type parameter is new in PyXLL 5.6.

Numpy Types

See *NumPy Array Types* for details about the supported Numpy types.

Pandas Types

See *Pandas Types* for details about the supported Pandas types.

Polars Types

See *Polars DataFrames* for details about the supported Polars types.

Dictionary Types

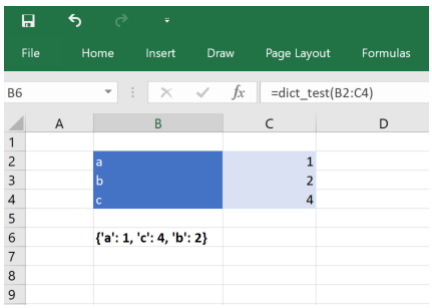
Python functions can be passed a dictionary, converted from an Excel range of values. Dicts in a spreadsheet are represented as a 2xN range of keys and their associated values. The keys are in the columns unless the range's `transpose` argument (see below) is true.

The following is a simple function that accepts an dictionary of integers keyed by strings. Note that the key and value types are optional and default to `var` if not specified.

```
from pyxll import xl_func

@xl_func("dict<str, int>: str")  # Keys are strings, values are integers
def dict_test(x):
    return str(x)
```

From PyXLL 5.8.0, if using Python 3.8 or higher, you can also use the standard `TypedDict` type annotation.

```
from typing import TypedDict
from pyxll import xl_func

class MyTypedDict(TypedDict):
  a: int
  b: int
  c: int

@xl_func
def dict_test(x: MyTypedDict) -> str:
    return str(x)
```



The `dict` type can be parameterized so that you can also specify the key and value types, and some other options.

- **dict**, when used as an argument type

  `dict<key=var, value=var, transpose=False, ignore_missing_keys=True, ignore_missing_values=False>`

  - `key` Type used for the dictionary keys.
  - `value` Type used for the dictionary values.
  - `transpose` - False (the default): Expect the dictionary with the keys on the first column of data and the values on the second. - True: Expect the dictionary with the keys on the first row of data and the values on the second. - None: Try to infer the orientation from the data passed to the function.
  - `ignore_missing_keys` If True, ignore any items where the key is missing.
  - `ignore_missing_values` If True, ignore any items where the value is missing *(new in PyXLL 5.7)*.

- **dict**, when used as an return type

  `dict<key=var, value=var, transpose=False, order_keys=True>`

- `key` Type used for the dictionary keys.
- `value` Type used for the dictionary values.
- `transpose` - False (the default): Return the dictionary as a 2xN range with the keys on the first column of data and the values on the second. - True: Return the dictionary as an Nx2 range with the keys on the first row of data and the values on the second.
- `order_keys` Sort the dictionary by its keys before returning it to Excel.

Dataclass Types

Python *dataclasses* are a convenient way of creating Python classes encapsulating a collection of typed data fields.

They can also be used to make passing structured data objects between Python and Excel simpler.

Working with *dataclasses* in Excel is similar to using *dictionaries*. From Excel, a 2d array of *key, value* pairs can be passed to a function expecting a dataclass and the correct dataclass will be constructed automatically.

For example, the following code defines a dataclass:

```
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0
```

To write an Excel function that accepts a dataclass of this type you simply need to add a type hint to the function argument:

```
from pyxll import xl_func

@xl_func
def cost_of_stock(item: InventoryItem) -> float:
    """Returns the total cost of inventory items on hand."""
    return item.unit_price * item.quantity_on_hand
```

Or, if passing a signature string to *`@xl_func`* instead of using type hints:

```
from pyxll import xl_func

@xl_func("InventoryItem item: float")
def cost_of_stock(item):
    """Returns the total cost of inventory items on hand."""
    return item.unit_price * item.quantity_on_hand
```

Lists of *dataclasses* can also be used. The first column of data is the field names, and subsequent columns are field values for each dataclass instance.

```
from pyxll import xl_func
from typing import List

@xl_func
def total_cost_of_stock(items: List[InventoryItem]) -> float:
    """Returns the total cost of inventory items on hand."""
    total = 0.0
    for item in items:
        total += item.unit_price * item.quantity_on_hand
    return total
```

If you data is laid out with the field names as column headers, use the `transpose` type parameter on the dataclass arguement, for example, using *@xl_arg*:

```
from pyxll import xl_func, xl_arg

@xl_func
@xl_arg("items", transpose=True)
def total_cost_of_stock(items: List[InventoryItem]) -> float:
    ...
```

Or using a signature string to *@xl_func*:

```
from pyxll import xl_func

@xl_func("InvetoryItem<transpose=True>[]: float")
def total_cost_of_stock(items: List[InventoryItem]) -> float:
    ...
```

Dataclasses can also be used as a return type.

> **Note**
>
> Using dataclasses requires PyXLL 5.8.0 and Python 3.7 or later.

Union Types

Union types can be used for functions that accept arguments or return objects of more than one types. In such examples, the `var` type can be used, but then the value may need to be converted in the function when it's more convenient to let PyXLL do the conversion.

> **Note**
>
> Union types are new in PyXLL 5.1

For example, support you have a function that can take either a `date` or a `string`. This is a common situation for functions that can either take an absolute date or a time period. Using the `var` type, Excel dates are passed as numbers since that is how Excel represents dates, and so the conversion from a number to a date must be done using *get_type_converter* (see *Manual Type Conversion*). Using a union type removes the need for thisextra conversion step.

Union types may be specified in the *@xl_func* function signature using the `union<...>` type, with the possible types passed as parameters. They can also be specified using Python type annotations using the `typing.Union` type annotation.

The following is a function that will accept either a `date` or a `str`. The conversion from an Excel date to a Python date is performed automatically if a date is passed, and the function can also accept a string argument from Excel.

```
from pyxll import xl_func
import datetime as dt

@xl_func("union<date, str> value: str")
def date_or_string(value):
    if isinstance(value, dt.date):
        return "Value is a date: %s" % value
    return "Value is a string: %s" % value
```

The same can be written using Python type annotations as follows

```
from pyxll import xl_func
import datetime as dt
import typing

@xl_func
def date_or_string(value: typing.Union[dt.date, str]) -> str:
    if isinstance(value, dt.date):
        return "Value is a date: %s" % value
    return "Value is a string: %s" % value
```

Tip

From Python 3.10 onwards, `Union[A, B]` can be written as `A | B`.

The order the union type arguments are specified in matters.

In the above example, if `str` was placed before `date` then it would not work as intended since the conversion to `str` would take precedence over the conversion to `date`.

PyXLL will attempt to convert the Excel value to a Python value as follows:

1. If the Excel value is a cached object and one or more of the types are `object` or a type based on `object`, each object-like type conversion will be attempted in order from left to right.
2. If the Excel value exactly matches one of the types then the value will not be converted. This check is only possible for primitive types.
3. An attempt will be made to convert the value to each of the listed types in order, from left to right. The result from the first attempted conversion that succeeds will be used.

Note

The logic around converting cached objects was refined in PyXLL 5.6.

Optional Types

Optional types are used for arguments that can be omitted by the user calling the function. If the argument is omitted then `None` will be passed to Python.

Note

Optional types are new in PyXLL 5.6

Optional types may be specified in the *`@xl_func`* function signature using the `optional<type>` type, with the actual type passed as the type parameter. They can also be specified using Python type annotations using the `typing.Optional` type annotation.

The following is a function that will accept an optional `date` argument. If no argument is provided then the function will be called with None.

```
from pyxll import xl_func
import datetime as dt

@xl_func("optional<date> value: str")
def optional_date(value):
    if value is None:
```

(continues on next page)

(continued from previous page)

```
        return "No date"
    return value.strftime("%Y-%m-%d")
```

The same can be written using Python type annotations as follows

```
from pyxll import xl_func
import datetime as dt
import typing

@xl_func
def optional_date(value: typing.Optional[dt.date]) -> str:
    if value is None:
        return "No date"
    return value.strftime("%Y-%m-%d")
```

Function Type

Python functions exposed to Excel using *@xl_func* can be passed in to other Python functions from Excel using the `function` type. This is useful when you have one function that takes another callback function that the user can select.

> **Note**
>
> The function type is new in PyXLL 5.4.

For example, the following function takes a list of values and a function and calls that function on each value and returns the result:

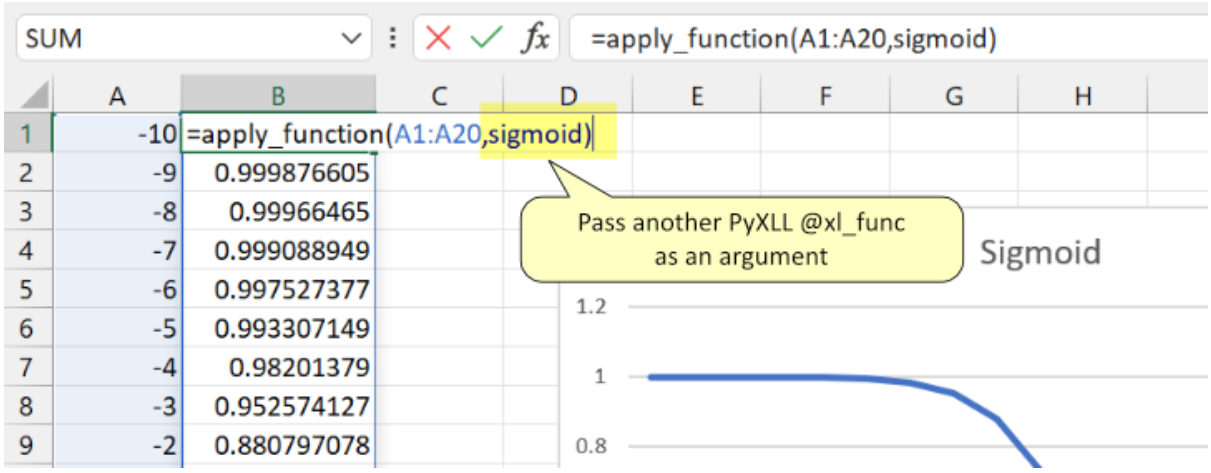
```
from pyxll import xl_func

@xl_func("float[] values, function fn: float[]")
def apply_function(values, fn):
    return [fn(value) for value in values]
```

This function can be called from Excel passing in another *@xl_func* function.

```
from pyxll import xl_func
import math

@xl_func("float x: float")
def sigmoid(x):
    return 1  / (1 + math.exp(x))
```

We can call our `sigmoid` function on a single value in Excel, or using the `apply_function` function above we can pass in the `sigmoid` function and call it for list of values.

```
=apply_function(A1:A20, sigmoid)
```



If you are using Python type annotations instead of passing a signature string the types `typing.Callable` or `collections.abc.Callable` may be used to specify the argument is a function.

> **Warning**
>
> Only PyXLL functions can be passed as function arguments. You cannot pass standard Excel functions, VBA functions or functions from other add-ins to Python functions using the function type.

Error Types

Also See *Error Handling*.

Excel errors can be passed to, or returned from, Python functions that use the `var` or `exception` types. Similarly, Python functions can return specific errors to Excel by returning Python Exception objects.

PyXLL maps Excel errors to Python Exception types as specified in the following table:

Excel error	Python exception type
#NULL!	`LookupError`
#DIV/0!	`ZeroDivisionError`
#VALUE!	`ValueError`
#REF!	`ReferenceError`
#NAME!	`NameError`
#NUM!	`ArithmeticError`
#NA!	`RuntimeError`

New in PyXLL 5.9

To pass errors as argument or return types, as well as using the `var` type the explicit `exception` type can be used. The `exception` type takes a type parameter `cls` to further specify the exception type.

Alternatively, a Python type annotation of `Exception` or other Exception type can be used.

Using the `exception` type explicitly is usually used when a function can accept or return either a value or an error. The `union` type can be used in conjunction with the `exception` type for this purpose.

The example below shows a function that can return a value or a `ValueError` exception:

```
from pyxll import xl_func

@xl_func("bool x: union<str, exception<cls=ValueError>>")
def string_or_error(x):
    if x:
```

(continues on next page)

(continued from previous page)

```
        return "OK!"
    return ValueError()
```

Below is an equivalent function written using Python type annotations:

```
from pyxll import xl_func

@xl_func
def string_or_error(x: bool) -> str | ValueError:
    if x:
        return "OK!"
    return ValueError()
```

Using Python Objects Directly

Not all Python types can be conveniently converted to a type that can be represented in Excel.

Even for types that can be represented in Excel it is not always desirable to do so (for example, and Pandas DataFrame with millions of rows could be returned to Excel as a range of data, but it would not be very useful and would make Excel very slow).

For cases like these, PyXLL can return a handle to the Python object to Excel instead of trying to convert the object to an Excel friendly representation. The actual object is held in PyXLL's object cache until it is no longer needed. This allows for Python objects to be passed between Excel functions easily, without the complexity or possible performance problems of converting them between the Python and Excel representations.

For more information about how PyXLL can automatically cache objects to be passed between Excel functions as object handles, see *Cached Objects*.

Custom Types

As well as the standard types listed above you can also define your own argument and return types, which can then be used in your function signatures.

Custom argument types need a function that will convert a standard Python type to the custom type, which will then be passed to your function. For example, if you have a function that takes an instance of type `X`, you can declare a function to convert from a standard type to `X` and then use *X* as a type in your function signature. When called from Excel, your conversion function will be called with an instance of the base type, and then your exposed UDF will be called with the result of that conversion.

To declare a custom type, you use the `@xl_arg_type` decorator on your conversion function. The `@xl_arg_type` decorator takes at least two arguments, the name of your custom type and the base type.

Here's an example of a simple custom type:

```
from pyxll import xl_arg_type, xl_func

class CustomType:
    def __init__(self, x):
        self.x = x

@xl_arg_type("CustomType", "string")
def string_to_customtype(x):
    return CustomType(x)

@xl_func("CustomType x: bool")
def test_custom_type_arg(x):
    # this function is called from Excel with a string, and then
    # string_to_customtype is called to convert that to a CustomType
```

(continues on next page)

(continued from previous page)

```
    # and then this function is called with that instance
    return isinstance(x, CustomType)
```

You can now use *CustomType* as an argument type in a function signature. The Excel UDF will take a string, but when your Python function is called the conversion function will have been used invisibly to automatically convert that string to a `CustomType` instance.

To use a custom type as a return type you also have to specify the conversion function from your custom type to a base type. This is exactly the reverse of the custom argument type conversion described previously.

The custom return type conversion function must be decorated with the *@xl_return_type* decorator.

For the previous example the return type conversion function could look like:

```
from pyxll import xl_return_type, xl_func

@xl_return_type("CustomType", "string")
def customtype_to_string(x):
    # x is an instance of CustomType
    return x.x

@xl_func("string x: CustomType")
def test_returning_custom_type(x):
    # the returned object will get converted to a string
    # using customtype_to_string before being returned to Excel
    return CustomType(x)
```

Any recognized type can be used as a base type. That can be a standard Python type, an array type or another custom type (or even an array of a custom type!). The only restriction is that it must resolve to a standard type eventually.

Custom types can be parameterized by adding additional keyword arguments to the conversion functions. Values for these arguments are passed in from the type specification in the function signature, or using *@xl_arg* and *@xl_return*:

```
from pyxll import xl_arg_type, xl_func

class CustomType2:
    def __init__(self, x, y):
        self.x = x
        self.y = y

@xl_arg_type("CustomType2", "string", y=None)
def string_to_customtype2(x):
    return CustomType(x, y)

@xl_func("CustomType2<y=1> x: bool")
def test_custom_type_arg2(x):
    assert x.y == 1
    return isinstance(x, CustomType)
```

Manual Type Conversion

Sometimes it's useful to be able to convert from one type to another, but it's not always convenient to have to determine the chain of functions to call to convert from one type to another.

For example, you might have a function that takes an array of *var* types, but some of those may actually be *datetimes*, or one of your own custom types.

To convert them to those types you would have to check what type has actually been passed to your function and then decide what to call to get it into exactly the type you want.

PyXLL includes the function `get_type_converter` to do this for you. It takes the names of the source and target types and returns a function that will perform the conversion, if possible.

Here's an example that shows how to get a `datetime` from a `var` parameter:

```
from pyxll import xl_func, get_type_converter
from datetime import datetime

@xl_func("var x: string")
def var_datetime_func(x):
    var_to_datetime = get_type_converter("var", "datetime")
    dt = var_to_datetime(x)
    # dt is now of type 'datetime'
    return "%s : %s" % (dt, type(dt))
```

3.3.3 Cached Objects

- *Introduction*
- *Accessing Cached Objects in Macros*
- *Populating the Cache On Loading*
- *Saving Objects in the Workbook*
- *Custom Object Handles*
- *Mixing Primitive Values and Objects*
- *Clearing the Cache on Reloading*

Introduction

PyXLL can pass Python objects between Excel functions even if the Python object can't be converted to a type that can be represented in Excel. It does this by maintaining an object cache and returning handles to objects in the cache. The cached object is automatically retrieved when an object handle is passed to another PyXLL function.

Even for types that can be represented in Excel it is not always desirable to do so (for example, a Pandas DataFrame with millions of rows could be returned to Excel as a range of data but it would not be very useful and would make Excel slow).

Instead of trying to convert the object to an Excel friendly representation, PyXLL can cache the Python object and return a handle to that cached object to Excel. The actual object is held in PyXLL's object cache until it is no longer needed. This allows for Python objects to be passed between Excel functions easily, and without the complexity or possible performance problems of converting them between the Python and Excel representations.

The following example shows a case where we return a compiled regex object. Here, it is not desirable to convert the returned Python object to a primitive Excel value, and so we return it as an object:

```
from pyxll import xl_func
import re

# This function returns a compiled regex object, which can't be represented as an␣
↪plain Excel value
@xl_func
def compile_pattern(pattern: str) -> object:
    """Compile a regex pattern and return it as a cached object."""
```

(continues on next page)

(continued from previous page)

```
    return re.compile(pattern)

# This function takes the earlier compiled regex object, passed from Excel as a␣
↪cached object handle
@xl_func
def count_matches(compiled: object, text: str) -> int:
    """Count matches of a compiled regex pattern in text."""
    return len(compiled.findall(text))
```

In Excel, we call these functions as normal and the compiled regex object is returned as a cached object handle, which can then be passed to our `count_matches` function.

Functions can be annotated explicitly to take or return cached object handles using the `object` type. If no type, or `var`, is used, object handle arguments will be converted automatically, and returned non-primitive types will be returned as cached object handles.

Tip

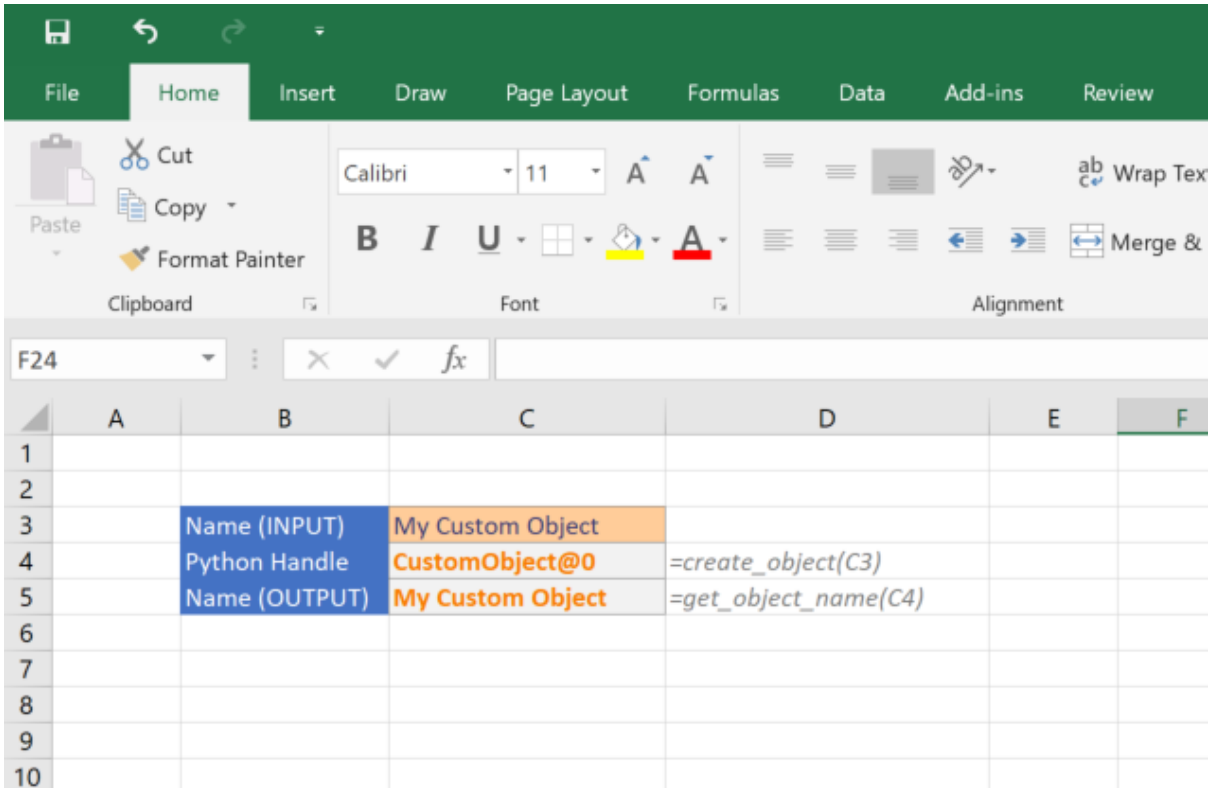
When to use `object` vs `var`?

Use `object` when your function returns something that has no meaningful Excel representation — a custom class instance, a large DataFrame you want to pass between functions efficiently, or any other Python object. PyXLL will store it in the object cache and return a handle to it instead. That handle can then be passed to other PyXLL functions, which will receive the original Python object.

Use `var` when your function returns a Python primitive — a number, string, boolean, date, or other value that Excel can represent natively. PyXLL will convert it automatically. If a non-primitive type is returned, the `var` type will automatically cache the object and return a handle.

The next example shows one function that returns a Python object, and another that takes that Python object as an argument:

```
from pyxll import xl_func

class CustomObject:
    def __init__(self, name):
        self.name = name

@xl_func("string name: object")
def create_object(x):
    return CustomObject(x)

@xl_func("object x: string")
def get_object_name(x):
    assert isinstance(x, CustomObject)
    return x.name
```



When an object is returned in this way it is added to an internal object cache. This cache is managed by PyXLL so that objects are evicted from the cache when they are no longer needed.

When using the `var` type, if an object of a type that has no converter is returned then the `object` type is used. When passing an object handle to a function where the argument type is the `var` type (or unspecified) then the object will be retrieved from the cache and passed to the function automatically.

> **Note**
>
> Objects returned from Excel to Python as cached objects are **not copied**. When an object handle is passed to another function, the object retrieved from the cache is the same object that was previously returned.
>
> You should be careful not to modify these objects. Instead, if you need to modify the object, you should copy or clone the object and only modify the copy.

Accessing Cached Objects in Macros

When writing an Excel macro, if you need to access a cached object from a cell or set a cell value and cache an object you can use the *`XLCell`* class. Using the *`XLCell.options`* method you can set the type to `object` before getting or setting the cell value. For example:

```
from pyxll import xl_macro, xl_app, XLCell

@xl_macro
def get_cached_object():
    """Get an object from the cache and print it to the log"""
    # Get the Excel.Application object
    xl = xl_app()

    # Get the current selection Range object
    selection = xl.Selection

    # Get the cached object stored at the selection
```

(continues on next page)

(continued from previous page)

```
    cell = XLCell.from_range(selection)
    obj = cell.options(type="object").value

    # 'value' is the actual Python object, not the handle
    print(obj)


@xl_macro
def set_cached_object():
    """Cache a Python object by setting it on a cell"""
    # Get the Excel.Application object
    xl = xl_app()

    # Get the current selection Range object
    selection = xl.Selection

    # Create our Python object
    obj = object()

    # Cache the object in the selected cell
    cell = XLCell.from_range(selection)
    cell.options(type="object").value = obj
```

Instead of using a cell reference it is also possible to fetch an object from the cache by its handle. To do this use *get_type_converter* to convert the `str` handle to an `object`, e.g.:

```
from pyxll import xl_func, get_type_converter

@xl_macro("str handle: bool")
def check_object_handle(handle):
    # Get the function to lookup and object from its handle
    get_cached_object = get_type_converter("str", "object")

    # Get the cached object from the handle
    obj = get_cached_object(handle)

    # Check the returned object is of the expected type
    return isinstance(obj, MyClass)
```

Populating the Cache On Loading

When Excel first starts the cache is empty and so functions returning objects must be run to populate the cache.

PyXLL has a feature that enables functions to be called automatically when loading a workbook (*Recalculating On Open*). When a workbook is opened any cell containing a function that has been set to recalculate on open will be recalculated when that workbook is opened and calculated.

By default, functions that return the type `object` are *not* automatically recalculated when a saved workbook is opened, so use `recalc_on_open=True` in the *@xl_func* decorator for any function that needs its cache entry to be populated as soon as the workbook opens, eg:

```
from pyxll import xl_func

@xl_func(": object", recalc_on_open=True)
def calc_on_open():
    # populate the object cache when the workbook opens
    return obj
```

You can change the default behaviour so that `recalc_on_open` is `True` for all functions returning objects unless explicitly marked otherwise by setting `recalc_cached_objects_on_open = 1`. This ensures the cache is populated at the time the workbook opens and is first calculated, avoiding the need to fully recalculate the entire workbook. Individual functions can still opt out by passing `recalc_on_open=False` to *`@xl_func`*.

```
[PYXLL]
recalc_cached_objects_on_open = 1
```

> **Note**
>
> This feature is new in PyXLL 4.5. For prior versions, or for workbooks saved using prior versions, the workbook will need to be recalculated by pressing *Ctrl+Alt+F9* to populate the cache.

Saving Objects in the Workbook

Rather than having to recalculate functions to recreate the cached objects PyXLL can serialize and save cached objects as part of the Excel *Workbook Metadata*.

This is useful if you have objects that take a long time to calculate and they don't need to be recreated each time the workbook is open.

Caution should be used when deciding whether or not to use this function. It is usually better to source data from an external data source and load it in each time. Referencing an external data source ensures that you always see a consistent, up to date view of the data. There are times when saving objects in the Workbook is more convenient and we only advise that you consider which option is right for your use-case.

Saving objects as part of the workbook will inevitably increase the size of the workbook file, and so you should also consider how large the objects to be saved are. Excel should never be used as a replacement for a database!

To have PyXLL save the result of a function use the `save` parameter to the `object` return type:

```
from pyxll import xl_func

@xl_func(": object<save=True>")
def function_with_saved_result():
    # Construct an object and return it
    return obj
```

When calling a function like the one above the object handle will be slightly different to a normal object handle. For objects that are saved the object handle needs to be globally unique and not just unique in the current Excel session. This is because when the object is loaded it will keep the same id and that must not conflict with any other objects that may already exist in the Excel session. If you are using your own *custom object handle* you must take this into consideration.

Objects that are to be saved must be **pickle-able**. This means that they must be able to be serialized and deserialized using Python's `pickle` module. They are serialized and added to the workbook metadata when the workbook is saved.

See https://docs.python.org/3/library/pickle.html for details about Python's pickle module.

Note that the Python code required to reconstruct the pickled objects must be available when opening a workbook containing saved objects in order for those objects to be deserialized and entered into the object cache.

Loading saved objects can be disabled by setting `disable_loading_objects = 1` in the `PYXLL` section of the pyxll.cfg config file.

```
[PYXLL]
disable_loading_objects = 1
```

> **Note**
>
> This feature is new in PyXLL 5.0.

Custom Object Handles

The method of generating object handles can be customized by setting `get_cached_object_id` in the `PYXLL` section of the *config file*.

The generated object handles must be unique as each object is stored in the cache keyed by its object handle. For objects that will be saved as part of the workbook it's important to use a globally unique identifier as those objects will be loaded with the same id later and must not conflict with other objects that may have already been loaded into the object cache.

Only one function can be registered for generating object handles for all cached objects. Different formats of object handles for different object types can be generated by inspecting the type of the object being cached.

The following example shows a simple function that returns an object handle from an object. Note that it uses the 'id' function to ensure that no two objects can have the same handle. When the kwarg `save` is set to `True` that indicates that the object may be serialized and saved as part of the workbook and so a globally unique identifier is used in that case.

```
def get_custom_object_id(obj, save=False):
    if save:
        return str(uuid.uuid4())
    return "[Cached %s <0x%x>]" % (type(obj), id(obj))
```

To use the above function to generate the object handles for PyXLL's object cache it needs to be configured in the `pyxll.cfg` config file. This is done using the fully qualified function name, including the module the function is declared in.

Listing 1: module_name.py

```
[PYXLL]
get_cached_object_id = module_name.get_custom_object_id
```

The `save` kwarg in the custom object id function indicates whether or not the object may be saved in the workbook. When objects are saved the same id is reused when loading the workbook later, and so these ids should be globally unique to avoid conflicts with other existing objects. See *Saving Objects in the Workbook*.

> **Note**
>
> Prior to PyXLL 5.0 the custom object handle function did not take the `save` kwarg.

Mixing Primitive Values and Objects

If you have a function that you want to return an object in some cases and a primitive value, like a number or string, in other cases then you can use the `skip_primitives` parameter to the `object` return type.

Listing 2: pyxll.cfg

```
from pyxll import xl_func
from random import random

@xl_func("int x: object<skip_primitives=True>")
def func(x):
    if x == 0:
        # returned as a number to Excel
```

(continues on next page)

(continued from previous page)

```
        return 0

    # return a list of values as an 'object'
    array = [random() for i in range(x)]
    return array
```

When `skip_parameters` is set to `True` then the following types will not be returned as object handles:

- int
- float
- str
- bool
- Exception
- datetime.date
- datetime.datetime
- datetime.time

If you need more control over what types are considered primitive you can pass a tuple of types as the `skip_primitives` parameter.

Clearing the Cache on Reloading

Whenever PyXLL is reloaded the object cache is cleared. This is because the cached objects may be instances of old class definitions that have since been reloaded. Using instances of old class definitions may lead to unexpected behaviour.

If you know that you are not reloading any classes used by cached objects, or if you are comfortable knowing that the cached objects may be instances of old classes, then you can disable PyXLL from clearing the cache when reloading. To do this, set `clear_object_cache_on_reload = 0` in your pyxll.cfg file.

This is only recommended if you completely understand the above and are aware of the implications of potentially using instances of old classes that have since been reloaded. One common problem is that methods that have been changed are not updated for these instances, and `isinstance` will fail if checked using the new reloaded class.

```
[PYXLL]
clear_object_cache_on_reload = 0
```

To have your functions recalculated automatically after reloading, and repopulate the object cache, you can do so using the `recalc_on_reload` option to *`@xl_func`* as desribed in the section *Recalculating On Reload*.

3.3.4 Cached Functions

New in PyXLL 5.11

If you have one or more functions that take a long time to execute, or are called repeatedly with the same arguments, caching the function can speed things up.

PyXLL includes an implementation of a *Least Recently Used Cache*, or LRU Cache, that will cache function results so that repeated calls with the same arguments return a cached result instead of repeatedly calling the actual Python function.

The *Least Recently Used* cache stores up to a maximum number of results, keyed by the function arguments. Once the maximum number of results is exceeded, the least recently used result is removed from the cache to make space for the latest result.

> **Note**
>
> For other cache methods aside from LRU caching, see *Alternative Caching Methods*.

The LRU cache is enabled for a function using the `lru_cache` keyword argument to `@xl_func`:

```
from pyxll import xl_func

@xl_func(lru_cache=3)
def cached_function(a, b):
    ...
```


- *Why does Excel recalculate a function?*
- *Enabling LRU Caching for a function*
- *LRU Cache Statistics*
- *Clearing the LRU Cache*
- *PyXLL's LRU Cache vs functools.lru_cache*
- *Why do dependent cells still get recalculated?*
- *Alternative Caching Methods*
  - *Example*
  - *Optional Cache Properties*

Why does Excel recalculate a function?

Excel tracks when each cell needs to be recalculated. To do this, it keeps a record of which cells depend on which other cells - this is referred to as the Excel Dependency Graph. When a cell is updated, Excel knows that any other cells that depend on that cell need to be recalculated.

When in automatic calculation mode, when Excel determines a cell needs to be recalculated it will do so immediately.

In manual calculation mode, when Excel determines a cell nees to be recalculated it will mark the cell as *dirty* and it will be calculated when Excel next recalculates (usually when the user presses *F9*).

Excel doesn't consider the cell value when deciding if a cell needs to be recalculated or not. If a cell is updated with the same value, Excel will still recalculate any dependent cells.

There are other reasons Excel can decide a cell needs to be recalculated. For example, if a function used in the cell, or if one of the cell's dependencies, is *volatile*. Various standard Excel functions are *volatile* functions, including `TODAY`, `NOW`, `RAND`, `RANDBETWEEN`, `INDIRECT`, `OFFSET`, `INFO`, `CELL`, and `SUMIF`.

Excel can also decide that a cell needs recalculating after rows or columns are inserted or deleted.

For a complete description of when Excel might determine a cell needs recalculating see the Excel documentation page Excel Recalcuation.

Enabling LRU Caching for a function

To enable LRU caching for a PyXLL worksheet function you pass the keyword argument `lru_cache` to the `@xl_func` decorator.

The value of the `lru_cache` keyword argument can be an integer, `True`, or `False`.

- If `lru_cache` is a positive integer then the LRU cache will cache up to a maximum number of results set by the number.

- If `lru_cache` is `True`, zero, or a negative integer, the size of the cache will be infinite.
- If `lru_cache` is `False` or `None` the function will not be cached.

For example, the following function will cache up to 3 results, keyed by its arguments:

```
from pyxll import xl_func

@xl_func(lru_cache=3)
def cached_function(a, b):
    ...
```

> **Note**
>
> PyXLL's LRU cache is only used when calling the function from Excel.
>
> If the function is called from Python then the cache is not used.
>
> For caching functions when called from Python, see `functools.lru_cache` in the standard Python library for an alternative.

LRU Cache Statistics

It is possible to query the cache for some basic statistics for each cached function.

The function *`lru_cache_info`* when called with a function returns a dictionary with the following keys:

- `maxsize` - maximum number of cached reuslts, or 0 if unbounded
- `currsize` - current number of cached results
- `hits` - number of times a cached result has been returned
- `misses` - number of times the function was called without finding a cached result

If no results are found an empty dictionary is returned.

`lru_cache_info` can also be called with no arguments. In that case it returns a dictionary of function names to dictionaries of the statistics listed above.

Clearing the LRU Cache

The LRU cache is always cleared when the PyXLL add-in is reloaded.

It can also be cleared using the *`lru_cache_clear`* function.

If *`lru_cache_clear`* is called with a function, only cached results for that function are cleared.

If *`lru_cache_clear`* is called with no arguments, all cached results are cleared.

PyXLL's LRU Cache vs functools.lru_cache

PyXLL's LRU cache is conceptually very similar to `functools.lru_cache` from the Python standard library. There are however some important differences.

Python's `functools.lru_cache` requires all function arguments to be *hashable*. This is required as it needs to store the arguments efficiently and quickly determine if a function call has been made with the same arguments before.

This can be problematic when using arguments such as DataFrames or other more complex Python objects that are not easily hashable.

PyXLL's LRU cache has the same requirement as it works in a similar way, *but* it tests the arguments *before* they are converted to the final Python values. Because Excel values are always simple primitives (strings, numbers, booleans, etc) they are always easily hashable. Furthermore, when a cached result already exists for a set of

arguments the cached result can be returned without doing the full conversion from the primitive Excel values to the final Python arguments, which can reduce the total time taken.

Python's `functools.lru_cache` has no knowledge of Excel's RTD or asynchronous functions. Wrapping one of these functions using `functools.lru_cache` takes a little more work. PyXLL's LRU cache on the other hand knows about these different types of functions and handles them automatically.

Once final difference is with how PyXLL's cached objects are handled. When using PyXLL's LRU cache, if a function returns a cached object back to Excel the object handle used doesn't update when a cached result is found. This is intentional as it means that dependent functions can also be cached and don't need to be recalculated unless the object handle updates. When using `functools.lru_cache`, if the function returns a cached object back to Excel then the object handle will update each time regardless of whether the result was cached or not.

Why do dependent cells still get recalculated?

If you have a sheet with cells dependent on a function using the LRU cache, those dependent cells will still be recalculated even if a cached result is returned.

For example, suppose you have the following:

```
A1 = 1
A2 = foo(A1)
A3 = bar(A2)
```

Where `foo` is a cached function using `lru_cache` and `bar` is not.

When `A1` is changed, Excel will mark `A2` and `A3` as being *dirty*. When Excel next recalculates, either automatically or when triggered manually, it will recalculate the cells in their dependency order `A2` followed by `A3`.

When recalculating `A2` if `foo` has a cached result for the updated argument that that cached result will be called without calling the Python function `foo`.

This doesn't affect the fact that Excel has already marked `A3` as being dirty though. Even if the result returned to cell `A2` hasn't changed, Excel will still consider `A3` as dirty.

Excel will continue its recalculation and still recalculate cell `A3` and call the function `bar`.

While it is not possible to change how Excel recalculates, enabling LRU caching for `bar` also will mean that if it has been called with the same arguments previously (and the result is still cached) the cached result can be used.

Alternative Caching Methods

New in PyXLL 5.12

You can provide your own cache to use instead of the standard LRU cache implementation using the `cache` kwarg to *`@xl_func`* and *`@xl_macro`*.

Your cache object must be a mutable mapping.

> **Tip**
>
> The third party package cachetools provides a number of cache implementations that can be used to cache PyXLL functions.

Each time the cached function is called the cache object will get queried for an existing result using it's `__getitem__` method. If a `KeyError` exception is raised then the function will be called and the result will be stored in the cache using `__setitem__`.

Cached items are keyed by a hashable and comparable type, *`pyxll.CacheKey`*. Values are not stored as the plain values returned by the function, but instead as instances of *`pyxll.CachedValue`*. This is because more than just the returned Python value is stored to avoid repeating work when returning the value to Excel.

If a function fails the exception is cached instead of the returned value. The exception is wrapped in a *pyxll.CachedValue* instance in the same way a successfully returned value is. Therefore, it is not necessary for the cache to treat failed function calls differently than those that have returned successfully.

Example

The following example shows a minimal cache class that only stores the most recent value.

```
from pyxll import xl_func, CacheKey, CachedValue

class CustomCache:
    """Caches only the most recent result"""

    def __init__(self):
        self.__last_key = None
        self.__last_value = None

    def __getitem__(self, key: CacheKey) -> CachedValue:
        # Retrieve a cached value
        if key == self.__last_key:
            return self.__last_value
        raise KeyError(key)

    def __setitem__(self, key: CacheKey, value: CachedValue):
        # Store the value in the cache
        self.__last_key = key
        self.__last_value = value

    def __len__(self):
        # Return the number of cached objects
        return 1 if self.__last_key is not None else 0

    def clear(self):
        # Clear any cached value
        self.__last_key = None
        self.__last_value = None

# Pass an instance of our custom cache as the function cache object
@xl_func(cache=CustomCache())
def cached_function(a, b, c):
    # If the Excel functions is called twice in a row with the same arguments
    # then the Python function will only be called once and the cached result
    # will be returned the second time.
    print(f"Cached function called with {a=}, {b=}, {c=}")
    return a + b + c
```

Optional Cache Properties

When implementing a custom cache you can add the following properties to your cache class. These are use to populate the information returned by *lru_cache_info*.

- `maxsize`: Return maximum number of cached reuslts, or 0 if unbounded.
- `currsize`: Return the current number of objects in the cache. If not available, `len(cache)` will be used instead.

3.3.5 Array Functions

- *Array Functions in Python*
- *Array Type Hints*
  - *Trimming Array Arguments*
- *Ctrl+Shift+Enter (CSE) Array Functions*
- *Auto Resizing Array Functions*
- *Dynamic Array Functions*

Any function that returns an array (or range) of data in Excel is called an *array function*.

Depending on what version of Excel you are using, array functions are either entered as a *Ctrl+Shift+Enter (CSE) formula*, or as a *dynamic array formula*. Dynamic array formulas have the advantage over CSE formulas that they automatically resize according to the size of the result.

To help users of older Excel versions, PyXLL array function results can be *automatically re-sized* .

The `#SPILL!` error indicates that the array would overwrite other data.

Array Functions in Python

Any function exposed to Excel using the `@xl_func` decorator that returns a list of values is an array function.

If a function returns a list of simple values (not lists) then it will be returned to Excel as a column of data. Rectanguler ranges of data can be returned by returning a list of lists, eg:

```
from pyxll import xl_func

@xl_func
def array_function():
    return [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
    ]
```

An optional function signature passed to `@xl_func` can be used to specify the return type. The suffix `[]` is used for a 1d array (column), e.g. `float[]`, and `[][]` is used for a 2d array, e.g. `float[][]`.

For example, the following function takes 1d array (list of values) and returns a 2d array of values (list of lists):

```
from pyxll import xl_func

@xl_func("float[]: float[][]")
def diagonal(v):
    d = []
    for i, x in enumerate(v):
        d.append([x if j == i else 0.0 for j in range(len(v))])
    return d
```

NumPy arrays and Pandas types (DataFrames, Series etc) can also be returned as arrays to Excel by specifying the relevant type in the function signature. See *NumPy Array Types* and *Pandas Types* for more details.

When entering an array formula in Excel it should be entered as a *Ctrl+Shift+Enter (CSE) formula*, or if using *Dynamic Arrays* or PyXLL's *array auto-sizing* feature then they can be entered in the same way as any other formula.

Array Type Hints

New in PyXLL 5.7

As well as specifying the argument or return type as an array using the `type[]` syntax in the *`@xl_func`* signature or to *`@xl_arg`* or *`@xl_return`* decorators, you can specify if you want the Python type to be a `list` (same as `type[]`), `tuple` or `set`.

Python type hints can also be used.

PyXLL Type	Python Type	Python Type Hint
list	`list`	`list` or `typing.List`
tuple	`tuple`	`list` or `typing.Tuple`
set	`set`	`set` or `typing.Set`

These array types have the following type parameters:

- **list, tuple, and set**
  - `T` Type used for the item type.
  - `trim` Set to true to trim the input range to exclude blank or empty cells. See *Trimming Array Arguments*.
- **list and tuple**
  - `ndim` 1 or 2, for 1d or 2d collections.

For example, the following function accepts a tuple of floats:

```
from pyxll import xl_func

@xl_func("tuple<float> x: str")
def tuple_func(x):
    # x is a tuple of strings
    return str(x)
```

This could also be specified using a Python type hint as follows:

```
from pyxll import xl_func

@xl_func
def tuple_func(x: tuple[float]):
    # x is a tuple of floats
    return str(x)
```

Two dimensional lists and tuples can also be used, for example:

```
from pyxll import xl_func

# Note: This is equivalent to 'int[][]'
@xl_func("list<int, ndim=2> x: str")
def tuple_func(x):
    # x is a list of lists of strings
    return str(x)
```

This could also be specified using a Python type hint as follows:

```
from pyxll import xl_func

@xl_func
```

(continues on next page)

(continued from previous page)

```
def tuple_func(x: list[list[[int]]):
    # x is a list of lists of strings
    return str(x)
```

Trimming Array Arguments

New in PyXLL 5.12

When passing an array as an argument to an Excel function sometimes you might want to pass a larger range than the actual data.

For example, when creating a sheet you might allow some extra rows in case the data grows, or to allow space for the user of the sheet to add more rows. You might even select a range of entire columns.

Converting a larger range than necessary takes additional time when constructing the Python collection object. Instead of constructing the entire object and then removing empty rows in your Python code, it is much more efficient (and easier!) for PyXLL to trim any trailing empty rows or columns from the Excel data before constructing the Python collection.

To tell PyXLL to trim empty space use the `trim` type parameter to the `list`, `tuple`, or `set` type.

For example:

```
from pyxll import xl_func

@xl_func("list<ndim=2, trim=True> data, int row: float")
def sum_row(data, row):
    """Sum the values in a row."""
    row_values = data[row]
    return sum(row_values)
```

The function above can now accept a large range including empty rows and columns around the actual data. Before the `list` is constructed the data is trimmed to remove empty rows and columns from the top, bottom, left and right of the data.

When specifying `trim=True` any empty rows from the top or bottom of the range, or empty columns from the left or right of the range, will be removed.

If you only want to trim rows or columns from certain directions instead of using `trim=True` you can use a string to specify top, left, bottom and/or right by using the characters `t`, `l`, `b`, `r`, respectively.

For example, to only trim rows from the bottom and columns from the right you would use `trim='br'`.

Ctrl+Shift+Enter (CSE) Array Functions

Ctrl+Shift+Enter or *CSE* formulas are what Excel used for static array formulas in versions of Excel before *Dynamic Arrays* were added. PyXLL has an *array auto-sizing* feature that can emulate dynamic arrays in earlier versions of Excel that do not implement them.

To enter an array formula in Excel you should do the following:

- Select the range you want the array formula to occupy.
- Enter the formula as normal, but don't press enter.
- Press *Ctrl+Shift+Enter* to enter the formula.

Note that unless you are using *Dynamic Arrays* or PyXLL's *array auto-sizing* feature then if the result is larger than the range you choose then you will only see part of the result. Similarly, if the result is smaller than the selected range you will see errors for the cells with no value.

To make changes to an array formula, change the formula as normal but use *Ctrl+Shift+Enter* to enter the new formula.

Auto Resizing Array Functions

Often selecting a range the exact size of the result of an array formula is not practical. You might not know the size before calling the function, or it may even change when the inputs change.

PyXLL can automatically resize array functions to match the result. To enable this feature you just add 'auto_resize=True' to the options passed to *@xl_func*. For example:

```
from pyxll import xl_func

@xl_func("float[]: float[][]", auto_resize=True)
def diagonal(v):
    d = []
    for i, x in enumerate(v):
        d.append([x if j == i else 0.0 for j in range(len(v))])
    return d
```

You can apply this to all array functions by setting the following option in your pyxll.cfg config file

```
[PYXLL]
;
; Have all array functions resize automatically
;
auto_resize_arrays = 1
```

If you are using a version of Excel that has *Dynamic Arrays* then the auto_resize option will have no effect by default. The native dynamic arrays are superior in most cases, but not yet widely available.

> **Warning**
>
> Auto-resizing is not available for RTD functions. If you are returning an array from an RTD function and need it to resize you can use ref:*Dynamic Arrays <dynamic>* in Excel from Excel 2016 onwards.
>
> If you are not able to update to a newer version of Excel, another solution is to return the array from your RTD function as an object, and then have a second non-RTD function to expand that returned object to an array using PyXLL's auto-resize feature.

Dynamic Array Functions

Dynamic arrays were announced as a new feature of Excel towards the end of 2018. This feature will be rolled out to Office 365 from early 2019. If you are not using Office 365, dynamic arrays are expected to be available in Excel 2022.

If you are not using a version of Excel with the dynamic arrays feature, you can still have array functions that re-size automatically using PyXLL. See *Auto Resizing Array Functions*.

Excel functions written using PyXLL work with the dynamic arrays feature of Excel. If you return an array from a function, it will automatically re-size without you having to do anything extra.

If you are using PyXLL's own *auto resize* feature, PyXLL will detect whether Excel's dynamic arrays are available and if they are it will use those in preference to its own re-sizing. This means that you can write code to work in older versions of Excel that are future-proof and will 'just work' when you upgrade to a newer version of Office.

If you want to keep using PyXLL's *auto resize* feature even when dynamic arrays are available, you can do so by specifying the following in your pyxll.cfg config file

```
[PYXLL]
;
; Use resizing in preference to dynamic arrays
;
allow_auto_resizing_with_dynamic_arrays = 1
```

Dynamic arrays are a great new feature in Excel and offer some advantages over CSE functions and PyXLL's auto-resize feature:

Characteristic	Advantage
Native to Excel	Dynamic arrays are deeply integrated into Excel and so the array resizing works with all array functions, not just ones written with PyXLL.
Spilling	If the results of an array formula would cause data to be over-written you will get a new *#SPILL* error to tell you there was not enough room. When you select the *#SPILL* error Excelwill highlight the spill region in blue so you can see what space it needs.
Referencing the spill range in A1# notation	Dynamic arrays may seamlessly resize as your data changes. When referencing a resizing dynamic arrays you can reference the whole array in a dependable, resilient way by following the cell reference with the # symbol. For example, the reference *A1#* references the entire spilled range for a dynamic array in *A1*.

3.3.6 NumPy Array Types

To be able to use `numpy` arrays you must first have installed the `numpy` package..

You can use `numpy` 1d and 2d arrays as argument types to pass ranges of data into your function, and as return types for returing for array functions. A maximum of two dimensions are supported, as higher dimension arrays don't fit well with how data is arranged in a spreadsheet. You can, however, work with higher-dimensional arrays as *Python objects*.

To specify that a function should accept a numpy array as an argument or as its return type, use the `numpy_array`, `numpy_row` or `numpy_column` types in the `@xl_func` function signature.

These types can be parameterized, meaning you can set some additional options when specifying the type in the function signature.

`numpy_array<dtype=float, ndim=2, casting='unsafe'>`

- `dtype` Data type of the items in the array (e.g. float, int, bool etc.).
- `ndim` Array dimensions, must be 1 or 2.
- `casting` Controls what kind of data casting may occur. Default is 'unsafe'.
  - `'unsafe'` Always convert to chosen dtype. Will fail if any input can't be converted.
  - `'nan'` If an input can't be converted, replace it with NaN.
  - `'no'` Don't do any type conversion.

`numpy_row<dtype=float, casting='unsafe'>`

- `dtype` Data type of the items in the array (e.g. float, int, bool etc.).
- `casting` Controls what kind of data casting may occur. Default is 'unsafe'.
  - `'unsafe'` Always convert to chosen dtype. Will fail if any input can't be converted.
  - `'nan'` If an input can't be converted, replace it with NaN.
  - `'no'` Don't do any type conversion.

`numpy_column<dtype=float, casting='unsafe'>`

- `dtype` Data type of the items in the array (e.g. float, int, bool etc.).
- `casting` Controls what kind of data casting may occur. Default is 'unsafe'.
  - `'unsafe'` Always convert to chosen dtype. Will fail if any input can't be converted.
  - `'nan'` If an input can't be converted, replace it with NaN.
  - `'no'` Don't do any type conversion.

For example, a function accepting two 1d numpy arrays of floats and returning a 2d array would look like:

```
from pyxll import xl_func
import numpy

@xl_func("numpy_array<float, ndim=1> a, numpy_array<float, ndim=1> b: numpy_array
↪<float>")
def numpy_outer(a, b):
    return numpy.outer(a, b)
```

The 'float' dtype isn't strictly necessary as it's the default. If you don't want to set the type parameters in the signature, use the `@xl_arg` and `@xl_return` decorators instead.

PyXLL will automatically resize the range of the array formula to match the returned data if you specify `auto_resize=True` in your py:func:*xl_func* call.

Floating point numpy arrays are the fastest way to get data out of Excel into Python. If you are working on performance sensitive code using a lot of data, try to make use of `numpy_array<float>` or `numpy_array<float, casting='nan'>` for the best performance.

See *Array Functions* for more details about array functions.

3.3.7 Pandas Types

Pandas DataFrames and Series can be used as function arguments and return types for Excel worksheet functions.

For polars DataFrames, see *Polars DataFrames*.

> **Tip**
>
> See *Using Pandas in Excel* for a more complete explaination of how pandas can be used in Excel.

When used as an argument, the range specified in Excel will be converted into a Pandas DataFrame or Series as specified by the function signature.

When returning a DataFrame or Series, a range of data will be returned to Excel. PyXLL will automatically resize the range of the array formula to match the returned data if `auto_resize=True` is set in `@xl_func`.

The following shows returning a random dataframe, including the index:

```
from pyxll import xl_func
import pandas as pd
import numpy as np

@xl_func("int rows, int columns: dataframe<index=True>", auto_resize=True)
def random_dataframe(rows, columns):
    data = np.random.rand(rows, columns)
    column_names = [chr(ord('A') + x) for x in range(columns)]
    return pd.DataFrame(data, columns=column_names)
```

Type Hints

Python type hints can be used instead of the type signature shown above.

Where type parameters are required you can use `typing.Annotated` along with `TypeParameters` (see *Using Type Hints*). Alternatively, you can use `@xl_arg` and `@xl_return` in conjunction with type annotations to specify the type parameters.

The above functions can be written using type hints as follows:

```
from pyxll import xl_func, xl_return
from pyxll import TypeParameters as TP
from typing import Annotated
import pandas as pd
import numpy as np

@xl_func(auto_resize=True)
def random_dataframe(
    rows: int,
    columns: int
) -> Annotated[pd.DataFrame, TP(index=True)]:
    data = np.random.rand(rows, columns)
    column_names = [chr(ord('A') + x) for x in range(columns)]
    return pd.DataFrame(data, columns=column_names)
```

Type Parameters

The following type parameters are available for the `dataframe` and `series` argument and return types:

- **dataframe**, when used as an argument type

  `dataframe<kind=None, index=0, columns=1, dtype=None, dtypes=None, index_dtype=None>`

  kind
  Set to `pandas` to specify that a pandas DataFrame is required. If not set the default DataFrame type will be used.

  The default DataFrame type can be set by the `default_dataframe_kind` option in the `[PYXLL]` section of the pyxll.cfg file.

  Defaults to `pandas` if not set otherwise.

  index
  Number of columns to use as the DataFrame's index. Specifying more than one will result in a DataFrame where the index is a MultiIndex.

  columns
  Number of rows to use as the DataFrame's columns. Specifying more than one will result in a DataFrame where the columns is a MultiIndex. If used in conjunction with *index* then any column headers on the index columns will be used to name the index.

  trim
  New in PyXLL 5.11

  Trim leading and trailing empty rows and columns from the data before constructing the DataFrame.

  See *Trimming DataFrame Arguments*.

  dtype
  Datatype for the values in the dataframe. May not be set with *dtypes*.

  dtypes
  Dictionary of column name -> datatype for the values in the dataframe. May not be set with *dtype*.

  The dictionary can be specified using standard Python dictionary syntax as part of a function signature string. However, often it is more convenient to use the `@xl_arg` or `@xl_return` decorators. These allow you to set type multiple complex parameters more easily, for example:

  ```
  @xl_func
  @xl_arg("df", dtypes={"A": "date"})
  def your_function(df: pd.DataFrame):
      ....
  ```

Not all column dtypes need to be specified. Any that are not specified will default to `var`.

index_dtype
Datatype for the values in the dataframe's index.

multi_sparse[1]
Return sparse results for MultiIndexes. Can be set to `True` or `False`, or `'index'` or `'columns'` if it should only apply to one or the other.

- **dataframe**, when used as a return type

  `dataframe<kind=None, index=None, columns=True>`

  kind
  Set to `pandas` force the kind of DataFrame expected to be a pandas DataFrame.

  If not set, any supported kind of DataFrame can be used.

  index
  If True include the index when returning to Excel, if False don't. If None, only include if the index is named.

  columns
  If True include the column headers, if False don't.

- **series**, when used as an argument type

  `series<index=1, transpose=None, dtype=None, index_dtype=None>`

  index
  Number of columns (or rows, depending on the orientation of the Series) to use as the Series index.

  transpose
  Set to True if the Series is arranged horizontally or False if vertically. By default the orientation will be guessed from the structure of the data.

  dtype
  Datatype for the values in the Series.

  index_dtype
  Datatype for the values in the Series' index.

  multi_sparse[1]
  Return sparse results for MultiIndexes.

- **series**, when used as a return type

  `series<index=True, transpose=False>`

  index
  If True include the index when returning to Excel, if False don't.

  transpose
  Set to True if the Series should be arranged horizontally, or False if vertically.

> **Tip**
>
> For specifying multiple or complex type parameters it can be easier to use the `@xl_arg` and `@xl_return` decorators.
>
> See *@xl_arg and @xl_return Decorators* for more details about how to use `@xl_arg` and `@xl_return` to specify type parameters.

When passing large DataFrames between Python functions, it is not always necessary to return the full DataFrame to Excel and it can be expensive reconstructing the DataFrame from the Excel range each time. In those cases

[1] The `multi_sparse` parameter is new in PyXLL 5.3.0.

you can use the `object` return type to return a handle to the Python object. Functions taking the `dataframe` and `series` types can accept object handles.

See *Using Pandas in Excel* for more information.

3.3.8 Polars DataFrames

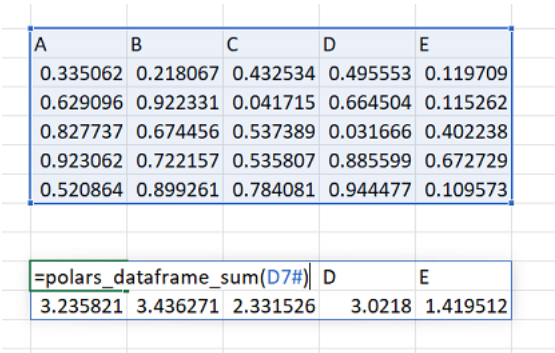
New in PyXLL 5.6

Polars DataFrames can be used as function arguments and return types for Excel worksheet functions in a similar way to *Pandas DataFrames*.

When used as an argument, the range specified in Excel will be converted into a Polars DataFrame as specified by the function signature.

The `dataframe` type specifier is the same as the one used for pandas DataFrames. When passing a polars DataFrame as an argument you need to specify the kind of DataFrame you want by setting `kind=polars`:

```
from pyxll import xl_func

@xl_func("dataframe<kind=polars>: dataframe", auto_resize=True)
def polars_dataframe_sum(df):
    # df is a polars.DataFrame
    return df.sum()
```



A	B	C	D	E
0.335062	0.218067	0.432534	0.495553	0.119709
0.629096	0.922331	0.041715	0.664504	0.115262
0.827737	0.674456	0.537389	0.031666	0.402238
0.923062	0.722157	0.535807	0.885599	0.672729
0.520864	0.899261	0.784081	0.944477	0.109573

=polars_dataframe_sum(D7#)			D	E
3.235821	3.436271	2.331526	3.0218	1.419512

When returning a polars DataFrame there is no need to specify the kind as that will be infered from the type of the returned object.

The following shows returning a random polars DataFrame:

```
from pyxll import xl_func
import polars as pl
import numpy as np

@xl_func("int rows, int columns: dataframe", auto_resize=True)
def random_polars_dataframe(rows, columns):
    data = np.random.rand(columns, rows)
    column_names = [chr(ord('A') + x) for x in range(columns)]
    return pl.DataFrame(data, columns=column_names)
```

> **Tip**
>
> The type alias `polars.dataframe` can be used in place of `dataframe<kind="polars">`, and `pandas.dataframe` can be used in place of `dataframe<kind="pandas">`. Both forms are equivalent.

Type Hints

Python type hints can be used instead of the type signature shown above.

Where type parameters are required you can use `typing.Annotated` along with *`TypeParameters`* (see *Using Type Hints*). Alternatively, you can use *`@xl_arg`* and *`@xl_return`* in conjunction with type annotations to specify the type parameters.

The above functions can be written using type hints as follows:

```
from pyxll import xl_func
import polars as pl
import numpy as np

@xl_func(auto_resize=True)
def random_polars_dataframe(rows: int, columns: int) -> pl.DataFrame:
    data = np.random.rand(columns, rows)
    column_names = [chr(ord('A') + x) for x in range(columns)]
    return pl.DataFrame(data, columns=column_names)

@xl_func(auto_resize=True)
def polars_dataframe_sum(df: pl.DataFrame) -> pl.DataFrame:
    # df is a polars.DataFrame
    return df.sum()
```

Type Parameters

The following type parameters are available for `dataframe` argument and return types:

- **dataframe**, when used as an argument type

  `dataframe<kind=None, dtype=None, dtypes=None>`

  `kind`
  Set to `polars` to specify that a polars DataFrame is required. If not set the default DataFrame type will be used.

  The default DataFrame type can be set by the `default_dataframe_kind` option in the `[PYXLL]` section of the pyxll.cfg file.

  Defaults to `pandas` if not set otherwise.

  `trim`
  New in PyXLL 5.11

  Trim leading and trailing empty rows and columns from the data before constructing the DataFrame.

  See *Trimming DataFrame Arguments*.

  `dtype`
  Datatype for the values in the dataframe. May not be set with *dtypes*.

  `dtypes`

  Dictionary of column name -> datatype for the values in the dataframe. May not be set with *dtype*.

The dictionary can be specified using standard Python dictionary syntax as part of a function signature string. However, often it is more convenient to use the *@xl_arg* or *@xl_return* decorators. These allow you to set type multiple complex parameters more easily, for example:

```
@xl_func
@xl_arg("df", dtypes={"A": "date"})
def your_function(df: pl.DataFrame):
    ....
```

Not all column dtypes need to be specified. Any that are not specified will default to `var`.

strict

New in PyXLL 5.12

Throw an error if any data value does not exactly match the given or inferred data type for that column. If set to False, values that do not match the data type are cast to that data type or, if casting is not possible, set to null instead.

Defaults to `True`.

- **dataframe**, when used as a return type

  `dataframe<kind=None>`

  kind

  Set to `polars` force the kind of DataFrame expected to be a polars DataFrame.

  If not set, any supported kind of DataFrame can be used.

> **Tip**
>
> For specifying multiple or complex type parameters it can be easier to use the *@xl_arg* and *@xl_return* decorators.
>
> See *@xl_arg and @xl_return Decorators* for more details about how to use *@xl_arg* and *@xl_return* to specify type parameters.

When passing large DataFrames between Python functions it is not always necessary, or desirable, to return the full DataFrame to Excel. It can be expensive reconstructing the DataFrame from the Excel range each time. In those cases you can use the `object` return type to return a handle to the Python object. Functions taking `dataframe<kind=polars>` can accept object handles.

3.3.9 Asynchronous Functions

- *Asynchronous Worksheet Functions*
- *The asyncio Event Loop*
- *Before Python 3.5*

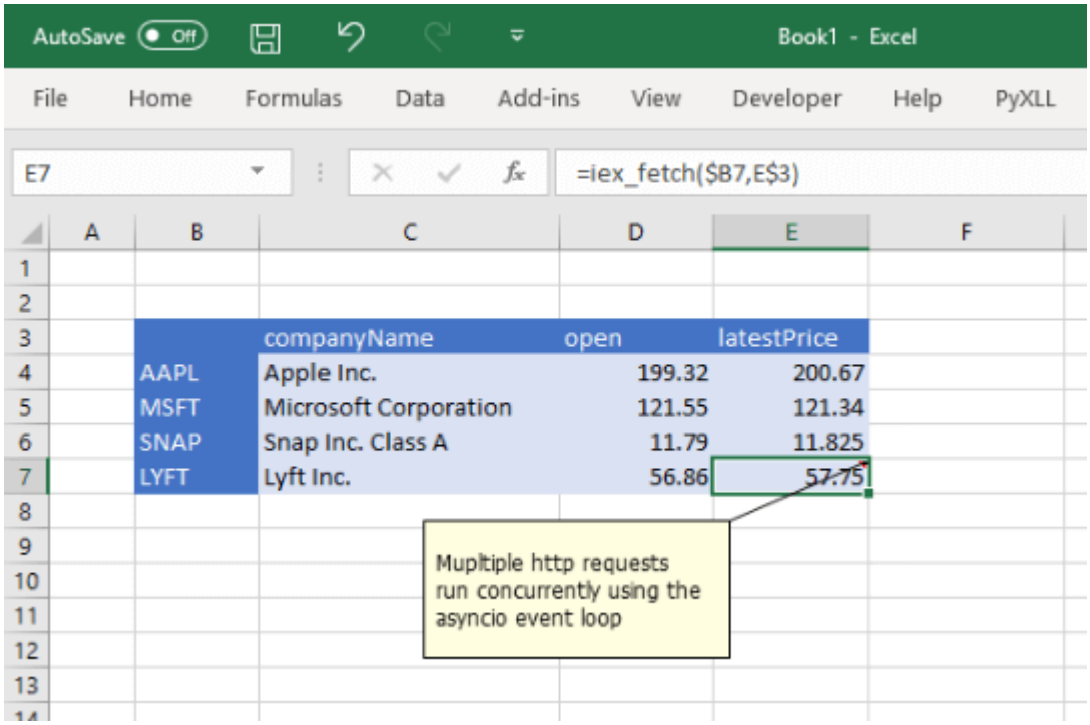
Excel has supported asynchronous worksheet functions since Office 2010. To be able to use asynchronous worksheet functions with PyXLL you will need to be using at least that version of Office.

Excel asynchronous worksheet functions are called as part of Excel's calculation in the same way as other functions, but rather than return a result, they can schedule some work and return immediately, allowing Excel's calculation to progress while the scheduled work for the asynchronous function continues concurrently. When the asynchronous work has completed, Excel is notified.

Asynchronous functions still must be completed as part of Excel's normal calculation phase. Using asynchronous functions means that many more functions can be run concurrently, but Excel will still show as calculating until all asynchronous functions have returned.

Functions that use IO, such as requesting results from a database or web server, are well suited to being made into asynchronous functions. For CPU intensive tasks[1] using the *thread_safe* option to `@xl_func` may be a better alternative.

If your requirement is to return the result of a very long running function back to Excel after recalculating has completed, you may want to consider using an RTD (*Real Time Data*) function instead. An RTD function doesn't have to keep updating Excel, it can just notify Excel once when a single calculation is complete. Also, it can be used to notify the user of progress which for very long running tasks can be helpful.



Note

If Excel is interupted during an async calculation it will return control of Excel to use the user.

When calculating, Excel will appear blocked but if the user clicks somewhere or presses a button on the keyboard Excel will interupt the calculation and return control to Excel.

If there are any asynchronous functions that have not yet completed, after a short time Excel will resume calculating.

New in PyXLL 5.9.0

Excel's async handles (how the result is returned back to Excel) are only valid for the calculation phase they were started in. Attempting to return a result after Excel's calculation has completed (or been interupted) results in an error.

As of PyXLL 5.9.0, PyXLL will manage the async handles so that any async functions started during on calculation phase can still return their result in the next calculation phase. This means that if Excel is interupted before an async function completes then it will continue and the result will be returned to Excel when it resumes calculating.

This behaviour can be disabled to give the same behaviour as earlier PyXLL versions by setting the following in your pyxll.cfg file:

```
[PYXLL]
disable_async_cache = 1
```

[1] For CPU intensive problems that can be solved using multiple threads (i.e. the CPU intensive part is done without the Python Global Interpreter Lock, or GIL, being held) use the `thread_safe` argument to `@xl_func` to have Excel automatically schedule your functions using a thread pool.

Asynchronous Worksheet Functions

Python 3.5 Required

Using the **async** keyword requires a minimum of Python 3.5.1 and PyXLL 4.2. If you do not have these minimum requirements see *Before Python 3.5*.

If you are using a modern version of Python, version 3.5.1 or higher, writing asynchronous Excel worksheet functions is as simple as adding the `async` keyword to your function definition. For earlier versions of Python, or for PyXLL versions before 4.2, or if you just don't want to use coroutines, see *Before Python 3.5*.

The following example shows how the asynchronous http package `aiohttp` can be used with PyXLL to fetch stock prices *without* blocking the Excel's calculation while it waits for a response[2]

```
from pyxll import xl_func
import aiohttp
import json

endpoint = "https://api.iextrading.com/1.0/"

@xl_func
async def iex_fetch(symbol, key):
    """returns a value for a symbol from iextrading.com"""
    url = endpoint + f"stock/{symbol}/batch?types=quote"
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            assert response.status == 200
            data = await response.read()

    data = json.loads(data)["quote"]
    return data.get(key, "#NoData")
```

The function above is marked `async`. In Python, as async function like this is called a *coroutine*. When the coroutine decorated with `@xl_func` is called from Excel, PyXLL schedules it to run on an *asyncio event loop*.

The coroutine uses `await` when calling `response.read()` which causes it to yield to the asyncio event loop while waiting for results from the server. This allows other coroutines to continue rather than blocking the event loop.

Note that if you do not yield to the event loop while waiting for IO or another request to complete, you will be blocking the event loop and so preventing other coroutines from running.

If you are not already familiar with how the `async` and `await` keywords work in Python, we recommend you read the following sections of the Python documentation:

- Coroutines and Tasks
- asyncio — Asynchronous I/O

Warning

Async functions cannot be automatically resized using the "auto_resize" parameter to `@xl_func`. If you need to return an array using an async function and have it be resized, it is recommended to return the array from the async function as an *object* by specifying *object* as the return type of your function, and then use a second non-async function to expand the array.

For example:

[2] Asynchronous functions are only available in Excel 2010. Attempting to use them in an earlier version will result in an error.

```
@xl_func("var x: object")
async def async_array_function(x):
    # do some work that creates an array
    return array

@xl_func("object: var", auto_resize=True)
def expand_array(array):
    # no need to do anything here, PyXLL will do the conversion
    return array
```

The asyncio Event Loop

Using the `asyncio` event loop with PyXLL requires a minimum of Python 3.5.1 and PyXLL 4.2. If you do not have these minimum requirements see *Before Python 3.5*.

When a coroutine (`async` function) is called from Excel, it is scheduled on the *asyncio event loop*. PyXLL starts this event loop on demand, the first time an asynchronous function is called.

For most cases, PyXLL default asyncio event loop is well suited. However the event loop that PyXLL uses can be replaced by setting `start_event_loop` and `stop_event_loop` in the `PYXLL` section of the pyxll.cfg file. See *PyXLL Settings* for more details.

To schedule tasks on the event loop outside of an asynchronous function, the utility function `get_event_loop` can be used. This will create and start the event loop, if it's not already started, and return it.

By default, the event loop runs on a single background thread. To schedule a function it is therefore recommended to use `loop.call_soon_threadsafe`, or `loop.create_task` to schedule a coroutine.

Before Python 3.5

Or with Python >= 3.5…

Everything in this section still works with Python 3.5 onwards.

If you are using an older version of Python than 3.5.1, of if you have not yet upgraded to PyXLL 4.2 or later, you can still use asynchronous worksheet functions but you will not be able to use the `async` keyword to do so.

Asynchronous worksheet functions are declared in the same way as regular worksheet functions by using the `@xl_func` decorator, but with one difference. To be recognised as an asynchronous worksheet function, one of the function argument must be of the type `async_handle`.

The `async_handle` parameter will be a unique handle for that function call, represented by the class `XLAsyncHandle` and it must be used to return the result when it's ready. A value must be returned to Excel using `xlAsyncReturn` or (new in PyXLL 4.2) the methods `XLAsyncHandle.set_value` and `XLAsyncHandle.set_error`. Asynchronous functions themselves should not return a value.

The `XLAsyncHandle` instance is only valid during the worksheet recalculation cycle in which that the function was called. If the worksheet calculation is cancelled or interrupted then calling `xlAsyncReturn` with an expired handle will fail. For example, when a worksheet calculated (by pressing F9, or in response to a cell being updated if automatic calculation is enabled) and some asynchronous calculations are invoked, if the user interrupts the calculation before those asynchronous calculations complete then calling `xlAsyncReturn` after the worksheet calculation has stopped will result in a exception being raised.

For long running calculations that need to pass results back to Excel after the sheet recalculation is complete you should use a *Real Time Data* function.

Here's an example of an asynchronous function[Page 87, 2]

```
from pyxll import xl_func, xlAsyncReturn
from threading import Thread
```

(continues on next page)

(continued from previous page)

```
import time
import sys

class MyThread(Thread):
    def __init__(self, async_handle, x):
        Thread.__init__(self)
        self.__async_handle = async_handle
        self.__x = x

    def run(self):
        try:
            # here would be your call to a remote server or something like that
            time.sleep(5)
            xlAsyncReturn(self.__async_handle, self.__x)
        except:
            self.__async_handle.set_error(*sys.exc_info())  # New in PyXLL 4.2


# no return type required as Excel async functions don't return a value
# the excel function will just take x, the async_handle is added automatically by␣
↪Excel
@xl_func("async_handle<int> h, int x")
def my_async_function(h, x):
    # start the request in another thread (note that starting hundreds of threads isn
↪'t advisable
    # and for more complex cases you may wish to use a thread pool or another␣
↪strategy)
    thread = MyThread(h, x)
    thread.start()

    # return immediately, the real result will be returned by the thread function
    return
```

The type parameter to `async_handle` (e.g. `async_handle<date>`) is optional. When provided, it is used to convert the value returned via *`xlAsyncReturn`* to an Excel value. If omitted, the `var` type is used.

3.3.10 Function Documentation

- *Writing Function Documentation*
  - *Google Style*
  - *NumPy Style*
  - *Choosing Which Docstring Styles Are Detected*
- *Linking to Additional Documentation*
  - *Linking to a URL (website)*
  - *Using a Compiled Help File*
- *IntelliSense (auto-completion)*

Writing Function Documentation

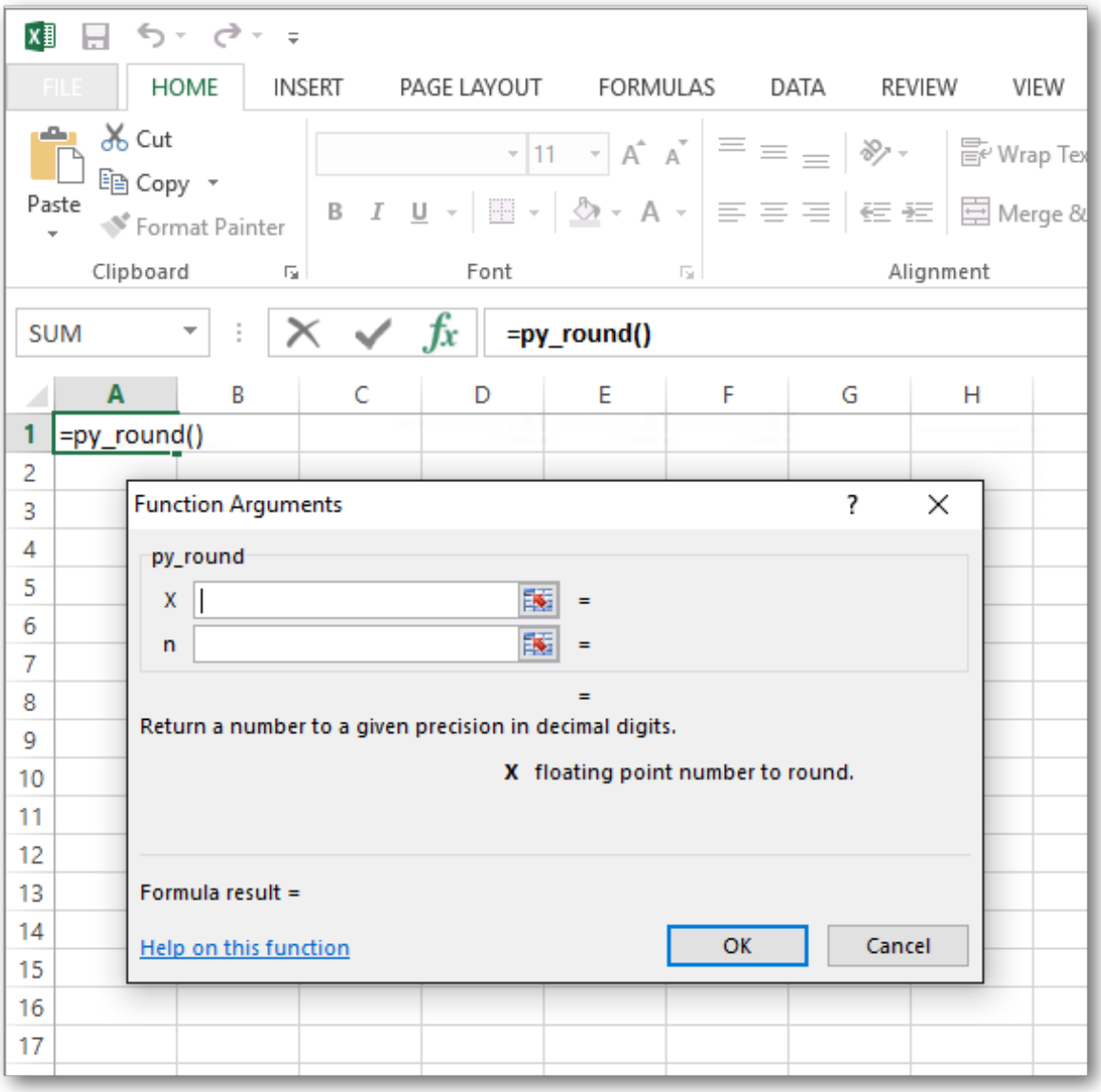
When a python function is exposed to Excel with the `@xl_func` decorator the docstring of that function is visible in Excel's function wizard dialog.

Parameter documentation may also be provided help the user know how to call the function. The most convenient way to add parameter documentation is to add it to the docstring as shown in the following example:

```
from pyxll import xl_func

@xl_func
def py_round(x, n):
    """
    Return a number to a given precision in decimal digits.

    :param x: floating point number to round
    :param n: number of decimal digits
    """
    return round(x, n)
```

PyXLL automatically detects parameter documentation written in the commonly used Sphinx style shown above, as well as Google style and NumPy style docstrings (**new in PyXLL 5.13**). Whichever style is used, the parameter descriptions are shown as help strings for the parameters when selected in Excel's function wizard, and the first line of the docstring is used as the function description.

The arguments and documentation provided are displayed in Excel's function wizard:



Parameter documentation may also be added by passing a dictionary of parameter names to help strings to *@xl_func* as the keyword argument `arg_descriptions` if it is not desirable to add it to the docstring for any reason.

Google Style

New in PyXLL 5.13

Parameters documented in an `Args:` (or `Arguments:/Parameters:/Params:`) section are also picked up:

```
from pyxll import xl_func

@xl_func
def py_round(x, n):
    """Return a number to a given precision in decimal digits.

    Args:
        x: floating point number to round
        n: number of decimal digits
    """
```

(continues on next page)

(continued from previous page)

```
    return round(x, n)
```

NumPy Style

New in PyXLL 5.13

Parameters documented in a NumPy-style `Parameters` section are also picked up:

```
from pyxll import xl_func

@xl_func
def py_round(x, n):
    """Return a number to a given precision in decimal digits.

    Parameters
    ----------
    x : float
        floating point number to round
    n : int
        number of decimal digits
    """
    return round(x, n)
```

A single description can be shared by more than one parameter by listing the comma separated parameter names on the same line, e.g. `x, n : float`.

Choosing Which Docstring Styles Are Detected

New in PyXLL 5.13

By default reST, Google and NumPy style docstrings are all auto-detected, and PyXLL uses whichever style it finds first in the docstring. If this detection ever misfires for one of your functions - for example a reST docstring that also happens to contain a line that looks like a NumPy/Google section header - the set of styles considered can be restricted using the `docstring_styles` setting in the `[PYXLL]` section of the *config file*:

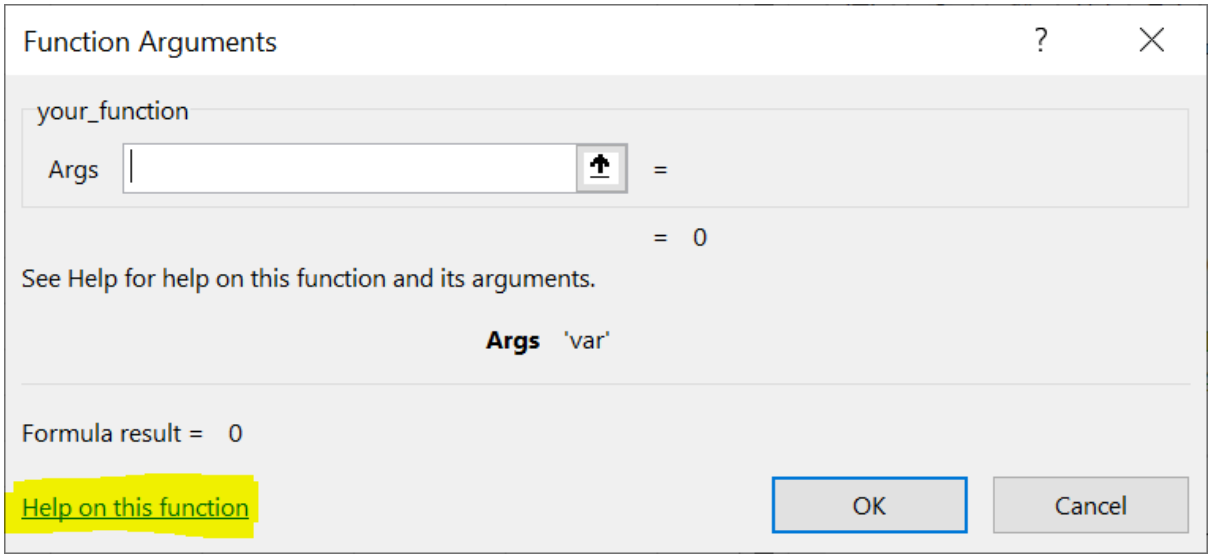
```
[PYXLL]
; Only detect reST/Sphinx style docstrings
docstring_styles = rst
```

Multiple styles may be listed, separated by commas or on separate lines, e.g. `docstring_styles = rst, numpy`. Valid values are `rst`, `google` and `numpy`.

Linking to Additional Documentation

In the Excel function wizard screen there is a "Help on this function" link. You can use this link to reference any URL or CHM (Compiled HTML File) containing your more detailed documentation.

The link to use for the "Help on this function" link is set using the `help_topic` keyword argument to the *`@xl_func`* decorator.



Linking to a URL (website)

To set the help topic for a function to a URL pass it as the `help_topic` keyword argument to the *@xl_func* decorator.

You must use the full URL including the "http://", "https://", or "file://" prefix. You can include any query string or anchors that you need in the URL. For example:

```
from pyxll import xl_func

@xl_func(help_topic="https://www.google.com/search?q=pyxll")
def your_function(...):
    ....
```

When you examine this function in the Excel Function Wizard it will now have a link to the URL specified.

> **Note**
>
> Linking to URLs was added in PyXLL 5.1. In earlier versions only linking to CHM files is possible (see below).

Using a Compiled Help File

CHM, or HTML Help, files are compiled from HPP and HTML files using tools like Microsoft HTML Html (https://docs.microsoft.com/en-us/previous-versions/windows/desktop/htmlhelp/microsoft-html-help-downloads) or other third party tools.

To reference a CHM file for a specific function, when registering the function with :py:deco`xl_func` set the `help_topic` kwarg to the absolute path of the .chm file to load, and the section of the file to open as a numeric *help context id* in the form "filename.chm!HelpContextID", e.g. "C:/folder/help.chm!0".

If you only want to open the chm file, you still have to provide a help context id but it can be 0.

The file path should be an absolute path and must be less than 255 characters long, so you may have to use a windows short path (this is a limitation of Excel).

To add context ids to your chm file you have to add aliases to your hpp file used to make the chm file, e.g.

```
---- MYHELP.HHP ----
[FILES]
index.html
myfunc1.html
```

(continues on next page)

(continued from previous page)

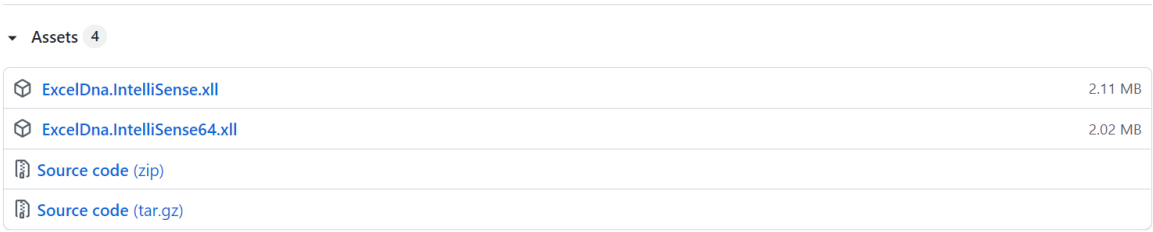
```
myfunc2.html
myfunc3.html

[ALIAS]
IDH_topic_1 = myfunc1.html

[MAP]
#define IDH_topic_1 1001
-------------------------
```

IntelliSense (auto-completion)

Excel doesn't natively provide IntelliSense or auto-completion for functions provided by add-ins such as PyXLL.

The third party add-in Excel-DNA IntelliSense implements IntelliSense-like auto-completion for Excel add-ins, and this is the recommended way to get auto-completion using PyXLL.

To install the Excel-DNA IntelliSense add-in, download it from https://github.com/Excel-DNA/IntelliSense/releases.



Once you have installed the ExcelDNA-IntelliSense add-in, PyXLL will detect it automatically and start using it to give you auto-completion and in-sheet IntelliSense for your PyXLL functions. The same function documentation as shown in the Excel function wizard is used.

If you need help installing the Excel-DNA IntelliSense add-in then this video will help https://www.youtube.com/watch?v=-BIOzl1igXU.

> **Note**
>
> The Excel-DNA IntelliSense add-in is not part of PyXLL, and is not supported or maintained by PyXLL. It is referenced here for information only. If you need assistance using this add-in you should contact the ExcelDNA team via their github page.

3.3.11 Variable and Keyword Arguments

- *Variable Arguments (*args)*
- *Keyword Arguments (**kwargs)*

Variable Arguments (*args)

In Python it is possible to declare a function that takes a variable number of positional arguments using the special **args* notation. These functions can be exposed to Excel as worksheet functions that also take a variable number of arguments.

The function shown below uses the first argument as a separator ond returns a string made up of the string values of all other arguments separated by the separator.

```
from pyxll import xl_func

@xl_func
def py_join(sep, *args):
    """Joins a number of args with a separator"""
    return sep.join(map(str, args))
```

You can also set the type of the args in the function signature. When doing that the type for all of the variable arguments must be the same. For mixed types, use the `var` type.

```
from pyxll import xl_func

@xl_func("str sep, str *args: str")
def py_join(sep, *args):
    """Joins a number of args with a separator"""
    return sep.join(args)
```

Unlike Python, Excel has some limits on the number of arguments that can be provided to a function. For practical purposes the limit is high enough that it is unlikely to be a problem. The absolute limit for the number of arguments is 255, however the actual limit for a function may be very slightly lower[1].

Keyword Arguments (**kwargs)

New in PyXLL 5.8

Python functions can take an arbitrary number of named arguments using the special ***kwargs* argument.

A keyword argument is where you pass an argument to a function by name. When using ***kwargs*, any named argument that is not one of the function's parameter names is added to a dictionary and passed to the function in the *kwargs* dictionary.

Excel does not support passing arguments by name when calling Excel worksheet functions (UDFs).

When registering a Python function to be called from Excel that has ***kwargs* the Excel function will expect a 2d array of values for the *kwargs* argument. PyXLL will convert that to a dictionary and pass that dictionary to the Python function as the ***kwargs*.

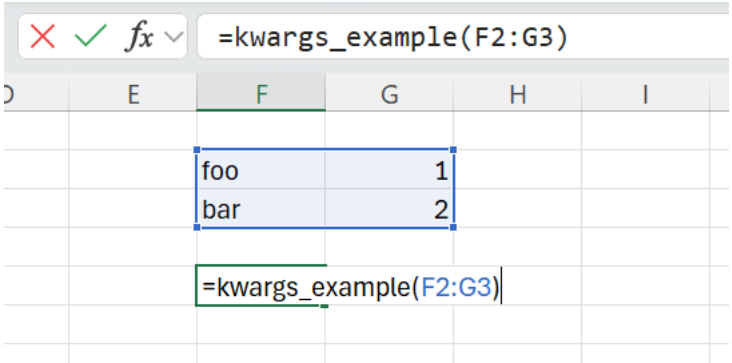
The array passed from Excel to the Python function should be a list of *key, value* pairs.

For example, the following Python function takes ***kwargs* and can be called from Excel passing a range of *key, value* pairs:

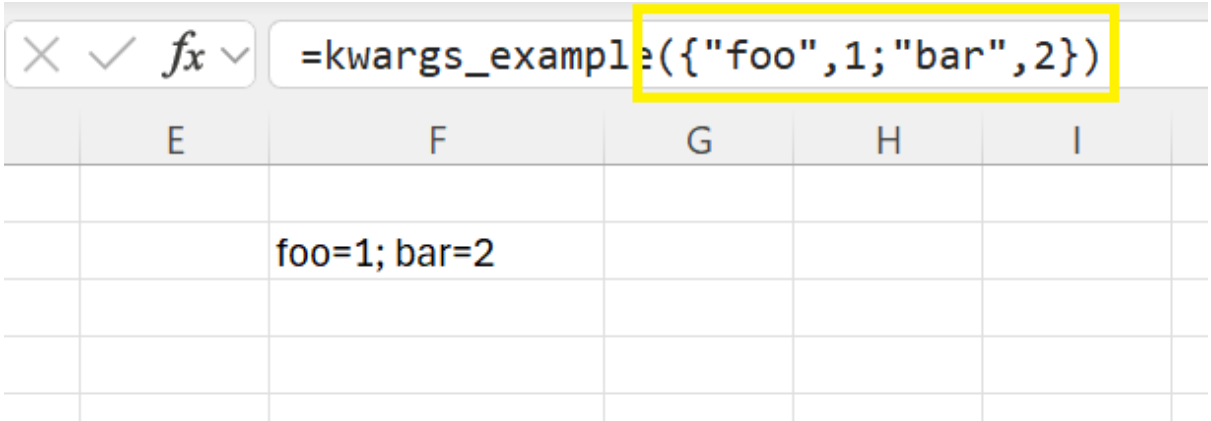
```
from pyxll import xl_func

@xl_func
def kwargs_example(**kwargs):
    return "foo={foo}; bar={bar}".format(**kwargs)
```

The function can be called from Excel, passing the ***kwargs* as an array of key value pairs:

[1] The technical reason this limit is lower is because when the function is registered with Excel, a string is used to tell Excel all the argument and return types, as well as any modifiers for things like whether the function is thread safe or not. The total length of this string cannot exceed 255 characters so, even though Excel might be able to handle 255 arguments, it may not be possible to register a function with 255 arguments because of the length limit on that string.



Arrays can be passed directly to Excel functions, without needing to refer to a range on a worksheet. The syntax for this varies depending on your language settings. In English, the format to pass an array is `{ "key1", value1; "key2", value2; "keyN", valueN }`. `,` is the *item* separator, and `;` is the *row* separator. In some languages the item and row separators are reversed.



Specifying types for ***kwargs* is done in the same way as for dictionay types. See *Dictionary Types* for full details of how to specify types for dictionaries.

If using Python 3.12 or higher, you can also use the standard `TypedDict` type annotation for ***kwargs*.

```
from typing import TypedDict
from pyxll import xl_func

class MyTypedKwargs(TypedDict):
  foo: int
  bar: int

@xl_func
def kwargs_example(**kwargs: MyTypedKwargs) -> str:
    return "foo={foo}; bar={bar}".format(**kwargs)
```

> **Note**
>
> It is not possible to combine **args* and ***kwargs* for functions that are exposed to Excel. This is due to the constraints of how Excel functions can be called.

3.3.12 Recalculating On Open

It can be useful to have worksheet functions (UDFs) that are automatically called when a workbook is opened. Typically this is achieved by making the function volatile, but making a function volatile means that it is recalculated every time Excel calculates and not just when the workbook is opened.

PyXLL functions can be made to automatically recalculate when a workbook is opened by passing `recalc_on_open=True` to *@xl_func*.

The recalculating on open feature makes use of *Workbook Metadata*.

- *Use-Cases*
- *Example*
- *Default Behaviour*
- *Disabling Completely*

Use-Cases

Use-cases for wanting to recalculate a function when a workbook is opened include:

- Opening a database connection to be used by other functions.
- Loading market data or other data required by the workbook.
- Values that depend on the current date or time, but should not constantly update.
- Functions returning objects, as the object needs to be created in order for the sheet to calculate.
- RTD functions, as they need to be called once to start ticking.

Example

The below function uses the `recalc_on_open=True` option to tell PyXLL that it should be recalculated when the saved workbook is opened. For this to work, this function is called from a cell (eg `=recalc_on_open_func()`) and then the workbook is saved. The next time that workbook is opened, the cell containing the function will be marked as dirty and if automatic calculations are enabled it will be recalculated.

```
from pyxll import xl_func

@xl_func(recalc_on_open=True)
def recalc_on_open_func():
    print("recalc_on_open_func called!")
    return "OK!"
```

Warning

When a workbook is opened cells in that workbook will be recalculated if the function in that cell was marked as needing to be recalculated on open *at the time the workbook was saved*.

Changing the `recalc_on_open` option for a function *after* the workbook has been saved will have no effect until the workbook has been recalculated and saved again.

Similarly, if opening a workbook saved with a version of PyXLL prior to 4.5, the workbook will need to be recalculated and saved before it will recalculate on being opened.

Default Behaviour

The default behaviour for non-volatile worksheet functions is not to recalculate on open unless the `recalc_on_open` option is set in *@xl_func*.

The `recalc_on_open` feature is especially useful for RTD functions and for functions returning Python objects. The default behaviour for these two types of functions can be modified such that the `recalc_on_open` feature applies by default.

- **Real Time Data (RTD) Functions**

  For RTD functions the option `recalc_rtd_on_open` can be set in the `PYXLL` section of the pyxll.cfg config file. If set, all RTD functions will recalculate on opening unless specifically disabled by setting `recalc_on_open=False` in the *@xl_func* decorator.

```
[PYXLL]
; Enable recalc on open for all RTD functions
recalc_rtd_on_open = 1
```

- **Functions Returning Objects**

  Similarly, any worksheet function that explicitly returns an `object` can be set to recalculate on opening via the config setting `recalc_cached_objects_on_open`.

  To enable recalculating all object functions on open set `recalc_cached_objects_on_open` to 1.

```
[PYXLL]
; Eenable recalc on open for all functions returning objects
recalc_cached_objects_on_open = 1
```

  With this setting enabled the following function would be recalculated when a workbook using it was opened, without needing to explicitly set `recalc_on_open=True` in *@xl_func*.

```
from pyxll import xl_func

@xl_func("int x: object")
def create_object(x)
    obj = SomeClass(x)
    return obj
```

  This can be overridden for individual functions by passing `recalc_on_open=False` to *@xl_func*.

As with the `recalc_on_open` setting, these settings only affect what metadata gets saved in the workbook. Changing the `recalc_cached_objects_on_open` option after the workbook has been saved will have no effect until the workbook has been recalculated and saved again.

Disabling Completely

If you do not want any functions to be recalculated when opening a workbook set `disable_recalc_on_open = 1` in your pyxll.cfg file.

This setting prevents any cells marked by PyXLL as needing to be recalculated from being recalculated, regardless of what settings were used at the time the file was saved. It does not prevent Excel from calculating other cells that need recalculating, such as volatile cells.

```
[PYXLL]
disable_recalc_on_open = 1
```

3.3.13 Recalculating On Reload

New in PyXLL 5.10

It can be useful to have worksheet functions (UDFs) that are automatically called whenevner the PyXLL add-in is reloaded.

One instance where this is useful is for functions returning objects. When reloading the PyXLL add-in any cached objects that have been returned from functions previously are cleared (unless the `clear_object_cache_on_reload = 0` option is used). These functions need to be recalculated to recreate those objects, and so recalculating on reload can be helpful in that instance.

PyXLL functions can be made to automatically recalculate after the PyXLL add-in is reloaded by passing `recalc_on_reload=True` to *@xl_func*.

- *Example*
- *Default Behaviour*
- *Disabling Completely*

Example

The below function uses the `recalc_on_reload=True` option to tell PyXLL that it should be recalculated whenever the add-in is reloaded.

This function must be called at least once first (which happened either when entering the formula, or recalculating the workbook). Then, the next time the add-in is reloaded the cell containing the function will be marked as dirty and if automatic calculations are enabled it will be recalculated.

```
from pyxll import xl_func

@xl_func(recalc_on_reload=True)
def recalc_on_reload_func():
    print("recalc_on_reload_func called!")
    return "OK!"
```

Default Behaviour

The default behaviour for non-volatile worksheet functions is not to recalculate on reload unless the `recalc_on_reload` option is set in *`@xl_func`*.

The `recalc_on_reload` feature is especially useful for RTD functions and for functions returning Python objects. The default behaviour for these two types of functions can be modified such that the `recalc_on_reload` feature applies by default.

- **Real Time Data (RTD) Functions**

  For RTD functions the option `recalc_rtd_on_reload` can be set in the `PYXLL` section of the pyxll.cfg config file. If set, all RTD functions will recalculate on reloading unless specifically disabled by setting `recalc_on_reload=False` in the *`@xl_func`* decorator.

  ```
  [PYXLL]
  ; Enable recalc on reload for all RTD functions
  recalc_rtd_on_reload = 1
  ```

- **Functions Returning Objects**

  Similarly, any worksheet function that explicitly returns an `object` can be set to recalculate on reloading via the config setting `recalc_cached_objects_on_reload`.

  To enable recalculating all object functions on reload set `recalc_cached_objects_on_reload` to 1.

  ```
  [PYXLL]
  ; Eenable recalc on reload for all functions returning objects
  recalc_cached_objects_on_reload = 1
  ```

  With this setting enabled the following function would be recalculated when a workbook using it was opened, without needing to explicitly set `recalc_on_reload=True` in *`@xl_func`*.

  ```
  from pyxll import xl_func

  @xl_func("int x: object")
  def create_object(x)
  ```

(continues on next page)

(continued from previous page)

```
    obj = SomeClass(x)
    return obj
```

This can be overridden for individual functions by passing `recalc_on_reload=False` to *@xl_func*.

Disabling Completely

If you do not want any functions to be recalculated when reloading set `disable_recalc_on_reload = 1` in your pyxll.cfg file.

This setting prevents any cells marked by PyXLL as needing to be recalculated when reloading the PyXLL add-in, regardles of any other settings or the `recalc_on_reload` option to *@xl_func*.

```
[PYXLL]
disable_recalc_on_reload = 1
```

3.3.14 Interrupting Functions

Long running functions can cause Excel to become unresponsive and sometimes it's desirable to allow the user to interrupt functions before they are complete.

Excel allows the user to signal they want to interrupt any currently running functions by pressing the *Esc* key. If a Python function has been registered with *allow_abort=True* (see *@xl_func*) PyXLL will raise a `KeyboardInterrupt` exception if the user presses *Esc* during execution of the function.

This will usually cause the function to exit, but if the `KeyboardInterrupt` exception is caught then notrmal Python exception handling takes place. Also, as it is a Python exception that's raised, if the Python function is calling out to something else (e.g. a C extension library) the exception may not be registered until control is returned to Python.

Enabling *allow_abort* registers a Python trace function for the duration of the call to the function. This can have a negative impact on performance and so it may not be suitable for all functions. The Python interpreter calls the trace function very frequently, and PyXLL checks Excel's abort status during this trace function. To reduce the performance overhead of calling this trace function, PyXLL throttles how often it checks Excel's abort status and this throttling can be fine-tuned with the config settings *abort_throttle_time* and *abort_throttle_count*. See *PyXLL Settings* for more details.

The *allow_abort* feature can be enabled for all functions by setting it in the configuration. *This feature should be used with caution because of the performance implications outlined above*.

```
[PYXLL]
;
; Make all Excel UDFs inherently interruptable
;
allow_abort = 1
```

It is not enabled by default because of the performance impact, and also because it can interfere with the operation of some remote debugging tools that use the same Python trace mechanism.

3.4 Macro Functions

3.4.1 Introduction

You can write an Excel macro (sometimes called 'Subs' in VBA) in Python. A Python macro function can do whatever you would previously have used VBA for. Macros work in a very similar way to worksheet functions.

Macro functions are almost identical to worksheet functions in how they're written. One key difference, however, is that macro functions called by Excel can use the Excel Object Model to automate or script Excel.

PyXLL macro functions written in Python are exactly the same as any other Excel macro function. They can be called from form items such as buttons, or from VBA.

To register a function as a macro you use the `@xl_macro` decorator, which works in almost exactly the same way as the `@xl_func` decorator but is used for macros instead of worksheet functions.

VBA macros are often used to automate tasks in Excel, but using PyXLL we can write equivalent macros entirely in Python without the need for any VBA. The function `xl_app` can be used to get the *Excel.Application* COM object (using either `win32com` or `comtypes`), which is the COM object corresponding to the *Application* object in VBA.

See *Python as a VBA Replacement* for details of how the Excel Object Model can be used from Python.

Writing an Excel Macro Function in Python

If you've not installed the PyXLL addin yet, see *Installing PyXLL*.

To tell the PyXLL add-in to expose a Python function so that we can call it from Excel as a macro, all that is needed is to add the `@xl_macro` decorator to a Python function:

```
from pyxll import xl_macro

@xl_macro
def hello(name):
    return "Hello, %s" % name
```

This function takes just a single argument, `name`, which can be passed in when we call the function from Excel.

PyXLL supports passing arguments and returning values of many different types, which is covered in detail in the *Argument and Return Types* section.

Configuring PyXLL with your Python Module

Once you have saved that code you need to ensure the interpreter can find it by modifying the following settings in your `pyxll.cfg` config file:

- `[PYXLL]` / `modules`

  The list of Python modules that PyXLL will import.

- `[PYTHON]` / `pythonpath`

  The list of folders that Python will look for modules in.

If you saved the above code into a new file called *my_module.py* in a folder *C:\Users\pyxll\modules* you would add the Python module *my_module* to the `modules` list, and *C:\Users\pyxll\modules* to the `pythonpath`.

Note that Python module *file names* end in *.py*, but the Python *module names* do not.

```
[PYXLL]
;
; Make sure that PyXLL imports the module when loaded.
;
; We use the module name here, not the file name,
; and so the ".py" file extension is omitted.
;
modules = my_module

[PYTHON]
;
; Ensure that PyXLL can find the module.
; Multiple modules can come from a single directory.
;
pythonpath = C:\Users\pyxll\modules
```

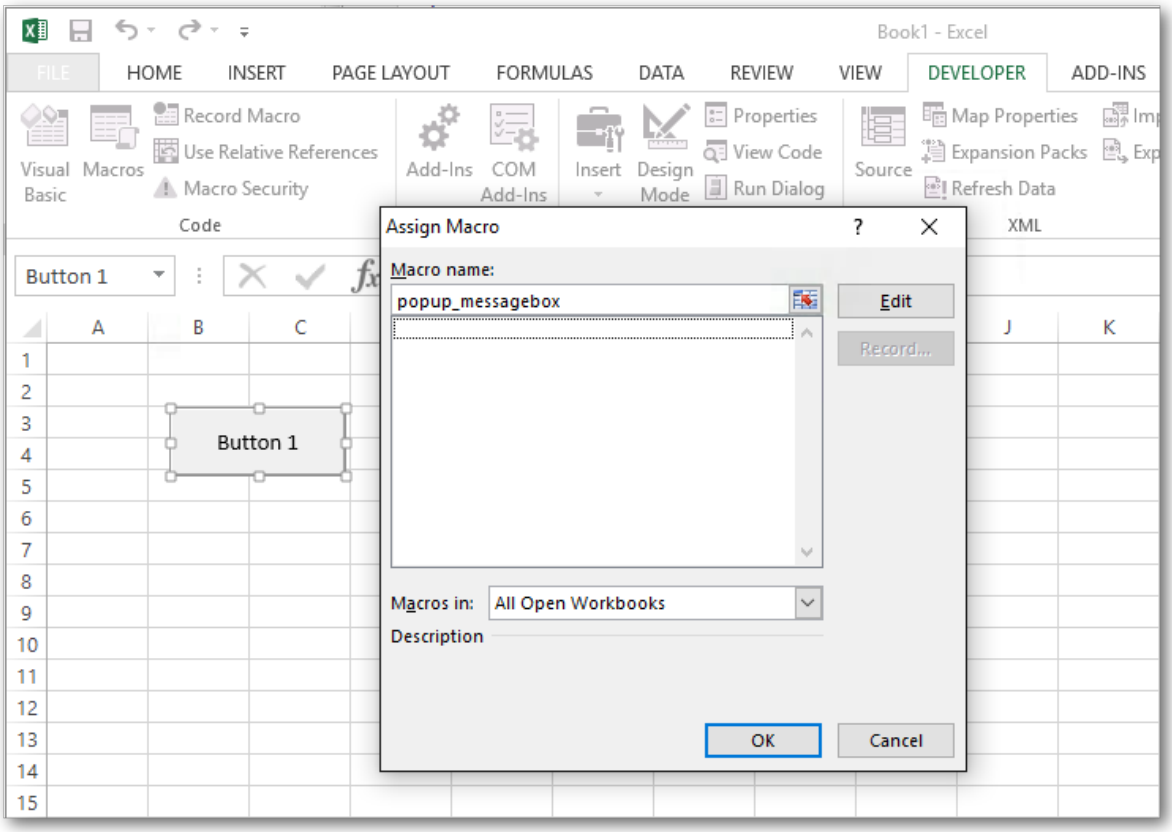
3.4.2 Calling Macros From Excel

Macros defined with PyXLL can be called from Excel the same way as any other Excel macros.

Once you have written your Python function, decorated it with the `@xl_macro` decorator, and configured PyXLL (as described in the *previous section*) you are ready to call the macro function from Excel.

> **Tip**
>
> Don't forget you need to reload after making changes to your Python code or the PyXLL config!

The most usual way is to assign a macro to a control. To do that, first add the Forms toolbox by going to the Tools Customize menu in Excel and check the Forms checkbox. This will present you with a panel of different controls which you can add to your worksheet. For the message box example above, add a button and then right click and select 'Assign macro…'. Enter the name of your macro, in this case *popup_messagebox*. Now when you click that button the macro will be called.



> **Warning**
>
> The *Assign Macro* dialog in Excel will only list macros defined in workbooks. Any macro defined in Python using `@xl_macro` will not show up in this list. Instead, you must enter the name of your macro manually and Excel will accept it.

It is also possible to call your macros from VBA. While PyXLL may be used to reduce the need for VBA in your projects, sometimes it is helpful to be able to call python functions from VBA.

Here's an example Python function exposed to Excel as a macro:

```
from pyxll import xl_macro

@xl_macro("str s: int")
def py_strlen(s):
    """Return the length of a string"""
    return len(s)
```

For the `py_strlen` example above, to call that from VBA you would use the Run VBA function, e.g.

```
Sub SomeVBASubroutine
    x = Run("py_strlen", "my string")
End Sub
```

3.4.3 Accessing Worksheet Data

Reading Excel Values from Python

Macros will often need to read values from Excel. We can do that using the Excel Object Model in the same way as you might have done from VBA previously.

For example, the following code gets a `Range` object and then gets the value using the `Range.Value` property.

```
from pyxll import xl_macro, xl_app

@xl_macro
def read_value_from_excel():
    # Get the Excel.Application COM object
    xl = xl_app()

    # Get a Range in the current active sheet
    xl_range = xl.ActiveSheet.Range("A1:D10")

    # Access the Value of the Range
    value = xl_range.Value
```

What you will notice when reading values this way is that you can't control the type of the value you get. For example, in the code above we are using a range and the Python value obtained will be a list of lists. If we were to use a single cell, the Python value would be a single value.

You can use *`get_type_converter`* to access PyXLL's type converters (including any custom type converters you may have written) and use that to convert the raw value into the type you require. However, there is a slightly easier way.

Using the PyXLL class *`XLCell`* we can access the value as a specified type as follows:

```
from pyxll import xl_macro, xl_app, XLCell

@xl_macro
def read_value_from_excel():
    # Get the Excel.Application COM object
    xl = xl_app()

    # Get a Range in the current active sheet
    xl_range = xl.ActiveSheet.Range("A1:D10")

    # Get an XLCell object from the Range object
    cell = XLCell.from_range(xl_range)
```

(continues on next page)

(continued from previous page)

```
    # Get the value as a DataFrame
    df = cell.options(type="dataframe").value
```

Writing Python Values to Excel

Writing values to Excel from Python is very similar to reading values, as shown above.

While you can set the `Range.Value` property directly, using *`XLCell.value`* is often more convenient as it can do any type conversions necessary for you.

For example, to write a `DataFrame` to a range you can do the following:

```
from pyxll import xl_macro, xl_app, XLCell

@xl_macro
def write_value_to_excel():
    # Get the Excel.Application COM object
    xl = xl_app()

    # Get a Range in the current active sheet
    xl_range = xl.ActiveSheet.Range("A1")

    # Get an XLCell object from the Range object
    cell = XLCell.from_range(xl_range)

    # Create the DataFrame we want to write to Excel
    df = your_code_to_construct_the_dataframe()

    # Write the DataFrame to Excel, automatically resizing the range to fit the data.
    cell.options(auto_resize=True, type="dataframe").value = df
```

In the code above you can see that as well as converting the `DataFrame` to the type expected by Excel, *`XLCell`* can also automatically expand the range that's being written to fit the entire `DataFrame`.

For more details about the options available, see the *`XLCell`* class reference.

Accessing Cached Objects

When writing an Excel macro, if you need to access a cached object from a cell or set a cell value and cache an object you can use the *`XLCell`* class. Using the *`XLCell.options`* method you can set the type to `object` before getting or setting the cell value. For example:

```
from pyxll import xl_macro, xl_app, XLCell

@xl_macro
def get_cached_object():
    """Get an object from the cache and print it to the log"""
    # Get the Excel.Application object
    xl = xl_app()

    # Get the current selection Range object
    selection = xl.Selection

    # Get the cached object stored at the selection
    cell = XLCell.from_range(selection)
    obj = cell.options(type="object").value

    # 'value' is the actual Python object, not the handle
```

(continues on next page)

(continued from previous page)

```
    print(obj)


@xl_macro
def set_cached_object():
    """Cache a Python object by setting it on a cell"""
    # Get the Excel.Application object
    xl = xl_app()

    # Get the current selection Range object
    selection = xl.Selection

    # Create our Python object
    obj = object()

    # Cache the object in the selected cell
    cell = XLCell.from_range(selection)
    cell.options(type="object").value = obj
```

3.4.4 Passing Python Objects

New in PyXLL 5.11

While it is possible to completely replace VBA macros with Python macros, sometimes it is necessary to mix the two and call Python macros from VBA. For example, when migrating a large VBA project to Python it may not be feasible to change everything in one go.

When mixing VBA and Python occasionally you may want to pass a complex Python object from Python to VBA, in order to pass it to another Python function later.

Not all Python types convert easily to standard Excel types. In order to pass Python objects from Python to VBA and from VBA to Python, PyXLL mainains an 'object cache'. Rather than return an actual Python object, PyXLL can store a returned object in this cache and return a handle to that object from the macro. This handle can then be passed to other Python macros, and PyXLL will retreive the object from the cache.

For example, here are two macros - one that returns an object, and one that expects an object. To tell PyXLL that we want to pass this object via the object cache the functions use the 'object' type:

```
from pyxll import xl_macro

class CustomObject:
    def __init__(self, name):
        self.name = name

@xl_macro("string name: object")
def create_object(x):
    return CustomObject(x)

@xl_macro("object x: string")
def get_object_name(x):
    assert isinstance(x, CustomObject)
    return x.name
```

There's no other code needed, just marking the object type as 'object' is enough to tell PyXLL that the return value should be put in the object cache and a handle to that object returned to Excel.

We can call these functions from VBA:

```
Sub Test

    ' a is an object handle we can pass to other Python Macros
    a = Run("create_object", "TEST")

    ' get_object_name is called with the object handle
    b = Run("get_object_name", a)

End Sub
```

Cached objects in macros work in the same way as *cached objects for worksheet functions*, with just one important difference:

Unlike objects that are returned to the worksheet, **objects returned via macros have an expiry time after which they are removed from the cache**. This is explained in more detail in the following section.

Cached Object Lifetime

New in PyXLL 5.11

When using cached objects from worksheet functions, the object handle is returned to the Excel sheet and the lifetime of the cached object is managed by PyXLL. When the cell containing the object handle is cleared, the cached object is removed from the cache.

For macro functions, the object handle is not written to the Excel sheet and so PyXLL has no way of knowing when the cached object is no longer required.

To prevent the object cache from growing indefinitely, objects returned from macros are removed from the cache after a pre-determined amount of time. Usually, as objects are passed to VBA in order to be passed to another Python function soon after, this amount of time only needs to be a few seconds.

Python objects returned from macros are called *transient objects* as they only remain in the cache for a finite period of time.

The amount of time a transient object is given before it will be removed from the cache is called the *time to live* or *TTL*.

The default TTL can be set in the `pyxll.cfg` config file as follows:

```
[PYXLL]
; Default time to live for transient cached objects, in seconds
object_cache_transient_ttl = 10
```

The TTL for an object can also be set in your Python code. The TTL can be set before the object is returned, or at any time after. Changing the object's TTL restarts the clock to when it will be removed from the cache.

The function `set_transient_object_ttl` can be used to set the TTL for an object. If an object should never be removed from the cache, a negative TTL can be used.

For example, if a returned object should be kept in cache for 5 minutes you would do the following:

```
from pyxll import xl_macro, set_transient_object_ttl

class CustomObject:
    def __init__(self, name):
        self.name = name

@xl_macro("string name: object")
def create_long_lived_object(x):
    obj = CustomObject(x)

    # Keep the object alive for at least 5 minutes
```

(continues on next page)

(continued from previous page)

```
    set_transient_object_ttl(obj, 300)

    return obj
```

3.4.5 Pausing Automatic Recalculation

For macros that modify the workbook, having Excel update or recalculate after each individual change can result in poor performance.

It is quite common to temporarily disable Excel's screen updating and automatic calculations while performing such changes to the workbook in a macro.

PyXLL provides the context manager `xl_disable` for this purpose. The same can be acheived using the Excel Object Model (see *Python as a VBA Replacement*), but using `xl_disable` can be more convenient.

The following example shows how `xl_disable` can be used to temporarily disable screen updating and automatic calculations while making an update to the workbook:

```
@xl_macro
def macro_function():
    with xl_disable():
        # do some work here that updates Excel where we do not
        # want Excel to automatically recalculate or update.
        xl = xl_app()
        xl.Range("A1").Value = 1

    # After the with block, Excel reverts to its previous calculation mode.
    return
```

Similar options are available when using the `@xl_macro` decorator to disable updating and calculations for the duration of the entire macro.

3.4.6 Keyboard Shortcuts

You can assign keyboard shortcuts to your macros by using the 'shortcut' keyword argument to the `@xl_macro` decorator, or by setting it in the *SHORTCUTS* section in the *config*.

Shortcuts should be one or more modifier key names (*Ctrl*, *Shift* or *Alt*) and a key, separated by the '+' symbol. For example, 'Ctrl+Shift+R'.

```
from pyxll import xl_macro, xlcAlert

@xl_macro(shortcut="Alt+F3")
def macro_with_shortcut():
    xlcAlert("Alt+F3 pressed")
```

If a key combination is already in use by Excel it may not be possible to assign a macro to that combination.

In addition to letter, number and function keys, the following special keys may also be used (these are not case sensitive and cannot be used without a modifier key):

- Backspace
- Break
- CapsLock
- Clear
- Delete
- Down

- End
- Enter
- Escape
- Home
- Insert
- Left
- NumLock
- PgDn
- PgUp
- Right
- ScrollLock
- Tab

3.5 Real Time Data

- *Introduction*
- *RTD Generators*
  - *Async RTD Generators*
  - *Setting an Initial Value*
- *Using the RTD Class*
  - *RTD Class Example*
  - *RTD Class Async Example*
- *RTD Data Types*
- *Restarting RTD Functions*
  - *Detaching*
  - *Detaching RTD Generators*
  - *Detaching RTD Instances*
- *Throttle Interval*
- *Starting RTD Functions Automatically*

3.5.1 Introduction

Real Time Data (or *RTD*) is data that updates asynchronously, according to its own schedule rather than just when it is re-evaluated (as is the case for a regular Excel worksheet function).

Examples of real time data include stock prices and other live market data, server loads or the progress of an external task.

Real Time Data has been a first-class feature of Excel since Excel 2002. It uses a hybrid push-pull mechanism where the source of the real time data notifies Excel that new data is available, and then some small time later Excel queries the real time data source for its current value and updates the value displayed.

PyXLL provides a convenient and simple way to stream real time data to Excel without the complexity of writing (and registering) a Real Time Data COM server.

Real Time Data functions are registered in the same way as other worksheet functions using the *@xl_func* decorator. By registering a Python generator function, or a function that returns an *RTD* object, streams of values can be returned to Excel as a real time data function.

RTD functions have the return type `rtd`.

3.5.2 RTD Generators

New in PyXLL 5.6

The simplest way to write an Excel RTD (Real Time Data) function using PyXLL is to write it as a Python generator.

A Python generator is a special type of Python function that yields a stream of results instead of just a single result.

The following is an example of a Python generator that yields a random number every 5 seconds:

```
import random
import time

def random_numbers():
    # Loop forever
    while True:
        # Yield a random number
        yield random.random()

        # Wait 5 seconds before continuing
        time.sleep(5)
```

To turn this into an RTD function in Excel all that's needed is to add the *@xl_func* decorator with the `rtd` return type:

```
from pyxll import xl_func
import random
import time

@xl_func(": rtd")
def random_numbers():
    while True:
        yield random.random()
        time.sleep(5)
```

When this `random_numbers` function is called from Excel it will tick every 5 seconds with a new random number.

To prevent the long running Python generator function from blocking the main Excel thread RTD generators are always run on a background thread. One thread is created for each generator. For more sophisticated thread management use a function that returns an *RTD* instance instead (see *Using the RTD Class*).

Each time the generator yields a value Excel is notified a new value is ready. Excel may not display every value as it throttles the updates that is displays (see *Throttle Interval*).

It's important not to create a tight loop that constantly updates as doing so will prevent other threads from having time to run (including the main Excel thread) and will cause Excel to hang.

> **Warning**
>
> Unlike other PyXLL functions, RTD generators are always run on a background thread.

Async RTD Generators

New in PyXLL 5.6

Async RTD generators work in a similar way to the RTD generators described above. Instead of running in a separate thread async RTD generators are run on PyXLL asyncio event loop.

Async generators are well suited to IO bound tasks such as receiving updates from a remote server. Care should be taken so that the asyncio event loop isn't blocked by ensuring that tasks are properly asynchronous and are awaited correctly.

The same example as above can be re-written as an async generator by replacing `time.sleep` with `asyncio.sleep`:

```
from pyxll import xl_func
import random
import asyncio

@xl_func(": rtd")
async def async_random_numbers():
    # Loop forever
    while True:
        # Yield a random number
        yield random.random()

        # Wait 5 seconds before continuing without blocking the event loop
        await asyncio.sleep(5)
```

When this `async_random_numbers` function is called from Excel it will tick every 5 seconds with a new random number.

Each time the generator yields a value Excel is notified a new value is ready. Excel may not display every value as it throttles the updates that is displays (see *Throttle Interval*).

> **Warning**
>
> Async RTD generators share the same asyncio loop as other async functions. Blocking the asyncio event loop will cause delays in other functions, or prevent them from running entirely.
>
> See *The asyncio Event Loop* for more details.

Setting an Initial Value

New In PyXLL 5.12

When using an RTD generator, Excel may already have finished calculating by the time your generator yields its first value. If this happens you may see an empty cell or 0 (depending on your Excel settings) until the first value is yielded from the generator.

To avoid this you can set an initial value using the `initial_value` type parameter to the `rtd` type. For example:

```
@xl_func("int x: rtd<inital_value='Please wait...'")
async def your_generator(x):
    while running:
        yield ...
```

This will return the string `Please wait...` back to Excel as the first value for the RTD function, before your generator has yielded its first value.

> **Tip**
>
> You can use the *@xl_return* decorator if you don't want to have the value in the signature string, for example:

```
@xl_func
@xl_return(initial_value='Please wait...')
async def your_generator(x: int) -> pyxll.RTD:
    while running:
        yield ...
```

3.5.3 Using the RTD Class

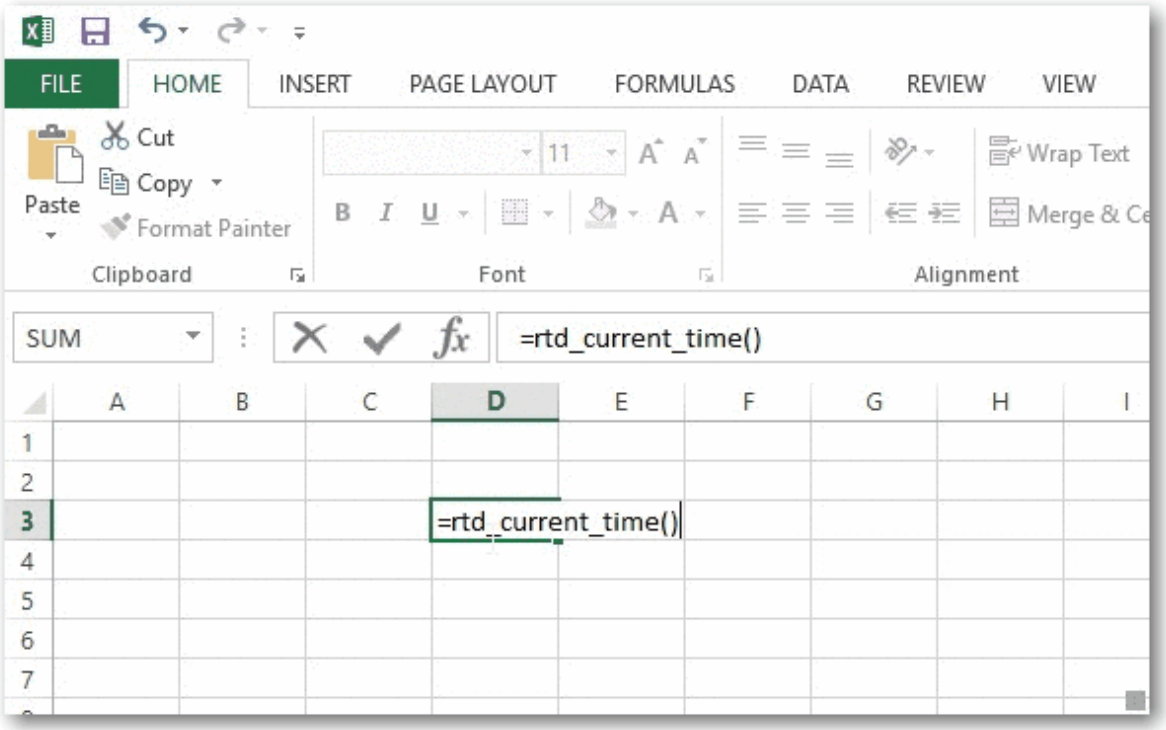
For more control over the RTD behaviour of an Real Time Data function an *RTD* object can be returned from an RTD function instead of using a generator.

The *RTD* class has a `value` property. Setting this property notifies Excel that a new value is ready. RTD functions that use the *RTD* class return an *RTD* object and update the value on that returned object each time a new value is available (for example, by scheduling an update function on a background thread). Typically this is done by writing a derived class that handles updating its own value.

If multiple function calls from different cells return the same *RTD* object then those cells are subscribed to the same object. All the cells will update whenever the *value* property of the one *RTD* object is set.

RTD Class Example

The following example shows a class derived from *RTD* that periodically updates its value to the current time.



It uses a separate thread to set the *value* property, which notifies Excel that new data is ready.

```
from pyxll import xl_func, RTD
from datetime import datetime
```

(continues on next page)

(continued from previous page)

```
import threading
import logging
import time

_log = logging.getLogger(__name__)


class CurrentTimeRTD(RTD):
    """CurrentTimeRTD periodically updates its value with the current
    date and time. Whenever the value is updated Excel is notified and
    when Excel refreshes the new value will be displayed.
    """

    def __init__(self, format):
        initial_value = datetime.now().strftime(format)
        super(CurrentTimeRTD, self).__init__(value=initial_value)
        self.__format = format
        self.__running = True
        self.__thread = threading.Thread(target=self.__thread_func)
        self.__thread.start()

    def connect(self):
        # Called when Excel connects to this RTD instance, which occurs
        # shortly after an Excel function has returned an RTD object.
        _log.info("CurrentTimeRTD Connected")

    def disconnect(self):
        # Called when Excel no longer needs the RTD instance. This is
        # usually because there are no longer any cells that need it
        # or because Excel is shutting down.
        self.__running = False
        _log.info("CurrentTimeRTD Disconnected")

    def __thread_func(self):
        while self.__running:
            # Setting 'value' on an RTD instance triggers an update in Excel
            new_value = datetime.now().strftime(self.__format)
            if self.value != new_value:
                self.value = new_value
            time.sleep(0.5)
```

In order to access this real time data in Excel all that's required is a worksheet function that returns an instance of this `CurrentTimeRTD` class.

```
@xl_func("string format: rtd")
def rtd_current_time(format="%Y-%m-%d %H:%M:%S"):
    """Return the current time as 'real time data' that
    updates automatically.

    :param format: datetime format string
    """
    return CurrentTimeRTD(format)
```

Note that the return type of this function is *rtd*.

When this function is called from Excel the value displayed will periodically update, even though the function `rtd_current_time` isn't volatile and only gets called once.

```
=rtd_current_time()
```

RTD Class Async Example

Instead of managing your own background threads and thread pools when writing RTD functions, you can use PyXLL's `asyncio` event loop instead (new in PyXLL 4.2 and requires Python 3.5.1 or higher).

This can be useful if you have RTD functions that are waiting on IO a lot of the time. If you can take advantage of Python's `async` and `await` keywords so as not to block the event loop then making your RTD function run on the asyncio event loop can make certain things much simpler.

The methods *`RTD.connect`* and *`RTD.disconnect`* can both be `async` methods. If they are then PyXLL will schedule them automatically on it's asyncio event loop.

The example below shows how using the event loop can eliminate the need for your own thread management.

See *The asyncio Event Loop* for more details.

```
from pyxll import RTD, xl_func
import asyncio

class AsyncRTDExample(RTD):

    def __init__(self):
        super().__init__(value=0)
        self.__stopped = False

    async def connect(self):
        while not self.__stopped:
            # Yield to the event loop for 1s
            await asyncio.sleep(1)

            # Update value (which notifies Excel)
            self.value += 1

    async def disconnect(self):
        self.__stopped = True


@xl_func(": rtd<int>")
def async_rtd_example():
    return AsyncRTDExample()
```

3.5.4 RTD Data Types

RTD functions can return all the same data types as normal *Worksheet Functions*, including array types and cached Python objects.

By default, the `rtd` return type will use the same logic as a worksheet function with no return type specified or the `var` type.

To specify the return type explicity you have to include it in the function signature as a parameter to the `rtd` type.

For example, the following is how an RTD function that returns Python objects via the internal object cache would be declared:

```
@xl_func("string x: rtd<object>")
def rtd_object_func(x):
    # MyRTD sets self.value to a non-trivial Python object
    return MyRTD(x)
```

RTD data types can be used for RTD generators in the same way.

Although RTD functions can return array types, they cannot be automatically resized and so the array formula needs to be entered manually using *Ctrl+Shift+Enter* (see *Array Functions*).

3.5.5 Restarting RTD Functions

Each time your RTD function ticks, Excel re-evaluates the RTD function. PyXLL keeps track of your RTD generator or RTD instance to ensure that each time Excel calls the RTD function the correct, already running, object is returned.

Without this caching behavior your Python function would be called each time the RTD function ticks, causing it to always restart and never progress past the initial value.

If the arguments to an RTD function are changed, or if it is moved from one cell to another, the Python function will be called and the RTD function will restart. Because the RTD object has to be cached the usual methods of forcing a cell to recalculate such as pressing *F2* to edit the cell and then entering, or force recalculating by pressing *Ctrl+Alt+F9*, do not work for RTD functions.

There are situations though where an RTD function only returns a fixed number of values before stopping. In those cases you may want the RTD function to restart the next time Excel calculates the cell.

For example, one common case is an RTD function that returns just one value after a delay:

```
@xl_func("str x: rtd")
async def fetch_from_database(x):
    # Let the user know we're working on it
    yield "Please wait..."

    # Fetch some data
    data = await async_fetch_data_from_database(x)

    # Finally yield the data to Excel
    yield data
```

In this case, if the user was to attempt to recalculate the cell after the data has been returned they might expect that the data be re-fetched. But, the default behaviour is that the RTD generator is cached and so its same current state is returned again.

Detaching

New in PyXLL 5.9

The solution is to *detach* the RTD generator or RTD object once completed. This *detaching* removes the RTD generator or object from PyXLL's RTD cache so that the next time the cell is evaluated there is no cached RTD instance. The Python function is called again, and the Excel RTD function restarts.

Detaching RTD Generators

When using an RTD generator or async generator you can tell PyXLL to automatically detach the RTD object when the generator stops.

To do that, add the `auto_detach=True` type paramter to the `rtd` return type.

For example:

```
@xl_func("str x: rtd<auto_detach=True>")
async def fetch_from_database(x):
    # Let the user know we're working on it
    yield "Please wait..."

    # Fetch some data
```

(continues on next page)

(continued from previous page)

```
    data = await async_fetch_data_from_database(x)

    # Finally yield the data to Excel.
    # The generator stops after this line.
    yield data
```

Once the generator has stopped, if the user then recalculates the cell it will restart.

> **Tip**
>
> If you want this *auto_detach* behaviour to be the default for all of your RTD generators you can set the following in your pyxll.cfg file:
>
> ```
> [PYXLL]
> rtd_auto_detach = 1
> ```

Detaching RTD Instances

If you are using the `RTD` class you can detach the RTD object by calling the `RTD.detach` method.

After calling `RTD.detach`, the RTD instance will no longer be cached and the next call to your RTD function will call your Python function, restarting the RTD function.

For example, if in the `AsyncRTDExample` code from above, if we wanted our RTD function to restart after the first few ticks we would do the following:

```
from pyxll import RTD, xl_func
import asyncio

class AsyncRTDExample(RTD):

    def __init__(self):
        super().__init__(value=0)
        self.__stopped = False

    async def connect(self):
        # Send just a few ticks
        for i in range(5):
            if self.__stopped:
                break

            # Yield to the event loop for 1s
            await asyncio.sleep(1)

            # Update value (which notifies Excel)
            self.value += 1

        # And then detach.
        # The next time =async_rtd_example is called the Python
        # function will be called, starting a new AsyncRTDExample instance.
        self.detach()

    async def disconnect(self):
        self.__stopped = True


@xl_func(": rtd<int>")
```

(continues on next page)

(continued from previous page)

```
def async_rtd_example():
    return AsyncRTDExample()
```

3.5.6 Throttle Interval

Excel throttles the rate of updates made via RTD functions. Instead of updating every time it is notified of new data it waits for a period of time and then updates all cells with new data at once.

The default throttle time is 2,000 milliseconds (2 seconds). This means that even if you are setting *value* on an *RTD* instance or yielding values from a generator more frequently you will not see the value in Excel updating more often than once every two seconds.

The throttle interval can be changed by setting *Application.RTD.ThrottleInterval* (in milliseconds). Setting the throttle interval is persistent across Excel sessions (meaning that if you close and restart Excel then the value you set the interval to will be remembered).

The following code shows how to set the throttle interval in Python.

```
from pyxll import xl_func, xl_app

@xl_func("int interval: string")
def set_throttle_interval(interval):
    xl = xl_app()
    xl.RTD.ThrottleInterval = interval
    return "OK"
```

Alternatively it can be set in the registry by modifying the following key. It is a DWORD in milliseconds.

```
HKEY_CURRENT_USER\Software\Microsoft\Office\10.0\Excel\Options\RTDThrottleInterval
```

3.5.7 Starting RTD Functions Automatically

When you enter an RTD function in an Excel formula is begins ticking automatically because the function has been called. When loading a workbook containing RTD functions however, they will not start ticking *until* the function is called.

To enable RTD functions to begin ticking as soon as a workbook is opened PyXLL RTD functions can be marked as needed to be recalculated when the workbook opens by using the *Recalculating On Open* feature of PyXLL.

To make a function recalculate when the workbook is loaded pass `recalc_on_open=True` to *`@xl_func`*. If applied to an RTD function this will cause the RTD function to start ticking when the workbook is loaded.

You can change the default behaviour so that `recalc_on_open` is `True` by default for RTD functions (unless explicitly marked otherwise) by setting `recalc_rtd_on_open = 1`, e.g.

```
[PYXLL]
recalc_rtd_on_open = 1
```

> **Warning**
>
> The default behaviour for RTD functions has changed between PyXLL 4 and PyXLL 5.
>
> From PyXLL 5 onwards RTD functions will no longer start automatically when a workbook is opened unless configured as above. This is consistent with other UDFs that are not called automatically when workbooks open by default.

3.6 Using Type Hints

There are many places in PyXLL where you need to specify type information to inform PyXLL how to convert between Python and Excel values.

Since Python 3, Python has its own way of adding type information to functions through type hinting. For new PyXLL users, type hinting is probably the most natural way to specify types within PyXLL.

For a list of the basic supported types, see *Standard Types*.

Array Types, *NumPy Array Types*, *Pandas Types*, and *Polars DataFrames* are also supported, as are *Custom Types*.

- *Type Hints for UDFs*
- *Type Parameters*
- *Cached Objects*
- *Real Time Data Types*

Note

For those interested in more details...

PyXLL has its own internal type representation that it uses to map between Python and Excel types, and there are several ways to specify types in PyXLL. The PyXLL internal type system was developed long before Python's own type hints, and so that is why there are now a few different ways of specifying types when using PyXLL.

Python type hints identify a type, whereas PyXLL's type system is subtly different in that it declares the exactly how a type will be converted between Excel and Python. For simple types there is no real difference, but for more complex types PyXLL has the concept of type parameters which modify how the type conversion happens.

Since PyXLL 5.12, annotated type hints can be used that add type parameters to type hints. Using annotated type hints, all PyXLL types can be represented by Python type hints.

In the documentation you will see *signature strings* often used. These are parsed into PyXLL's type representation, as are type hints. Which you use is mostly down to your own preference.

3.6.1 Type Hints for UDFs

The most common place where type information is necessary is when exposing a Python function to Excel via the `@xl_func` and `@xl_macro` decorators.

The argument types determine how the function will be registered in Excel, as well as how the Excel values will be converted into Python values. Similarly, the return type dictates how the returned Python value will be converted before being returned to Excel.

To understand why type hinting necessary first consider a function without types:

```
from pyxll import xl_func

@xl_func
def some_function(a, b, c):
    return a + b + c
```

This function will do what you expect, if you pass it numbers! PyXLL will convert the Excel types to the closest Python type, and so Excel numbers will be converted to Python `float` types. So far so good...

Now consider this next example:

```
from pyxll import xl_func

@xl_func
def pick_a_value(data, row_index, col_index):
    return data[row_index][col_index]
```

If you pass this function a range of values, a row index, and a column index - for example, `=pick_a_value(A1:C3, 1, 1)`, you might be surprised that it fails!

The range `A1:C3` will be converted to a Python list of lists (list of rows, where each row is itself a list), and we can index into that in the way shown – but, the indexing will fail because the numbers passed as `row_index` and `col_index` are Excel numbers which will be converted to Python `float` types. We need Python `int` types in order to index correctly.

Also, what would happen if we were to pass a single cell as our `data` argument? Since PyXLL doesn't know that we always want that argument to be passed as a list, it will pass a scalar value in that case.

The solution is to add some type information to control how the values are converted. Using type hints, we can do that as follows:

```
from pyxll import xl_func
import typing

@xl_func
def pick_a_value(data: list[list],
                 row_index: int,
                 col_index: int) -> typing.Any:
    return data[row_index][col_index]
```

Note that we didn't need to change the actual function, just the argument and return types (although the return type is not strictly necessary here).

The type hints above are equivalent to using a PyXLL signature string as follows:

```
from pyxll import xl_func

@xl_func("list<ndim=2> data, int row_index, int col_index: var")
def pick_a_value(data, row_index, col_index):
    return data[row_index][col_index]
```

Either syntax is fine, and both achieve the same thing.

> **Note**
>
> Python type hints have changed over the Python 3.x releases.
>
> The exact syntax that you need to use will depend on exactly what version of Python you're using, and may vary from what is shown in the document.
>
> As an example, `list[list]]` used above requires Python >= 3.9. In earlier Python versions, `typing.List[typing.List]` can be used instead.
>
> If you are restricted on the version of Python you can use, and a particular type hint is not possible in your chosen Python version, you can always revert to using PyXLL signature strings as those work in all Python versions.

3.6.2 Type Parameters

In PyXLL, argument and return types are used for more than just to specify the Python type – they control exactly how the conversion between the Python and Excel types happen. For many types there are options to how this conversion happens, as well as the type information.

For example, `DataFrame` types can be converted to and from ranges of values in Excel. But, the fact that an argument or return type is a `DataFrame` is not enough to control *how* that conversion happens.

In the `DataFrame` example, we can choose whether we want the index and columns to be included, or if we want to trim blank space from the range before converting, or various other options. These other options are what are referred to as **type parameters** in the PyXLL documentation.

Specifying type parameters using a type signature string is done as follows:

```
from pyxll import xl_func

@xl_func("dataframe<index=True> df, str column: float")
def sum_column(df, column):
    return df[column].sum()
```

In the above, `index=True` specifies that when converting from the Excel range to the Python `DataFrame` the first column of the data should be interpreted as the DataFrame's index.

To do the same using only Python type hints we need to add this information to the type using the `typing.Annotated` class and PyXLL's *`TypeParameters`* class. The code above can be re-written as:

```
from pyxll import xl_func
from pyxll import TypeParameters as TP
from typing import Annotated
import pandas as pd

@xl_func
def sum_column(
    df: Annotated[pd.DataFrame, TP(index=True)],
    column: str
) -> float:
    return df[column].sum()
```

Using `typing.Annotated` and *`TypeParameters`* all of the options for PyXLL's various type converters can be specified using Python type hints.

3.6.3 Cached Objects

Python objects can be returned from Excel functions as *cached object handles*, to be passed seamlessly to other Python functions. The Python objects are cached by PyXLL and retrieved when passing objects by their handle into another PyXLL function.

To tell PyXLL to pass a Python object as an object handle the `object` type is used, but when writing Python code with type hints using the `object` type directly is often too broad and not helpful as a type hint.

For this reason PyXLL provides the generic type alias *`Object`*[1], which can be used as `Object[T]` where `T` is the actual type. Type checkers will see `T`, but the PyXLL add-in will understand that an object handle is to be used.

For example, the following returns a `pandas.DataFrame` to Excel as an object handle. Type checkers will correctly see this function as returning a `DataFrame`.

```
from pyxll import xl_func, Object
import pandas as pd

@xl_func
def load_dataframe() -> Object[pd.DataFrame]:
    df = ... # load or build DataFrame object
    return df
```

See also *Cached Objects*.

> **Note**
>
> Generic type aliases require Python 3.12+. If you are using an earlier version of Python you will need to specify the return type as `object` using either a signature string or the *`@xl_return`* decorator.

3.6.4 Real Time Data Types

When defining a Real Time Data, or RTD, function, it's necessary to set the return type to *rtd*. This tells the PyXLL add-in to register the function differently, as it's an RTD function.

When using type hints, if writing an RTD function that returns an *`RTD`* instance that is no problem and you can simply use *`RTD`* as your return type. The *`RTD`* class is a generic type[2] and you can specify the inner type of the RTD value as well, for example:

```
from pyxll import xl_func, RTD

# This class handles the RTD updates and self.value is set to 'int' values
class MyRtdType(Rtd):
    ...

@xl_func
def my_rtd_function(...) -> RTD[int]:
    return MyRtdType(...)
```

> **Note**
>
> Subscripting the *`RTD`* class requires Python 3.9+ and PyXLL 5.12+. For earlier versions, use *`@xl_return`* to add the RTD return type information.

When using a generator or async generator, we still need to tell the PyXLL add-in to treat the generator as an RTD function, but the return type hint now needs to be a generator type and not the plain *`RTD`* type.

PyXLL provides two generic type aliases for this purpose, *`RTDGenerator`* and *`RTDAsyncGenerator`*[3].

[1] The generic type alias *`Object`* requires a minimum of Python 3.12 and PyXLL 5.12.
[2] Indexing the *`RTD`* type requires a minimum of Python 3.9 and PyXLL 5.12.
[3] The generic type aliases *`RTDGenerator`* and *`RTDAsyncGenerator`* require a minimum of Python 3.12 and PyXLL 5.12.

```
from pyxll import xl_func, RTDGenerator, AsyncRTDGenerator
from typing import Any
import asyncio
import time

@xl_func
def my_rtd_generator(...) -> RTDGenerator[Any]:
    i = 0
    while True:
        i += 1
        yield i
        time.sleep(5)

@xl_func
async def my_async_rtd_generator(...) -> AsyncRTDGenerator[Any]:
    i = 0
    while True:
        i += 1
        yield i
        await asyncio.sleep(5)
```

To type checkers, `RTDGenerator[T]` is an alias for `typing.Generator[T, None]` and `AsyncRTDGenerator[T]` is an alias for `typing.AsyncGenerator[T, None]`, but the PyXLL add-in recognizes these as meaning that the function should be registered as an RTD function.

PyXLL's `RTD` type has its own type parameters, such as `initial_value`.We can set those using `typing.Annotated` and *`TypeParameters`* in the same way as shown earlier:

```
from pyxll import xl_func, RTDGenerator
from pyxll import TypeParameters as TP
from typing import Annotated
import time

@xl_func
def my_rtd_generator(...) -> Annotated[RTDGenerator[int], TP(initial_value=0)]:
    i = 0
    while True:
        i += 1
        yield i
        time.sleep(5)
```

This is equivalent to the type signature string `rtd<int, initial_value=0>`.

> **Note**
>
> Generic type aliases require Python 3.12+. If you are using an earlier version of Python you will need to specify the return type as `rtd<T>` using either a signature string or the *`@xl_return`* decorator.

3.7 Cell Formatting

When returning values or arrays from a *worksheet function*, or when setting values on a sheet using a *macro function*, often you will also want to set the formatting of the values in Excel. This can be to make sure a returned value has the correct date or number format, or styling a whole table.

Standard formatters are provided for common cases, and you can also write your own formatters to achieve the exact style you need.

3.7.1 Formatting Worksheet Functions

Worksheet functions registered using `@xl_func` can format their results using a `Formatter`.

To specify what formatter should be used for a function use the `formatter` kwarg to the `@xl_func` decorator. For example:

```
from pyxll import xl_func, Formatter
import datetime as dt

date_formatter = Formatter(number_format="yyyy-mm-dd")

@xl_func(formatter=date_formatter)
def get_date():
    return dt.date.today()
```

When the function is called from Excel, any previous formatting is cleared and the formatter is applied to the cell.

The standard `Formatter` handles many common formatting requirements and takes the following options:

Formatter kwargs	
interior_color	Color value to set the interior color to.
text_color	Color value to set the text color to.
bold	If True, set the text style to bold.
italic	If True, set the text style to italic.
font_size	Value to set the font size to.
number_format	Excel number format to use.
auto_fit	Auto-fit to the content of the cells. May be any of: True (fit column width); False (don't fit); "columns" (fit column width); "rows" (fit row width); "both" (fit column and row width);

Color values can be obtained using the static method `Formatter.rgb`.

More complex formatting can be done using a *custom formatter*.

The `Formatter` clears all formatting before applying the new formatting, but you can also control how the formatting is cleared using a *custom formatter*.

> **Note**
>
> When formatting is applied to Dynamic Array functions PyXLL will keep track of the current array size and save it in the *Workbook Metadata*.
>
> This is so the previous range can be cleared before re-applying formatting. Without doing this the formatting would remain if the array contracted.

3.7.2 Pandas DataFrame Formatting

Array formulas can also be formatted, and PyXLL provides the `DataFrameFormatter` class specifically for functions that return pandas DataFrames.

```
from pyxll import xl_func, xl_return, Formatter, DataFrameFormatter
import pandas as pd

df_formatter = DataFrameFormatter(
```

(continues on next page)

(continued from previous page)

```
    index=Formatter(bold=True, interior_color=Formatter.rgb(0xA9, 0xD0, 0x8E)),
    header=Formatter(bold=True, interior_color=Formatter.rgb(0xA9,0xD0,0x8E)),
    rows=[
        Formatter(interior_color=Formatter.rgb(0xE4, 0xF1, 0xDB)),
        Formatter(interior_color=Formatter.rgb(0xF4, 0xF9, 0xF1)),
    ],
    columns={
        "C": Formatter(number_format="0.00%")
    }
)

@xl_func(formatter=df_formatter, auto_resize=True)
@xl_return("dataframe<index=True>")
def get_dataframe():
    df = pd.DataFrame({
        "A": [1, 2, 3],
        "B": [4, 5, 6],
        "C": [0.3, 0.6, 0.9]
    })
    return df
```

When the function is called from Excel, any previous formatting is cleared and the formatter is applied to the range for the DataFrame.

The `DataFrameFormatter` class handles many common formatting requirements, but more complex formatting can be done by a *custom formatter*.

If the size of the DataFrame changes when inputs change, as long as the formula stays the same the previous range will be cleared before formatting the new range. This allows the returned range to contract without the formatting being left behind.

Conditional Formatting

As well as formatting specific rows and columns based on their position in the DataFrame as shown above, it is also possible to apply formatting that is conditional on the values in the DataFrame.

This is done using the `ConditionalFormatter` class.

The `ConditionalFormatter` class is constructed with an expression string and a formatter object. The expression string is passed to the `DataFrame.eval` method which returns a Series where that expression evaluates to True. The formatter will be applied to the rows where that expression is True. The formatting can be further restricted to only apply to specific columns.

A list of `ConditionalFormatter` objects can be passed as the `conditional_formatters` argument to `DataFrameFormatter`. The conditional formatters are applied in order after any other formatting has been applied.

The following example shows how to color rows green where column A is greater than 0 and red where column A is less than 0.

```
from pyxll import DataFrameFormatter, ConditionalFormatter, Formatter, xl_func
import pandas as pd

green_formatter = Formatter(interior_color=Formatter.rgb(0x00, 0xff, 0x00))
red_formatter = Formatter(interior_color=Formatter.rgb(0xff, 0x00, 0x00))

a_gt_zero = ConditionalFormatter("A > 0", formatter=green_formatter)
b_lt_zero = ConditionalFormatter("A < 0", formatter=red_formatter)

df_formatter = DataFrameFormatter(conditional_formatters=[
```

(continues on next page)

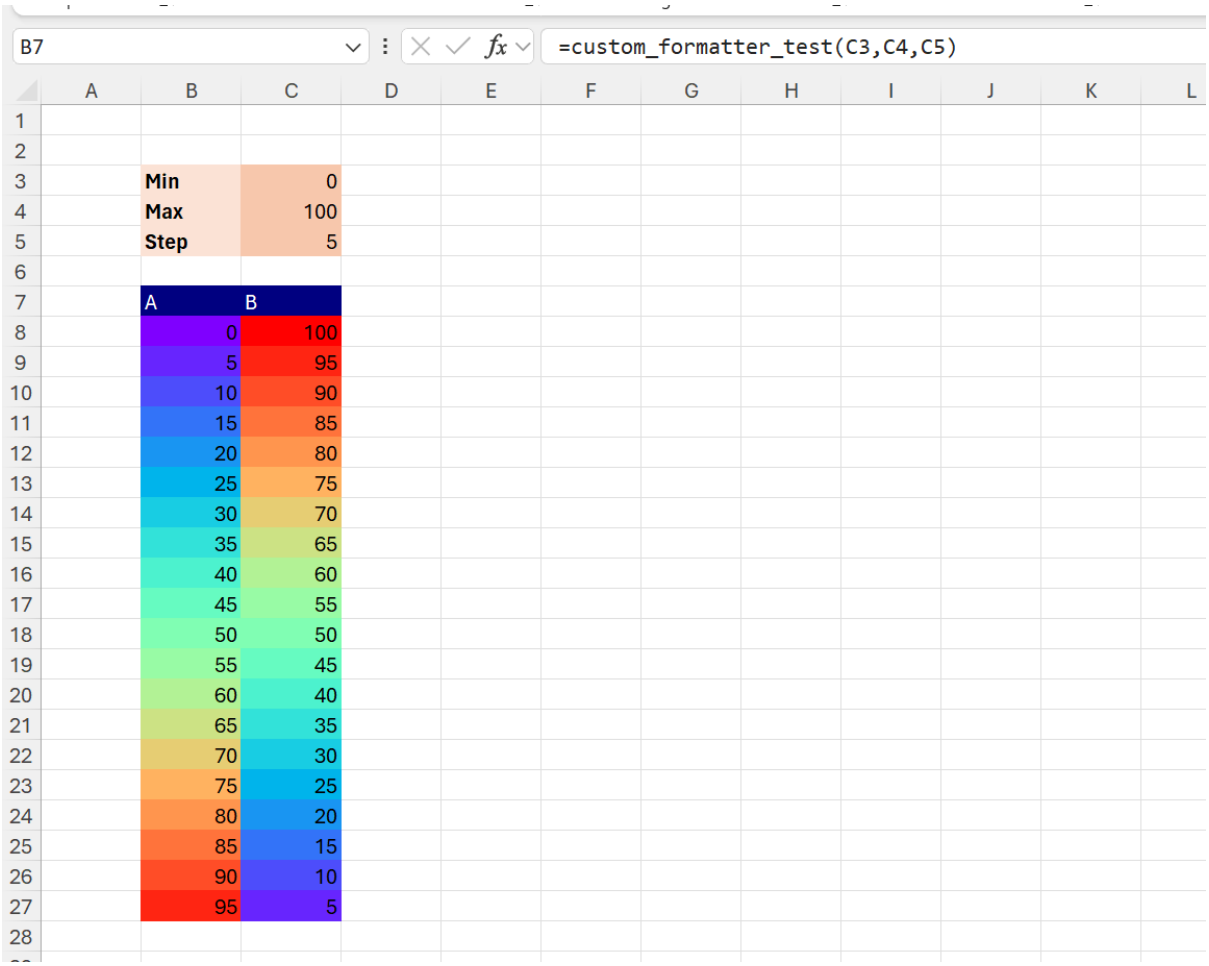
(continued from previous page)

```
                                    a_gt_zero,
                                    b_lt_zero])

@xl_func(": dataframe<index=False>", formatter=df_formatter, auto_resize=True)
def get_dataframe():
    df = pd.DataFrame({
        "A": [-1, 0, 1],
        "B": [1, 2, 3],
        "C": [4, 5, 6]
    })
    return df
```

To restrict the formatting to certain columns the `columns` argument to *`ConditionalFormatter`* can be used. This can be a list of column names or a function that takes a `DataFrame` and returns a list of columns.

Custom Conditional Formatters

For more complex conditional formatting a custom conditional formatter class can be derived from *`ConditionalFormatterBase`*.



The method *`ConditionalFormatterBase.get_formatters`* should be implemented to return a `DataFrame` of *`Formatter`* objects where any formatting is to be applied.

The returned `DataFrame` must have the same index and columns as the `DataFrame` being formatted.

The following example shows how a custom ConditionalFormatter can be written that changes the background color of cells in a DataFrame based on their value.

```
from pyxll import xl_func, DataFrameFormatter, ConditionalFormatterBase, Formatter
from matplotlib import colors, cm
import pandas as pd


class RainbowFormatter(ConditionalFormatterBase):

    def __init__(self, column, min=0, max=100, cmap="rainbow"):
        self.column = column
        self.min = min
        self.max = max
        self.cmap = cmap

    def get_formatters(self, df):
        # Create an empty DataFrame with the same index and columns as df.
        formatters = pd.DataFrame(None, index=df.index, columns=df.columns)

        # Normalize the column values into the range [0, 1]
        normalizer = colors.Normalize(self.min, self.max)
        values = normalizer(df[self.column])

        # Get a list of (r,g,b,a) colors from a colormap.
        colormap = cm.get_cmap(self.cmap)
        color_values = colormap(values)

        # Create the Formatter objects, remembering Formatter.rgb takes integers from␣
↪0 to 255.
        # This could use any Formatter class, including your own custom formatters.
        formatters[self.column] = [
            Formatter(interior_color=Formatter.rgb(int(r * 255), int(g * 255), int(b␣
↪* 255)))
            for r, g, b, a in color_values
        ]

        # Return the DataFrame containing Formatter objects for the cells we want to␣
↪format
        return formatters


# Construct a DataFrameFormatter using our custom RainbowFormatter class.
# Multiple formatters can be combined by adding them together.
df_formatter = DataFrameFormatter(
    header=Formatter(interior_color=Formatter.rgb(255, 255, 0)),
    conditional_formatters=[
        RainbowFormatter("A"),
        RainbowFormatter("B")
    ]
)


# This worksheet function uses our DataFrameFormatter and RainbowFormatters
@xl_func("int min, int max, int step: dataframe", formatter=df_formatter)
def custom_formatter_test(min=0, max=100, step=5):
    df = pd.DataFrame({"A": range(min, max, step), "B": range(max, min, -step)})
    return df
```

3.7.3 Custom Formatters

Although the standard formatters provide basic functionality to handle many common cases, you may want to apply your own formatting. This can be achieved using a custom formatter derived from `Formatter`.

For applying basic styles in your own formatter you can use `Formatter.apply_style`, but for everything else you can use the *Excel Object Model*.

With VBA it's possible to style cells and ranges by changing the background color, adding borders, and changing the font among other things. In Python it's no different as the entire *Excel Object Model is available to you in Python*, just as it is in VBA.

To write a custom formatter create a class that inherits from `Formatter`. The methods `Formatter.apply`, `Formatter.apply_cell` and `Formatter.clear` can be overridden to apply any formatting you require.

For example, if you wanted to apply borders using a formatter you would do the following:

```
from pyxll import Formatter, xl_func

# Needed to get VBA constants
from win32com.client import constants

class BorderFormatter(Formatter):

    # Formatter.apply takes other args and kwargs, like the value returned from
    # the function decorated with @xl_func, but we don't need those here.
    # See Formatter.apply for the full signature.
    def apply(self, cell, *args, **kwargs):
        # get the Excel.Range COM object from the XLCell
        xl_range = cell.to_range()

        # add a border to each edge
        for edge in (constants.xlEdgeLeft,
                     constants.xlEdgeRight,
                     constants.xlEdgeTop,
                     constants.xlEdgeBottom):
            border = xl_range.Borders[edge]
            border.LineStyle = constants.xlContinuous
            border.ColorIndex = 0
            border.TintAndShade = 0
            border.Weight = constants.xlThin

        # call the super class to apply any other styles
        super().apply(cell, *args, **kwargs)

border_formatter = BorderFormatter()

@xl_func(formatter=border_formatter, auto_resize=True)
def func_with_borders():
    return [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
    ]
```

You can use the VBA Macro Recorder to record a VBA Macro to apply any style you want, and then examine the recorded VBA code to see what you need to do. The recorded VBA code can be transformed into Python code.

For example, the following VBA code was recorded setting the left edge border. From the recorded code we can see what needs to be done and translate that into the required Python code as demonstrated above.

```
Sub Macro1()
    Range("D4:G8").Select

    With Selection.Borders(xlEdgeLeft)
      .LineStyle = xlContinuous
      .ColorIndex = 0
      .TintAndShade = 0
      .Weight = xlThin
    End With

End Sub
```

See *Python as a VBA Replacement* for more information on how to translate VBA code to Python.

Combining Multiple Formatters

Formatters can be combined so you do not have to implement every combination in a single formatter.

Formatters are combined by adding them to each other.

For example, to combine the above formatter with the standard `DataFrameFormatter` you add them together.

```
from pyxll import xl_func, DataFrameFormatter

df_formatter = DataFrameFormatter()
add_borders = BorderFormatter()

df_formatter_with_borders = df_formatter + add_borders

@xl_func(formatter=df_formatter_with_borders, auto_resize=True)
@xl_return("dataframe<index=True>")
def get_dataframe():
    df = pd.DataFrame({
        "A": [1, 2, 3],
        "B": [4, 5, 6],
        "C": [0.3, 0.6, 0.9]
    })
    return df
```

3.7.4 Formatting in Macros Functions

Formatters can also be used from macro functions, as well as from worksheet functions.

To apply a formatter in a macro function use the `formatter` option to when setting `XLCell.value`.

For example, to use the standard `DataFrameFormatter` when setting a DataFrame to a range from an Excel macro you would do the following:

```
from pyxll import xl_macro, xl_app, XLCell, DataFrameFormatter
import pandas as pd

@xl_macro
def set_dataframe():
    # Get the current selected cell
    xl = xl_app()
    selection = xl.Selection

    # Get an XLCell instance for the selection
    cell = XLCell.from_range(selection)
```

(continues on next page)

(continued from previous page)

```
# Create a DataFrame
df = pd.DataFrame({
    "A": [1, 2, 3],
    "B": [4, 5, 6],
    "C": [0.3, 0.6, 0.9]
})

# Construct the formatter to be applied
formatter = DataFrameFormatter()

# Set the 'value' on the current cell with the formatter
# and using the auto-resize option
cell.options(type="dataframe<index=True>",
             auto_resize=True,
             formatter=formatter).value = df
```

The same method can be used from a *menu function* or *ribbon action*.

Warning

Formatting cells in Excel uses an Excel macro. Macros in Excel do not preseve the "UnDo" list, and so after any formatting has been applied you will not be able to undo your recent actions.

Warning

Formatting is new in PyXLL 4.5.

For prior versions formatting can be applied using the *Excel Object Model*.

Calls to Excel cannot be made from an `@xl_func` function, but can be scheduled using `schedule_call`.

Note

Formatters applied to Dynamic Array functions make use of *Workbook Metadata* to keep track of formatting applied in order to clear it if the array later contracts.

3.8 Charts and Plotting

As well as using Excel's own charting capabilities, PyXLL allows you to use Python's other plotting libraries within Excel.

PyXLL has support for the following Python plotting libraries, and can be extended to support other via custom code.

3.8.1 Matplotlib

- *Plotting with matplotlib*
- *Using matplotlib.pyplot*
- *Animated Plots*

Plotting with matplotlib

To plot a Matplotlib figure in Excel you first create the figure in exactly the same way you would in any Python script using matplotlib, and then use PyXLL's `plot` function to show it in the Excel workbook.

Plots created using matplotlib are displayed in Excel as images and are not interactive controls.

> **Note**
>
> Using matplotlib with PyXLL requires matplotlib to be installed. This can be done using `pip install matplotlib`, or `conda install matplotlib` if you are using Anaconda.

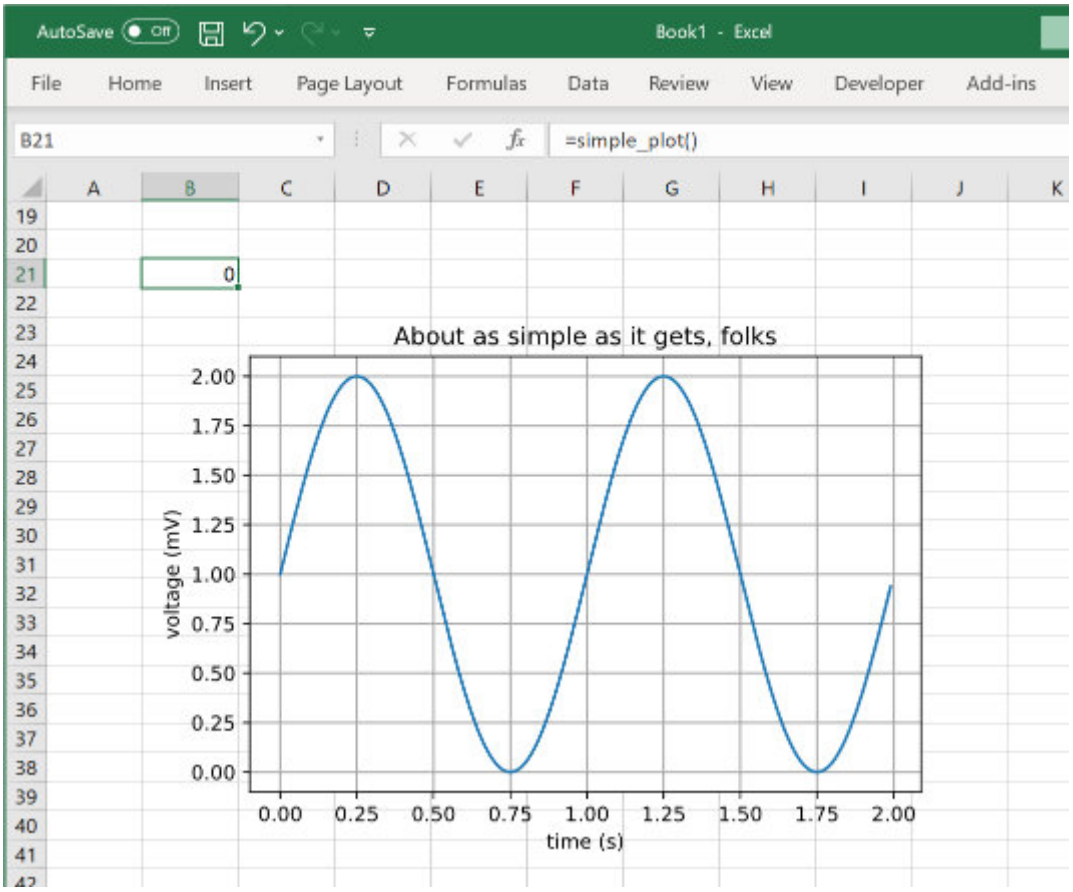
For example, the code below is an Excel worksheet function that generates a matplotlib chart and then displays it in Excel.

```
from pyxll import xl_func, plot
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

@xl_func
def simple_plot():
    # Data for plotting
    t = np.arange(0.0, 2.0, 0.01)
    s = 1 + np.sin(2 * np.pi * t)

    # Create the figure and plot the data
    fig, ax = plt.subplots()
    ax.plot(t, s)

    ax.set(xlabel='time (s)', ylabel='voltage (mV)',
           title='About as simple as it gets, folks')
    ax.grid()

    # Display the figure in Excel
    plot(fig)
```

> **Note**
>
> There is no need to select a backend using `matplotlib.use`. PyXLL will select the backend automatically.

When this function is called from Excel the matplotlib figure is drawn below the cell the function was called from.

The plotting code above was taken from the matplotlib examples. You can find many more examples on the matplotlib website as well as documentation on how to use all of matplotlib's features.



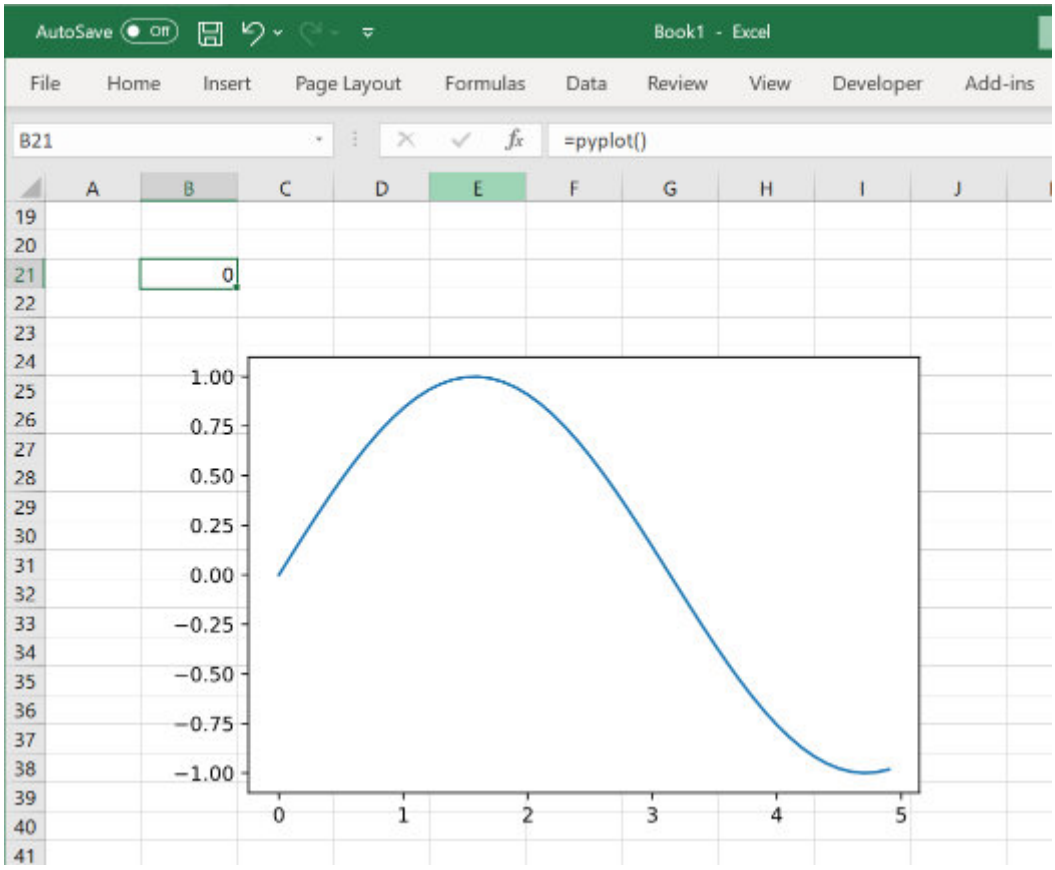
> **Tip**
>
> Whenever you change an input argument to your plotting function, the chart will be redrawn.
>
> You can use this to create interactive dashboards where the Excel user can control the inputs to the plot and see it redraw automatically.

Using matplotlib.pyplot

Pyplot is part of matplotlib and provides a convenient layer for interactive work. If you are more familiar with pyplot and want to use it with PyXLL then that is no problem!

Instead of calling `pyplot.show()` to show the current plot, use *`plot`* without passing a figure and it will show the current plot in Excel. After plotting the current `pyplot` figure is closed.

```
from pyxll import xl_func, plot
import numpy as np
import matplotlib.pyplot as plt

@xl_func
def pyplot():
    # Draw a plot using pyplot
    x = np.arange(0, 5, 0.1);
    y = np.sin(x)
    plt.plot(x, y)

    # Display it in Excel using pyxll.plot
    plot()
```



As with the previous example when this function is called from Excel the plot is shown below the calling cell.

Animated Plots

Matplotlib can be used to create animated plots as well as static ones. These can also be used in Excel with PyXLL.

> **Note**
>
> Support for animated matplotlib plots is new in PyXLL 5.4.0.

Animated plots using matplotlib are created using the `matplotlib.animation.Animation` type. The animation object can be passed to *`plot`* in the same way a `Figure` was used above. The animated plot will be rendered to an animated GIF and embedded in the Excel workbook.

> **Warning**
>
> If you see an error saying that the image cannot be displayed then this will be because your version of Excel is not capable of displaying animated GIFs and you will need to update to a newer version of Excel.

The following code shows how to construct a simple animated plot with matplotlib and display the results in Excel. It can take a small amount of time to render the animation, depending on the number of frames and complexity of the plot.

```
from pyxll import xl_func, plot

from matplotlib.animation import FuncAnimation
from matplotlib import pyplot as plt
```

(continues on next page)

(continued from previous page)

```
import numpy as np


@xl_func
def plot_sine_wave(frequency=1, amplitude=1):
    # Create the matplotlib Figure object, axes and a line
    fig = plt.figure(facecolor='white')
    ax = plt.axes(xlim=(0, 4), ylim=(-2 * amplitude, 2 * amplitude))
    line, = ax.plot([], [], lw=3)

    # The init function is called at the start of the animation
    def init():
        line.set_data([], [])
        return line,

    # The animate function is called for each frame of the animation
    def animate(i):
        x = np.linspace(0, 4, 1000)
        y = np.sin(frequency * 2 * np.pi * (x - 0.01 * i)) * amplitude
        line.set_data(x, y)
        return line,

    # Construct the Animation object
    anim = FuncAnimation(fig,
                         animate,
                         init_func=init,
                         frames=100,
                         interval=20,
                         blit=True)

    # Call pyxll.plot with the Animation object to render the animation
    # and display it in Excel.
    plot(anim)
```

For more information about animated plots in matplotlib please refer to the matplotlib user guide.

3.8.2 Plotting with Pandas

Pandas provides some convenient plotting capabilities based on the matplotlib package. Using pandas to plot DataFrames can be more convenient than using matplotlib directly, and because the result is a matplotlib figure it can be used with PyXLL's *plot* function.

Plots created using matplotlib via pandas are displayed in Excel as images and are not interactive controls.

The `DataFrame.plot` plots using `matplotlib.pyplot` and plots to the current `pyplot` figure. This can then be displayed in Excel using *plot*. When passed no arguments, *plot* plots the current `matplotlib.pyplot` figure and closes it.

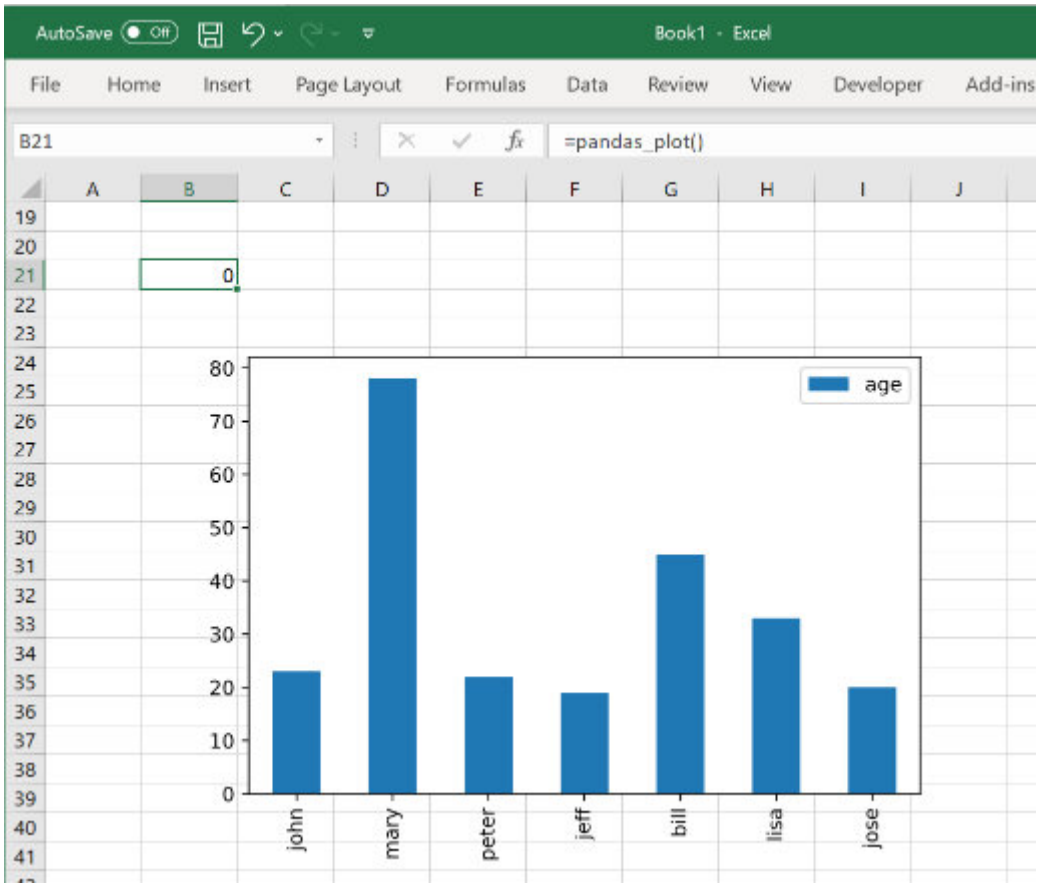
```
from pyxll import xl_func, plot
import pandas as pd

@xl_func
def pandas_plot():
    # Create a DataFrame to plot
    df = pd.DataFrame({
        'name':['john','mary','peter','jeff','bill','lisa','jose'],
        'age':[23,78,22,19,45,33,20],
```

(continues on next page)

(continued from previous page)

```
        'gender':['M','F','M','M','M','F','M'],
        'state':['california','dc','california','dc','california','texas','texas'],
        'num_children':[2,0,0,3,2,1,4],
        'num_pets':[5,1,0,5,2,2,3]
    })

    # A simple bar chart, plotted using matplotlib.pyplot
    df.plot(kind='bar',x='name',y='age')

    # Show the current matplotlib.pyplot figure using pyxll.plot
    plot()
```

As with the previous *matplotlib examples*, when this function is called from Excel the plot is shown below the calling cell.



The Pandas `plot` function optionally takes a `matplotlib.Axes` object. This can be used to plot to a specific Axes object instead of to the current `matplotlib.pyplot` figure. For example, for doing subplots or if you need more control over the `matplotlib.Figure` being used.

```
from pyxll import xl_func, plot
import matplotlib.pyplot as plt
import pandas as pd

    @xl_func
    def pandas_plot():
        # Create a DataFrame to plot
        df = pd.DataFrame({
            'name':['john','mary','peter','jeff','bill','lisa','jose'],
```

(continues on next page)

(continued from previous page)

```
            'age':[23,78,22,19,45,33,20],
            'gender':['M','F','M','M','M','F','M'],
            'state':['california','dc','california','dc','california','texas','texas
↪'],
            'num_children':[2,0,0,3,2,1,4],
            'num_pets':[5,1,0,5,2,2,3]
        })

        # Create the matplotlib Figure and Axes objects
        fig, ax = plt.subplots()

        # Plot a bar chart to the Axes we just created
        df.plot(kind='bar',x='name',y='age', ax=ax)

        # Show the matplotlib Figure created above
        plot(fig)
```

3.8.3 Plotly

To plot a plotly figure in Excel you first create the figure in exactly the same way you would in any Python script using plotly, and then use PyXLL's `plot` function to show it in the Excel workbook.

Plotly supports interactive charts and these are displayed in Excel using an interactive web control, where available[1].

The code below shows an Excel worksheet function that generates a plotly figure displayed it in Excel.

```
from pyxll import xl_func, plot
import plotly.express as px

@xl_func
def plotly_plot():
    # Get some sample data from plotly.express
    df = px.data.gapminder()

    # Create a scatter plot figure
    fig = px.scatter(df.query("year==2007"),
                     x="gdpPercap", y="lifeExp",
                     size="pop", color="continent",
                     log_x=True, size_max=60)

    # Show the figure in Excel using pyxll.plot
    plot(fig)
```

When this function is run in Excel the plot is shown just below the calling cell.

If the interactive web control is not available, the figure will instead be exported as a static image. This is done by PyXLL using plotly's `write_image` method. This requires an additional package *kaleido* to be installed.

To install *kaleido* use `pip install -U kaleido`, or `conda install -c plotly python-kaleido` if you are using Anaconda.

PyXLL also supports using the legacy *orca* package, but from plotly 4.9 onwards it is recommended that you use *kaleido*.

[1] The interactive web control for displaying html based charts is new in PyXLL 5.9.0. In earlier versions the chart will be displayed as a static image that is not interactive.

The web control for displaying html based charts requires the Microsoft `WebView2` component to be installed. This is usually installed as part of the `Microsoft Edge` web browser.

Tip

Whenever you change an input argument to your plotting function, the chart will be redrawn.

You can use this to create interactive dashboards where the Excel user can control the inputs to the plot and see it redraw automatically.

If you have any problems with exporting plots as html or using the interactive web control, you can tell PyXLL to use a static image format instead by passing `allow_html=False` to *plot*.

When exporting as an image and not html, an SVG image may be used. If you version of Excel does not support SVG images (Excel 2016 or earlier) or you are having problems with the SVG image not displaying correctly you can tell PyXLL to use the PNG format instead by passing `allow_svg=False` to *plot*.

Warning

If you are not using the interactive web control and the figure is being exported as an image, plotly launches a kaleido or orca subprocess to do the export.

The first time you export an image from plotly it can take a few seconds.

If you have anti-virus software installed it may warn you about this subprocess being launched.

3.8.4 Seaborn

Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

As Seaborn builds on matplotlib it works perfectly with PyXLL's *plot* function.

All Seaborn plot functions use `matplotlib.pyplot` to plot to the current `pyplot` figure. This can then be displayed in Excel using *plot*. When passed no arguments, *plot* plots the current `matplotlib.pyplot` figure and closes it.

Plots created using Seaborn are displayed in Excel as images and are not interactive controls.

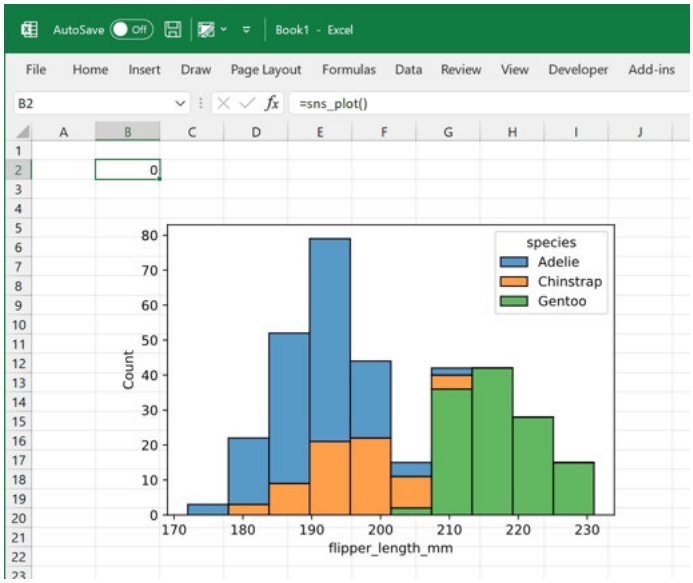
```
from pyxll import plot, xl_func
import seaborn as sns

@xl_func
def sns_plot():
    # Load a dataset to plot
    penguins = sns.load_dataset("penguins")

    # Plot a histogram, plotted to the current matplotlib.pyplot figure
    sns.histplot(data=penguins, x="flipper_length_mm", hue="species", multiple="stack
↪")

    # Show the current matplotlib.pyplot figure using pyxll.plot
    plot()
```

As with the previous *matplotlib examples*, when this function is called from Excel the plot is shown below the calling cell.

The Seaborn plotting functions also optionally take `matplotlib.Axes` objects. This can be used to plot to a specific Axes object instead of to the current `matplotlib.pyplot` figure. For example, for doing subplots or if you need more control over the `matplotlib.Figure` being used.

```
from pyxll import plot, xl_func
import matplotlib.pyplot as plt
```

(continues on next page)



(continued from previous page)

```
import seaborn as sns

@xl_func
def sns_plot():
    # Load a dataset to plot
    penguins = sns.load_dataset("penguins")

    # Create the matplotlib Figure and Axes objects
    fig, ax = plt.subplots()

    # Plot a histogram to the Axes we just created
    sns.histplot(data=penguins, x="flipper_length_mm", hue="species", multiple="stack
↪", ax=ax)

    # Show the matplotlib Figure created above
    plot(fig)
```

Tip

Whenever you change an input argument to your plotting function, the chart will be redrawn.

You can use this to create interactive dashboards where the Excel user can control the inputs to the plot and see it redraw automatically.

3.8.5 Bokeh

To plot a *bokeh* figure in Excel you first create the figure in exactly the same way you would in any Python script using bokeh, and then use PyXLL's `plot` function to show it in the Excel workbook.

Bokeh supports interactive charts and these are displayed in Excel using an interactive web control, where available[1].

The code below shows an Excel worksheet function that generates a bokeh figure and displays it in Excel.

[1] The interactive web control for displaying html based charts is new in PyXLL 5.9.0. In earlier versions the chart will be displayed as a static image that is not interactive.

The web control for displaying html based charts requires the Microsoft `WebView2` component to be installed. This is usually installed as part of the `Microsoft Edge` web browser.

```
# Download the bokeh sample data first
import bokeh
bokeh.sampledata.download()

from math import pi
import pandas as pd
from bokeh.plotting import figure, output_file, show
from bokeh.sampledata.stocks import MSFT

@xl_func
def bokeh_plot():
    # Get some sample data to plot
    df = pd.DataFrame(MSFT)[:50]
    df["date"] = pd.to_datetime(df["date"])

    # Select dates based on open <> close
    inc = df.close > df.open
    dec = df.open > df.close
    w = 12*60*60*1000 # half day in ms

    # Set up the figure
    p = figure(x_axis_type="datetime", plot_width=1000, title="MSFT Candlestick")
    p.xaxis.major_label_orientation = pi/4
    p.grid.grid_line_alpha = 0.3

    # Plot lines for high/low and vbars for open/close
    p.segment(df.date, df.high, df.date, df.low, color="black")
    p.vbar(df.date[inc], w, df.open[inc], df.close[inc], fill_color="#D5E1DD", line_
↪color="black")
    p.vbar(df.date[dec], w, df.open[dec], df.close[dec], fill_color="#F2583E", line_
↪color="black")

    # Show the plot in Excel using pyxll.plot
    plot(p)
```

When this function is run in Excel the plot is shown just below the calling cell.

If the interactive web control is not available, the figure will instead be exported as a static image. This is done using *Selenium* and so that must be installed before Bokeh can be used with PyXLL.

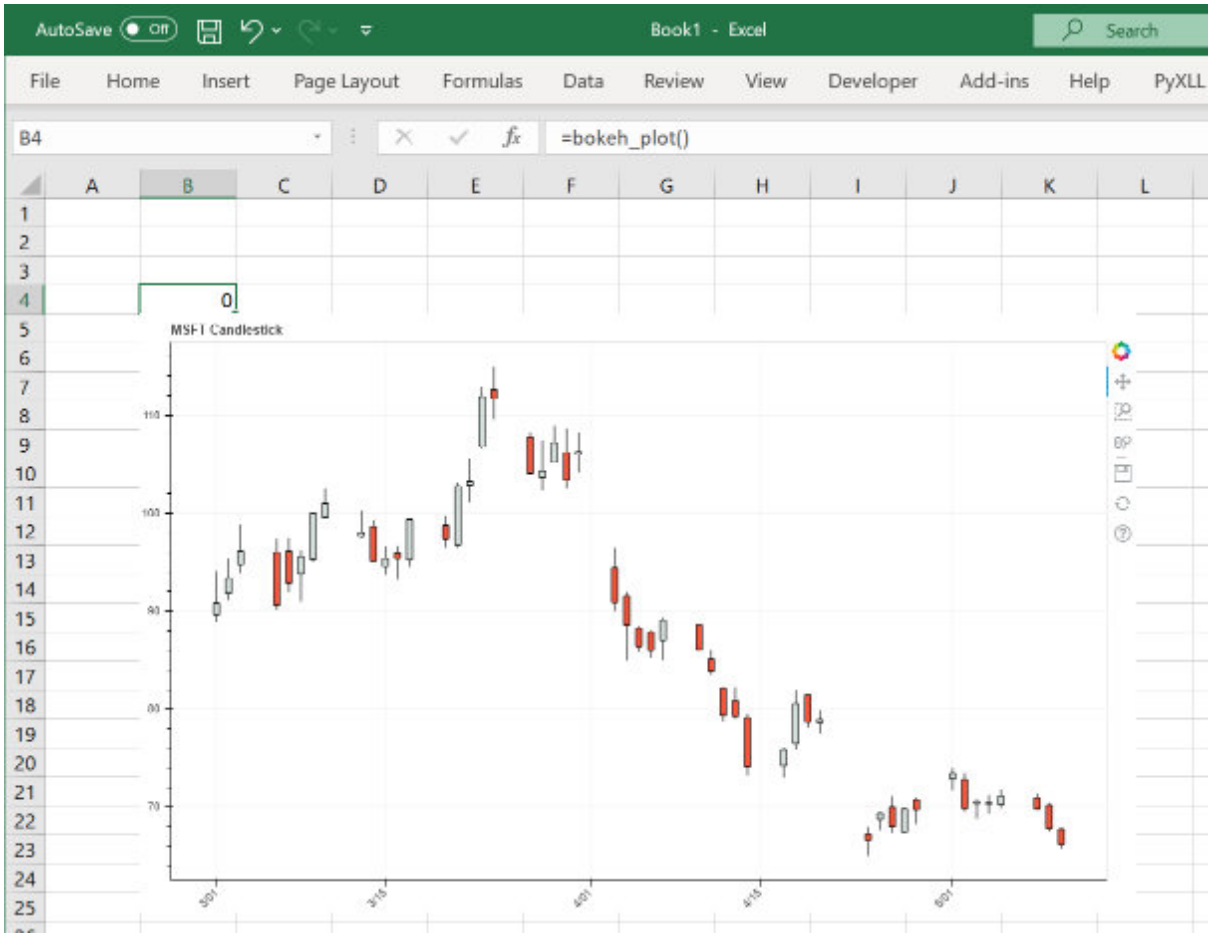
The easiest way to install Selenium is to use Anaconda and install it using either of the following commands:

```
conda install selenium geckodriver firefox -c conda-forge
```

or

```
conda install selenium python-chromedriver-binary -c conda-forge
```

If you are not using Anaconda you can use `pip install selenium` but you will also need to install a suitable web browser backend. See https://pypi.org/project/selenium/ for additional details about how to install Selenium.



Tip

Whenever you change an input argument to your plotting function, the chart will be redrawn.

You can use this to create interactive dashboards where the Excel user can control the inputs to the plot and see it redraw automatically.

If you have any problems with exporting plots as html or using the interactive web control, you can tell PyXLL to use a static image format instead by passing `allow_html=False` to *`plot`*.

When exporting as an image and not html, an SVG image may be used. If you version of Excel does not support SVG images (Excel 2016 or earlier) or you are having problems with the SVG image not displaying correctly you can tell PyXLL to use the PNG format instead by passing `allow_svg=False` to *`plot`*.

Warning

If you are not using the interactive web control and the figure is being exported as an image, bokeh launches a Selenium subprocess to do the export.

The first time you export an image from bokeh it can take a few seconds.

If you have anti-virus software installed it may warn you about this subprocess being launched.

3.8.6 Altair

To plot a *altair* figure in Excel you first create the figure in exactly the same way you would in any Python script using altair, and then use PyXLL's `plot` function to show it in the Excel workbook.

Altair supports interactive charts and these are displayed in Excel using an interactive web control, where available[1].

The code below shows an Excel worksheet function that generates an altair figure and displays it in Excel.

```
# This example requies vega_datasets.
# Install using 'pip install vega_datasets'
from vega_datasets import data
from pyxll import xl_func, plot
import altair as alt
```

(continues on next page)

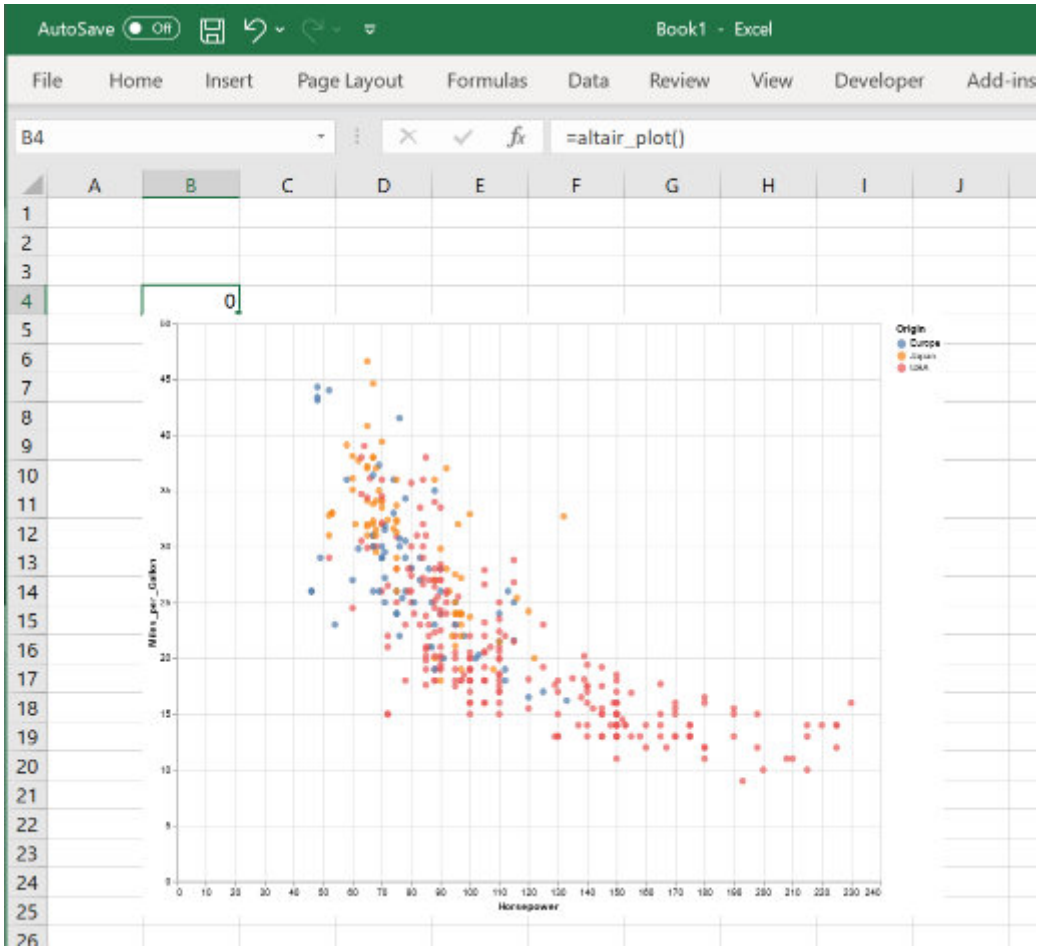
[1] The interactive web control for displaying html based charts is new in PyXLL 5.9.0. In earlier versions the chart will be displayed as a static image that is not interactive.

The web control for displaying html based charts requires the Microsoft `WebView2` component to be installed. This is usually installed as part of the `Microsoft Edge` web browser.

(continued from previous page)

```
@xl_func
def altair_plot():
    # Get the sample data set
    source = data.cars()

    # Create the chart
    chart = alt.Chart(source).mark_circle(size=60).encode(
        x='Horsepower',
        y='Miles_per_Gallon',
        color='Origin'
    )

    # Enable some basic interactive features
    chart = chart.interactive()

    # Show it in Excel using pyxll.plot
    plot(chart)
```

When this function is run in Excel the plot is shown just below the calling cell.



If the interactive web control is not available, the figure will instead be exported as a static image. This is done using *altair_saver* which also requires *Selenium*. **Both** of these must be installed before Altair can be used with PyXLL unless using the web control[Page 139, 1].

- *altair_saver* can be installed using `pip install altair_saver` or `conda install -c conda-forge altair_saver`.
- The easiest way to install Selenium is to use Anaconda and install it using either of the following commands:

```
conda install selenium geckodriver firefox -c conda-forge
```

or

```
conda install selenium python-chromedriver-binary -c conda-forge
```

If you are not using Anaconda you can use `pip install selenium` but you will also need to install a suitable web browser backend. See https://pypi.org/project/selenium/ for additional details about how to install Selenium.

Tip

Whenever you change an input argument to your plotting function, the chart will be redrawn.

You can use this to create interactive dashboards where the Excel user can control the inputs to the plot and see it redraw automatically.

If you have any problems with exporting plots as html or using the interactive web control, you can tell PyXLL to use a static image format instead by passing `allow_html=False` to *`plot`*.

When exporting as an image and not html, an SVG image may be used. If you version of Excel does not support SVG images (Excel 2016 or earlier) or you are having problems with the SVG image not displaying correctly you can tell PyXLL to use the PNG format instead by passing `allow_svg=False` to *`plot`*.

Warning

If you are not using the interactive web control and the figure is being exported as an image, altair launches a Selenium subprocess to do the export.

The first time you export an image from altair it can take a few seconds.

If you have anti-virus software installed it may warn you about this subprocess being launched.

3.8.7 Plotnine

Plotnine is a Python implementation of a grammar of graphics, based on and compatible with the syntax of R's *ggplot2*.

To plot a plotnine chart in Excel you first create the `ggplot` object in exactly the same way you would in any Python script using plotnine, and then use PyXLL's *`plot`* function to show it in the Excel workbook.

```
from pyxll import xl_func, plot
from plotnine import ggplot, aes, geom_point
from plotnine.data import mtcars

@xl_func
def plotnine_plot():
    # Create the ggplot object
    chart = (
        ggplot(mtcars, aes(x="wt", y="mpg", color="factor(cyl)"))
        + geom_point()
    )

    # Show it in Excel using pyxll.plot
    plot(chart)
```

When this function is run in Excel the plot is shown just below the calling cell.

Unlike plotly, bokeh and altair, plotnine charts are always displayed in Excel as static images and not as interactive web controls, since plotnine has no interactive HTML output of its own.

If you set an explicit figure size using plotnine's `theme(figure_size=...)` that size is used as the initial size of the image in Excel. Otherwise the default image size is used, in the same way as for a matplotlib figure.

```
from pyxll import xl_func, plot
from plotnine import ggplot, aes, geom_point, theme

@xl_func
def plotnine_plot():
    chart = (
        ggplot(mtcars, aes(x="wt", y="mpg", color="factor(cyl)"))
        + geom_point()
        + theme(figure_size=(8, 6))
    )

    plot(chart)
```

When exporting as an image an SVG image may be used. If your version of Excel does not support SVG images (Excel 2016 or earlier), or you are having problems with the SVG image not displaying correctly, you can tell PyXLL to use the PNG format instead by passing `allow_svg=False` to *`plot`*.

Tip

Whenever you change an input argument to your plotting function, the chart will be redrawn.

You can use this to create interactive dashboards where the Excel user can control the inputs to the plot and see it redraw automatically.

Note

Support for plotnine charts was added in PyXLL 5.13.

3.8.8 Other Plotting Packages

PyXLL provides support for *matplotlib* (including *pyplot* and *pandas*), *plotly*, *bokeh*, *altair* and *plotnine*.

If you want to use another Python plotting package that's not already supported then you can. To do so you need to provide you own implementation of PyXLL's *`PlotBridgeBase`* class.

The *Plot Bridge* is what PyXLL uses to export the chart or figure to an image, and so long as the plotting library you want to use can export to SVG or PNG format you can write a plot bridge class to use it in PyXLL.

See the API reference for *`PlotBridgeBase`* for details of the methods you need to implement.

Once you have implemented your Plot Bridge you pass it to *`plot`* as the `bridge_cls` keyword argument. Whatever object you pass as the figure to *`plot`* will be used to construct your Plot Bridge object, which will be used to export the figure to an image. PyXLL will take care of the rest of inserting or updating that image in Excel.

Using Python's plotting packages is preferable to using Excel's own charts in some situations.

- You can plot directly from Python and so this can reduce the need to return a lot of data to Excel and make your sheets smaller and simpler.
- Using the Python plotting libraries gives you more control over how your charts appear and gives you access to chart types that are not available using Excel's own chart types.

Tip

Some plotting libraries such as `plotly` and `bokeh` produce web-based interactive charts[1]. PyXLL will display these charts using a web control, where possible[2]. When the web control is used, the charts are interactive in Excel in the same way as they are when shown in a browser.

To show a plot or chart in Excel you use whichever Python plotting library you prefer to generate the chart and then use PyXLL's `plot` function to render it to Excel. See the individual guides linked above for specific instructions for each.

You can plot directly from an Excel worksheet function decorated with `@xl_func`, and so you can provide your own inputs to your plotting function. These can be used to let the user of your function have some control over how the chart is plotted to make it interactive. Each time they change an input the plot will be re-drawn.

Depending of which plotting library you use the plot itself will be either be inserted into Excel as an interactive web control[1], or as a static image.

Note

Depending on the version of Excel you are using and the plotting library, the chart may be exported as an HTML document[1] or an SVG image when plotting to Excel.

If you are experiencing problems with HTML charts, or the web viewer control, you can disable exporting as HTML by passing `allow_html=False` to `plot`. This will cause it to export the image in a static image format instead.

Some plotting libraries can occasionally show problems when plotting to SVG. If you see any visual errors (for example, borders being too thick or the background color showing through) you can set `allow_svg=False` when calling `plot`. This will cause it to export the image in a bitmap format instead.

3.8.9 Plotting from Worksheet Functions

When you use `plot` from an Excel worksheet function using `@xl_func` the image inserted into the Excel workbook will be placed just below the cell the function is being called from.

Additionally, subsequent calls to the same function will replace the image rather than creating a new one each time the function is called. This is done by giving the image a unique name for the calling cell. If you perform multiple plots from the same function you will need to pass a name for each to the `plot` function.

Tip

See *Plotting Settings* for the config options available to customize how the `plot` function behaves.

3.8.10 Plotting from Menus, Macros and Elsewhere

The `plot` function when called from anywhere other than a worksheet function will always add a new image to the Excel workbook. By default, the location of the image will be just underneath the currently selected cell.

If you want to replace an existing image rather than add a new one, use the `name` argument to `plot` and when plotting an image with the same name multiple times the existing image in Excel will be replaced instead of creating a new one.

[1] The web control for displaying html based charts is new in PyXLL 5.9.0. In earlier versions the chart will be displayed as a static image that is not interactive.

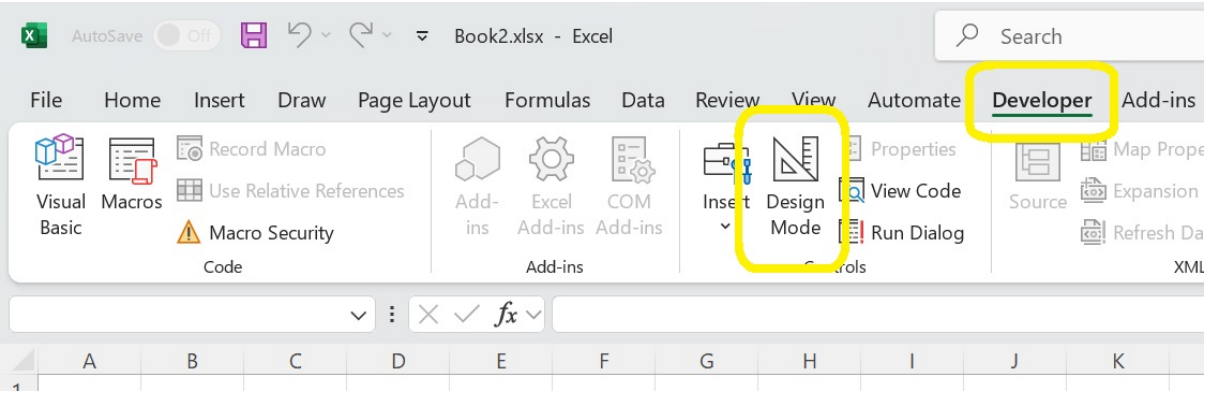
[2] The web control for displaying html based charts requires the Microsoft `WebView2` component to be installed. This is usually installed as part of the `Microsoft Edge` web browser.

3.8.11 Moving and Resizing

Interactive HTML plots[Page 143, 1] use a special web control widget. To move or resize these plots your first need to enable `Design Mode`.

To enable Design Mode, go to the `Developer` tab in the Excel ribbon and select `Design Mode`. Whilst in Design Mode a bitmap preview will be displayed instead of the web control. You can now move and resize this shape. There may be some lag between resizing the preview image and the preview image updating.

To return to the interactive web widget, unselect Design Mode.



Other plot types use a Picture shape, which can be moved and resized without entering Design Mode.

New in PyXLL 5.7

After resizing a plot in Excel, when you change the selection to something else (e.g. click off the plot and into another cell) the figure will be redrawn to the new size of the image displayed in Excel.

This can be disabled by passing `allow_resize=False` to the `plot` function, or by setting the following in your pyxll.cfg file

```
[PYXLL]
plot_auto_resize = 0
```

The above only applies for plots that render as a static image and not as an interactive HTML plot.

3.9 Excel Application Events

New in PyXLL 5.12

The `@xl_event` decorators are used to register Python functions that will be called when specific Excel Application events occur.

The events that can be handled are the same as the Excel Application events available through VBA, and the Python functions have the same arguments as their VBA counterparts.

- *Registering Event Handlers*
- *List of Events*
- *Event Arguments*
- *Canceling Events*
- *Execution Context*
- *Registration and Reloading*
- *Event Filters*

3.9.1 Registering Event Handlers

Excel Application events can be handled using Python function decorated with one of the *@xl_event* decorators.

When an Excel Application event fires, all the registered event handler functions will be called. When there are multiple event handler functions registered for the same event the order in which the registered functions will be called is not guaranteed.

Event handler signatures mirror the VBA Application event signatures. Some events pass no arguments. Refer to *Excel Application Events* and the Excel VBA Object Browser for the precise argument list.

The following code registers a function to be called whenever the Excel Application `AfterCalculate` event is fired:

```
from pyxll import xl_event

@xl_event("AfterCalculate")
def on_after_calculate():
    pass
```

To simplify registering Excel event handlers, helpers are provided for each of the Excel Application events so that the event name can be omitted.

If you use auto-completion in your Python IDE then you will prefer to use these helpers over the event name. Elsewhere in these docs you will see the helpers used in preference to the named form, but both are equivalent.

Each shortcut decorator uses the underscore naming style instead of the VBA camel case style.

For example, the below is equivalent to the code above, but using the `@xl_event.after_calculate` helper in place of `@xl_event("AfterCalculate")`

```
from pyxll import xl_event

@xl_event.after_calculate
def on_after_calculate():
    print("Excel has been calculated")
```

3.9.2 List of Events

All of the Excel Application events are available through the *@xl_event* decorator and helper decorators.

For a complete list of available events please see the API documentation *Excel Application Events*.

- *@xl_event.new_workbook*
- *@xl_event.sheet_selection_change*
- *@xl_event.sheet_before_double_click*
- *@xl_event.sheet_before_right_click*
- *@xl_event.sheet_activate*
- *@xl_event.sheet_deactivate*
- *@xl_event.sheet_calculate*
- *@xl_event.sheet_change*
- *@xl_event.workbook_open*
- *@xl_event.workbook_activate*
- *@xl_event.workbook_deactivate*
- *@xl_event.workbook_before_close*
- *@xl_event.workbook_before_save*

- *@xl_event.workbook_before_print*
- *@xl_event.workbook_new_sheet*
- *@xl_event.workbook_addin_install*
- *@xl_event.workbook_addin_uninstall*
- *@xl_event.window_resize*
- *@xl_event.window_activate*
- *@xl_event.window_deactivate*
- *@xl_event.sheet_follow_hyperlink*
- *@xl_event.sheet_pivot_table_update*
- *@xl_event.workbook_pivot_table_close_connection*
- *@xl_event.workbook_pivot_table_open_connection*
- *@xl_event.workbook_sync*
- *@xl_event.workbook_before_xml_import*
- *@xl_event.workbook_after_xml_import*
- *@xl_event.workbook_before_xml_export*
- *@xl_event.workbook_after_xml_export*
- *@xl_event.workbook_rowset_complete*
- *@xl_event.after_calculate*
- *@xl_event.sheet_pivot_table_after_value_change*
- *@xl_event.sheet_pivot_table_before_allocate_changes*
- *@xl_event.sheet_pivot_table_before_commit_changes*
- *@xl_event.sheet_pivot_table_before_discard_changes*
- *@xl_event.protected_view_window_open*
- *@xl_event.protected_view_window_before_edit*
- *@xl_event.protected_view_window_before_close*
- *@xl_event.protected_view_window_resize*
- *@xl_event.protected_view_window_activate*
- *@xl_event.protected_view_window_deactivate*
- *@xl_event.workbook_after_save*
- *@xl_event.workbook_new_chart*

3.9.3 Event Arguments

The Excel Application events are the same as the Application events available through VBA, and are documented in the VBA Object Browser in Excel, as well as online in Microsoft's documentation.

For event handler functions that take Excel objects as arguments, these arguments are passed as COM objects using the configured Python COM package. Supported Python COM packages are pywin32 (the default option) or comtypes.

The following example shows how to write a function that handles the `Application.SheetActivate` event. In VBA, this event has a `Sheet` argument, and in Python the sheet is passed as a Sheet COM object using the default Python COM package.

```
from pyxll import xl_event

@xl_event.sheet_activate
def on_sheet_activated(sheet):
    """This handles the Excel Workbook.SheetActivate event and is passed a COM Sheet␣
↪object."""

    # The sheet is an Excel Sheet object as has the same properties as the␣
↪corresponding VBA Sheet object.
    sheet_name = sheet.Name

    # Print a message to show the event has been handled.
    print("EXAMPLE: Workbook.SheetActivate event for '%s' handled by %s" % (sheet_
↪name, __file__))
```

To specify which Python COM package should be used, use the *com_package* kwargs to `@xl_event`.

The following example is the same as the above, but using the `comtypes` package.

```
from pyxll import xl_event

@xl_event.sheet_activate(com_package="comtypes")
def on_sheet_activated(sheet):
    """This handles the Excel Workbook.SheetActivate event and is passed a comtypes␣
↪Sheet object."""

    # The sheets is a ``comtypes`` COM object corresponding to the Excel Sheet object␣
↪type
    sheet_name = sheet.Name

    # Print a message to show the event has been handled.
    print("EXAMPLE: Workbook.SheetActivate event for '%s' handled by %s" % (sheet_
↪name, __file__))
```

The default `com_package` can be selected by setting `com_package` in the `[PYXLL]` section of the `pyxll.cfg` config file.

3.9.4 Canceling Events

New in PyXLL 5.13

Some event handlers take a `cancel` argument. For example, `@xl_event.workbook_before_close`. Canceling an event prevents Excel from continuing once all the registered handlers for that event have run. When there are multiple handlers registered for the same event, each one is passed whatever `cancel` value was left by the handlers called before it, and can change that value for the handlers called after it.

In VBA, the cancel argument is passed by reference and the value can be modified to change the cancel state of the event. In Python, the `cancel` argument is passed to the event handler as a plain `bool` and cannot be modified in place.

Instead, to set the cancel state from a Python event handler the handler should return either `True` or `False`. The event's cancel state is updated to the returned value, and so to cancel an event the handler should return `True`. If the return value is `None`, or any value other than `True` or `False`, the cancel state is left unchanged.

```
from pyxll import xl_event

# List of names of workbooks that we don't want to be closed.
DONT_CLOSE_WB_NAMES = {...}
```

(continues on next page)

(continued from previous page)

```
@xl_event.workbook_before_close
def on_workbook_before_close(wb, cancel):
    """This handles the Excel WorkbookBeforeClose event."""
    # If the workbook is one of our named workbooks that should be
    # closed, return True to prevent the workbook from closing.
    if wb.Name in DONT_CLOSE_WB_NAMES:
        print(f"Not closing workbook %s" % wb.Name)
        return True

    # Otherwise we can return the current cancel state.
    # Returning None here would also leave the current state unchanged.
    return cancel
```

3.9.5 Execution Context

Event handlers run in Excel's main thread in a macro-context. It is safe to use *xl_app* and Excel C API functions inside an event handler.

Event handlers may be called while Excel is busy. To avoid slowing down Excel, keep handlers fast and avoid long running work.

If you need to do heavier processing, schedule it on a background thread. Remember however that you cannot call back into Excel from a background thread and must use *schedule_call* if you need to call into Excel.

Exceptions raised by event handlers are logged and do not prevent other handlers from running. Use logging in handlers to aid troubleshooting.

3.9.6 Registration and Reloading

Event handlers are registered when their modules are loaded by PyXLL.

Handlers are un-registered and re-registered when the module they're declared in is reloaded.

3.9.7 Event Filters

When registering a function as an event handler, by default, that function will be called whenever the Excel event is fired.

Sometimes you might only want your handler to be called for some events. For example, if you have a `SheetSelectionChange` event you might want to only do something if the sheet belongs to a specific workbook.

All *@xl_event* decorators take a `filter` kwarg. This can be set to another function that takes the same arguments as the event handler. It will be called before the event handler, and the event handler will only be called if the filter function returns `True`.

Filters run on every event fire and will block Excel. Keep them lightweight to avoid performance problems.

For example, the following filter function returns `True` if the worksheet belongs to a workbook with a specific name:

```
from pyxll import xl_event

# The filter function has the same signature as the event handler
def sheet_in_correct_workbook(sheet):
    # sheet is an Excel Worksheet COM object
    book = sheet.Parent

    # return True if the parent workbook name is what we're looking for
    return book.Name == "MyBookName.xlsx"
```

(continues on next page)

(continued from previous page)

```
@xl_event.sheet_activate(filter=sheet_in_correct_workbook)
def on_sheet_activate(sheet):
    # This only gets called if the filter function returns True
    print(f"Sheet activated: {sheet.Name}")
```

Using a filter function as above avoids cluttering your event handler with conditional code that you might like to reuse between event handlers.

A common pattern is to use a more generic factory function to return a filter function. We could re-write the above example with a filter function factory so that it can be reused for different workbook names as follows:

```
from pyxll import xl_event

def sheet_in_workbook(workbook_name):
    """Return a filter function that takes a Sheet object."""
    # This is the function our outer function will return and is
    # what will be used as our filter function.
    def sheet_in_correct_workbook(sheet):
        # sheet is an Excel Worksheet COM object
        book = sheet.Parent

        # return True if the parent workbook name matched our chosen 'workbook_name'
        return book.Name == workbook_name

    # Return the filter function that is using our workbook_name
    return sheet_in_correct_workbook

# Now we can specify the workbook name when creating the filter function
@xl_event.sheet_activate(filter=sheet_in_workbook("MyBookName.xlsx"))
def on_sheet_activate(sheet):
    # This only gets called if the filter function returns True
    print(f"Sheet activated: {sheet.Name}")
```

3.10 Custom Task Panes

Python UI controls can be embedded into Excel *Custom Task Panes* so they seamlessly fit in with the rest of the Excel user interface.

> **Note**
>
> For user interface controls that are directly on the worksheet, see *ActiveX Controls*.

PyXLL has support for the following Python UI tookits.

3.10.1 PySide and PyQt

PySide and *PyQt* are both Python packages wrapping the popular *Qt* UI toolkit. They are quite similar but have different licenses and so which one you choose will be down to your own preference. Both work equally well with PyXLL.

User interfaces developed using Qt can be hosted in Excel Custom Task Panes as docked or floating windows within Excel. If instead you want the control embedded directly on the Excel worksheet, see *ActiveX Controls*.

This document is not a guide to use PySide or PyQt. It is only intended to instruct you on how to use PySide and PyQt with the Custom Task Pane feature of PyXLL. You should refer to the relevant package documentation for details of how to use each package.

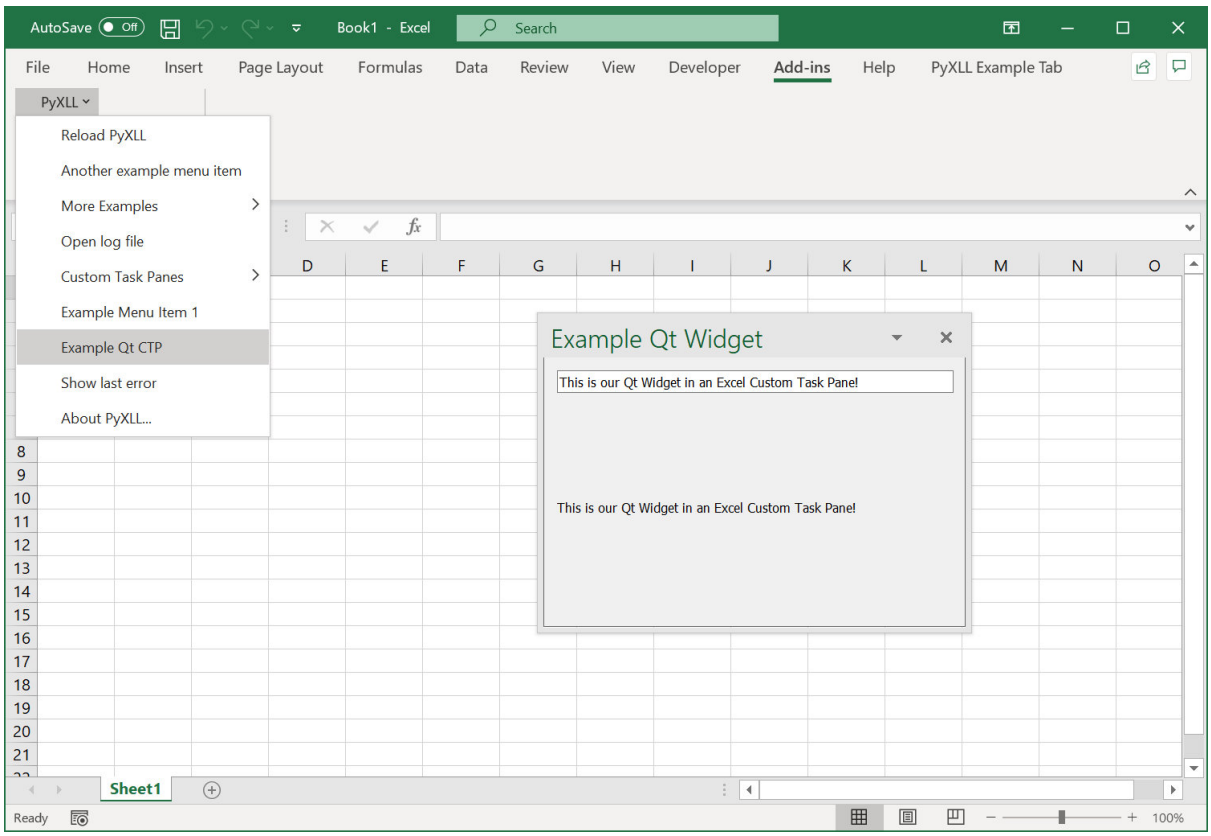
Both PySide and PyQt can be installed using pip or conda, for example:

```
> pip install pyside6
# or
> pip install pyqt6
# or
> conda install pyside6
# or
> conda install "pyqt>=6"
```

Typically you will only want to install one or the other, and you should install it using pip *or* conda and not both.

You can find more information about PySide and PyQt on the websites, https://wiki.qt.io/Qt_for_Python and https://www.riverbankcomputing.com/software/pyqt/ respectively.

> **Note**
>
> Qt6 support was added in PyXLL 5.1, for both PySide6 and PyQt6.
>
> Any of PySide2, PySide6, PyQt5 and PyQt6 can be used with PyXLL.



Creating a Qt Widget

One of the main classes in Qt is the `QWidget` class. To create your own user interface it is this `QWidget` class that you will use, and it's what PyXLL will embed into Excel as a Custom Task Pane.

The following code demonstrates how to create simple Qt widget. If you run this code as a Python script then you will see the widget being shown.

```
from PySide6 import QtWidgets
# or from PyQt6 import QtWidgets
```

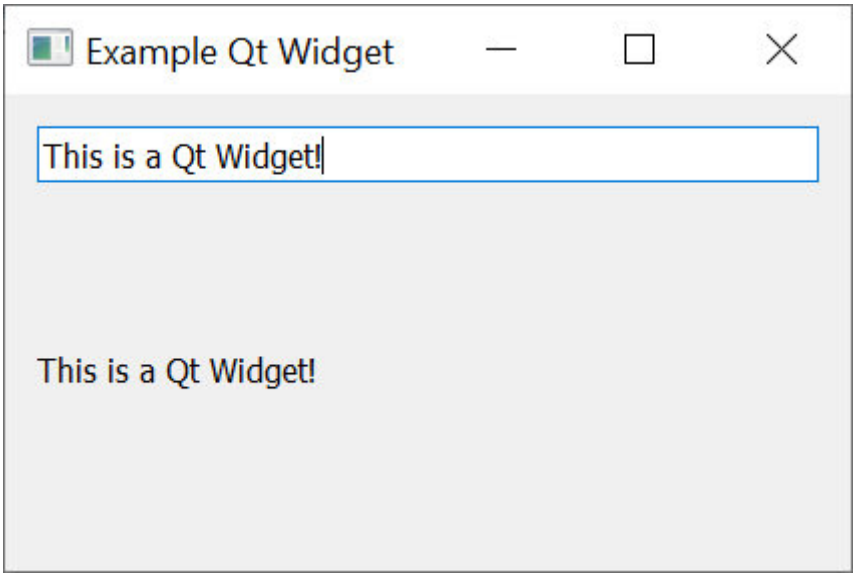
(continues on next page)

(continued from previous page)

```
import sys

class ExampleWidget(QtWidgets.QWidget):

    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        """Initialize the layout and child controls for this widget."""
        # Give the widget a title
        self.setWindowTitle("Example Qt Widget")

        # Create a "Layout" object to help layout the child controls.
        # A QVBoxLayout lays out controls vertically.
        vbox = QtWidgets.QVBoxLayout(self)

        # Create a QLineEdit control and add it to the layout
        self.line_edit = QtWidgets.QLineEdit(self)
        vbox.addWidget(self.line_edit)

        # Create a QLabel control and add it to the layout
        self.label = QtWidgets.QLabel(self)
        vbox.addWidget(self.label)

        # Connect the 'textChanged' event to our 'onChanged' method
        self.line_edit.textChanged.connect(self.onChanged)

        # Set the layout for this widget
        self.setLayout(vbox)

    def onChanged(self, text):
        """Called when the QLineEdit's text is changed"""
        # Set the text from the QLineEdit control onto the label control
        self.label.setText(text)
        self.label.adjustSize()

if __name__ == "__main__":
    # Create the Qt Application
    app = QtWidgets.QApplication(sys.argv)

    # Create our example widget and show it
    widget = ExampleWidget()
    widget.show()

    # Run the Qt app
    sys.exit(app.exec_())
```

When you run this code you will see our example widget being display, and as you enter text into the line edit control the label below will be updated.

Next we'll see how we can use this widget in Excel.



Creating a Custom Task Pane from a Qt Widget

To show a QWidget in Excel using PyXLL we use the `create_ctp` function.

As above, before we can create the widget we have to make sure the QApplication has been initialized. Unlike the above script, our function may be called many times and so we don't want to create a new application each time and so we check to see if one already exists.

The QApplication object must still exist when we call `create_ctp`. If it has gone out of scope and been released then it will cause problems later so always make sure to keep a reference to it.

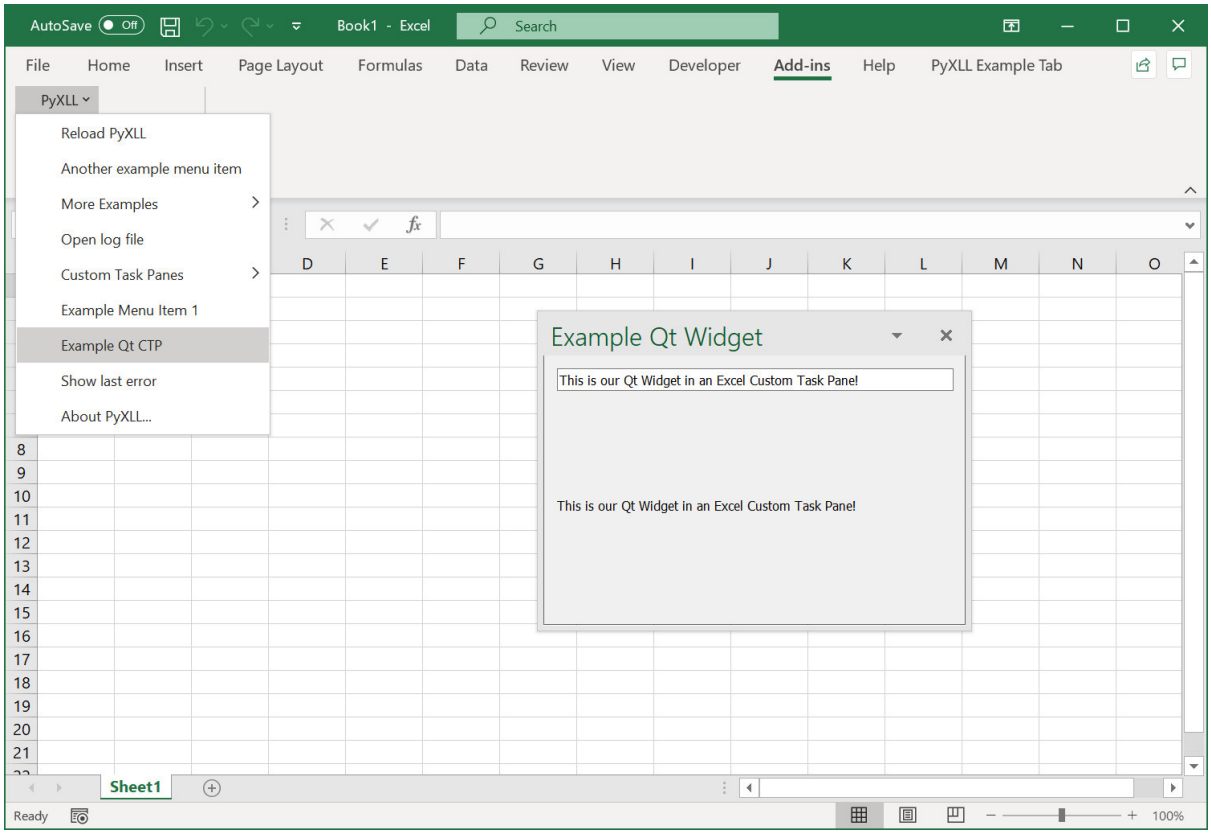
We can create the Custom Task Pane from many different places, but usually it will be from a *ribbon function* or a *menu function*.

The following code shows how we would create a custom task pane from an Excel menu function, using the `ExampleWidget` control from the example above.

```
from pyxll import xl_menu, create_ctp, CTPDockPositionFloating
from PySide6 import QtWidgets
# or from PyQt5 import QtWidgets

@xl_menu("Example Qt CTP")
def example_qt_ctp():
    # Before we can create a Qt widget the Qt App must have been initialized.
    # Make sure we keep a reference to this until create_ctp is called.
    app = QtWidgets.QApplication.instance()
    if app is None:
        app = QtWidgets.QApplication([])

    # Create our example Qt widget from the code above
    widget = ExampleWidget()

    # Use PyXLL's 'create_ctp' function to create the custom task pane.
    # The width, height and position arguments are optional, but for this
    # example we'll create the CTP as a floating widget rather than the
    # default of having it docked to the right.
    create_ctp(widget,
               width=400,
               height=400,
               position=CTPDockPositionFloating)
```

When we add this code to PyXLL and reload the new menu function "Example Qt CTP" will be available, and when that menu function is run the ExampleWidget is opened as a Custom Task Pane in Excel.

Unlike a modal dialog, a Custom Task Pane does not block Excel from functioning. It can be moved and resized, and even docked into the current Excel window in exactly the same way as the native Excel tools.



See the API reference for `create_ctp` for more details.

3.10.2 wxPython

wxPython is a Python packages that wraps the UI toolkit *wxWindows*.

This document is not a guide to use wxPython or wxWindows. It is only intended to instruct you on how to use wxPython with the Custom Task Pane feature of PyXLL. You should refer to the relevant package documentation for details of how to use wxPython and wxWindows.

Both wxWindows can be installed using pip or conda, for example:

```
> pip install wxpython
# or
> conda install wxpython
```

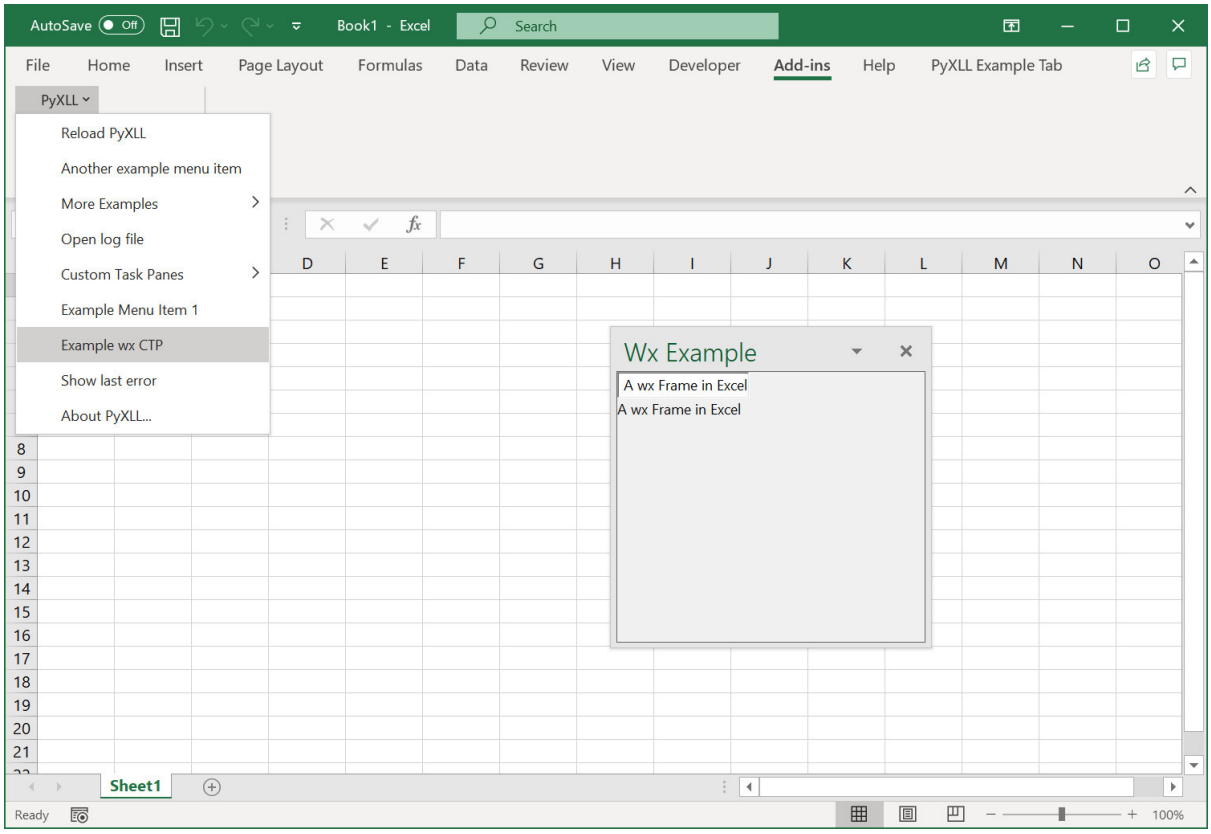
You should install it using pip *or* conda and not both.

You can find more information about wxPython on the website https://www.wxpython.org/.

Creating a wx Frame

Two of the main classes we'll use in wxPython are the `wx.Frame` and `wx.Panel` classes.

A `wx.Frame` is the main window type, and it's this that you'll create to contain your user interface that will be embedded into Excel as a Custom Task Panel. Frames typically host a single `wx.Panel` which is where all the controls that make up your user interface will be placed.



The following code demonstrates how to create simple wx.Frame and corresponding wx.Panel. If you run this code as a Python script then you will see the frame being shown.

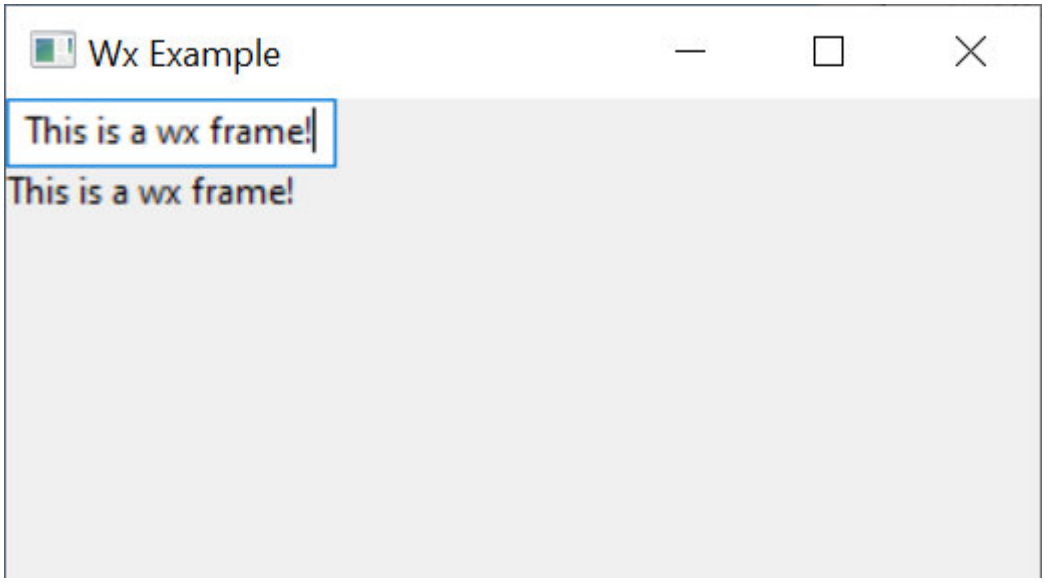
```
import wx

class ExamplePanel(wx.Panel):

    def __init__(self, parent):
        super().__init__(parent=parent)

        # Create a sizer that will lay everything out in the panel.
        # A BoxSizer can arrange controls horizontally or vertically.
        sizer = wx.BoxSizer(orient=wx.VERTICAL)

        # Create a TextCtrl control and add it to the layout
        self.text_ctrl = wx.TextCtrl(self)
        sizer.Add(self.text_ctrl)

        # Create a StaticText control and add it to the layout
        self.static_text = wx.StaticText(self)
        sizer.Add(self.static_text)

        # Connect the 'EVT_TEXT' event to our 'onText' method
        self.text_ctrl.Bind(wx.EVT_TEXT, self.onText)

        # Set the sizer for this panel and layout the controls
        self.SetSizer(sizer)
        self.Layout()

    def onText(self, event):
```

(continues on next page)

(continued from previous page)

```
        """Called when the TextCtrl's text is changed"""
        # Set the text from the event onto the static_text control
        text = event.GetString()
        self.static_text.SetLabel(text)

class ExampleFrame(wx.Frame):

    def __init__(self):
        super().__init__(parent=None)

        # Give this frame a title
        self.SetTitle("Wx Example")

        # Create the panel that contains the controls for this frame
        self.panel = ExamplePanel(parent=self)


if __name__ == "__main__":
    # Create the wx Application object
    app = wx.App()

    # Construct our example Frame and show it
    frame = ExampleFrame()
    frame.Show()

    # Run the application's main event loop
    app.MainLoop()
```

When you run this code you will see our example frame being display, and as you enter text into the text control the static text below will be updated.



Next we'll see how we can use this frame in Excel.

Creating a Custom Task Pane from a wx.Frame

To show a wx.Frame in Excel using PyXLL we use the `create_ctp` function.

As above, before we can create the frame we have to make sure the wx.App application object has been initialized. Unlike the above script, our function may be called many times and so we don't want to create a new application

each time and so we check to see if one already exists.

The wx.App object must still exist when we call `create_ctp`. If it has gone out of scope and been released then it will cause problems later so always make sure to keep a reference to it.

We can create the Custom Task Pane from many different places, but usually it will be from a *ribbon function* or a *menu function*.

The following code shows how we would create a custom task pane from an Excel menu function, using the `ExampleFrame` control from the example above.

```
from pyxll import xl_menu, create_ctp, CTPDockPositionFloating
import wx

@xl_menu("Example wx CTP")
def example_wx_ctp():
    # Before we can create a wx.Frame the wx.App must have been initialized.
    # Make sure we keep a reference to this until create_ctp is called.
    app = wx.App.Get()
    if app is None:
        app = wx.App()

    # Create our example frame from the code above
    frame = ExampleFrame()

    # Use PyXLL's 'create_ctp' function to create the custom task pane.
    # The width, height and position arguments are optional, but for this
    # example we'll create the CTP as a floating window rather than the
    # default of having it docked to the right.
    create_ctp(frame,
               width=400,
               height=400,
               position=CTPDockPositionFloating)
```

When we add this code to PyXLL and reload the new menu function "Example wx CTP" will be available, and when that menu function is run the ExampleFrame is opened as a Custom Task Pane in Excel.

Unlike a modal dialog, a Custom Task Pane does not block Excel from functioning. It can be moved and resized, and even docked into the current Excel window in exactly the same way as the native Excel tools.

See the API reference for `create_ctp` for more details.

3.10.3 Tkinter

tkinter is a Python packages that wraps the *Tk GUI toolkit*.

tkinter is included with Python and so is available to use without needing to install any additional packages.

User interfaces developed using Tk can be hosted in Excel Custom Task Panes as docked or floating windows within Excel. If instead you want the control embedded directly on the Excel worksheet, see *ActiveX Controls*.

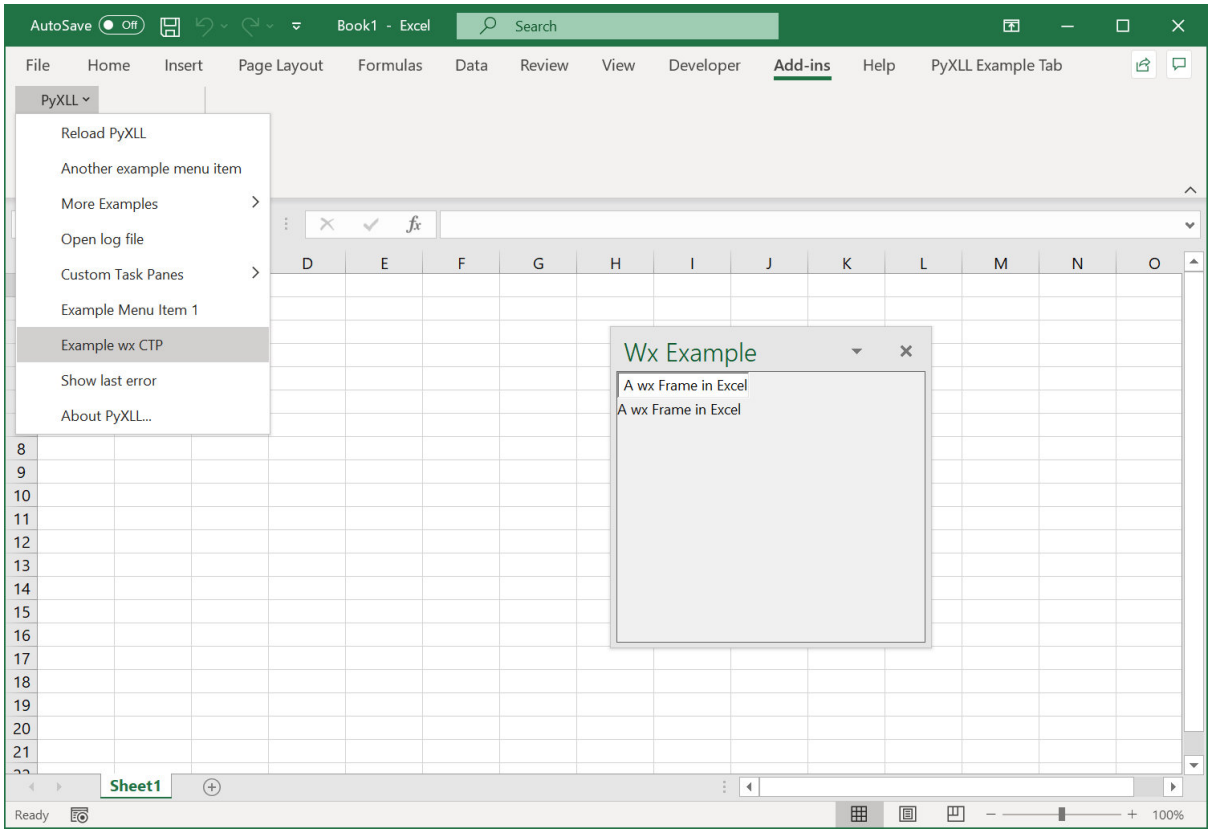
This document is not a guide to use tkinter. It is only intended to instruct you on how to use Tkinter with the Custom Task Pane feature of PyXLL. You should refer to the tkinter documentation for details of how to use tkinter.

You can find more information about tkinter in the Python docs website https://docs.python.org/3/library/tkinter.html.

Creating a tk Frame

One of the main classes in tkinter is the `Frame` class. To create your own user interface it is this `Frame` class that you will use, and it's what PyXLL will embed into Excel as a Custom Task Pane.

The following code demonstrates how to create simple `tkinter.Frame`. If you run this code as a Python script then you will see the frame being shown.



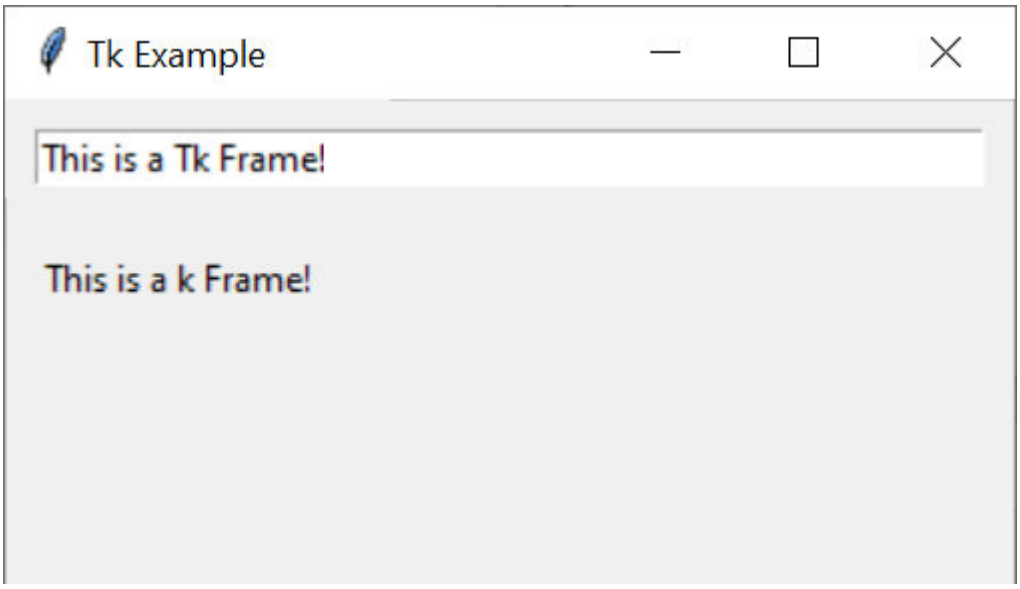

```
import tkinter as tk

class ExampleFrame(tk.Frame):

    def __init__(self, master):
        super().__init__(master)
        self.initUI()

    def initUI(self):
        # allow the widget to take the full space of the root window
        self.pack(fill=tk.BOTH, expand=True)

        # Create a tk.Entry control and place it using the 'grid' method
        self.entry_value = tk.StringVar()
        self.entry = tk.Entry(self, textvar=self.entry_value)
        self.entry.grid(column=0, row=0, padx=10, pady=10, sticky="ew")

        # Create a tk.Label control and place it using the 'grid' method
        self.label_value = tk.StringVar()
        self.label = tk.Label(self, textvar=self.label_value)
        self.label.grid(column=0, row=1, padx=10, pady=10, sticky="w")

        # Bind write events on the 'entry_value' to our 'onWrite' method
        self.entry_value.trace("w", self.onWrite)

        # Allow the first column in the grid to stretch horizontally
        self.columnconfigure(0, weight=1)

    def onWrite(self, *args):
```

(continues on next page)

(continued from previous page)

```
        """Called when the tk.Entry's text is changed"""
        # Update the label's value to be the same as the entry value
        self.label_value.set(self.entry_value.get())

if __name__ == "__main__":
    # Create the root Tk object
    root = tk.Tk()

    # Give the root window a title
    root.title("Tk Example")

    # Construct our frame object
    ExampleFrame(master=root)

    # Run the tk main loop
    root.mainloop()
```

When you run this code you will see our example frame being display, and as you enter text into the text entry control the static text label below will be updated.



Next we'll see how we can use this frame in Excel.

Creating a Custom Task Pane from a tkinter.Frame

To show a `tkinter.Frame` in Excel using PyXLL we use the *`create_ctp`* function.

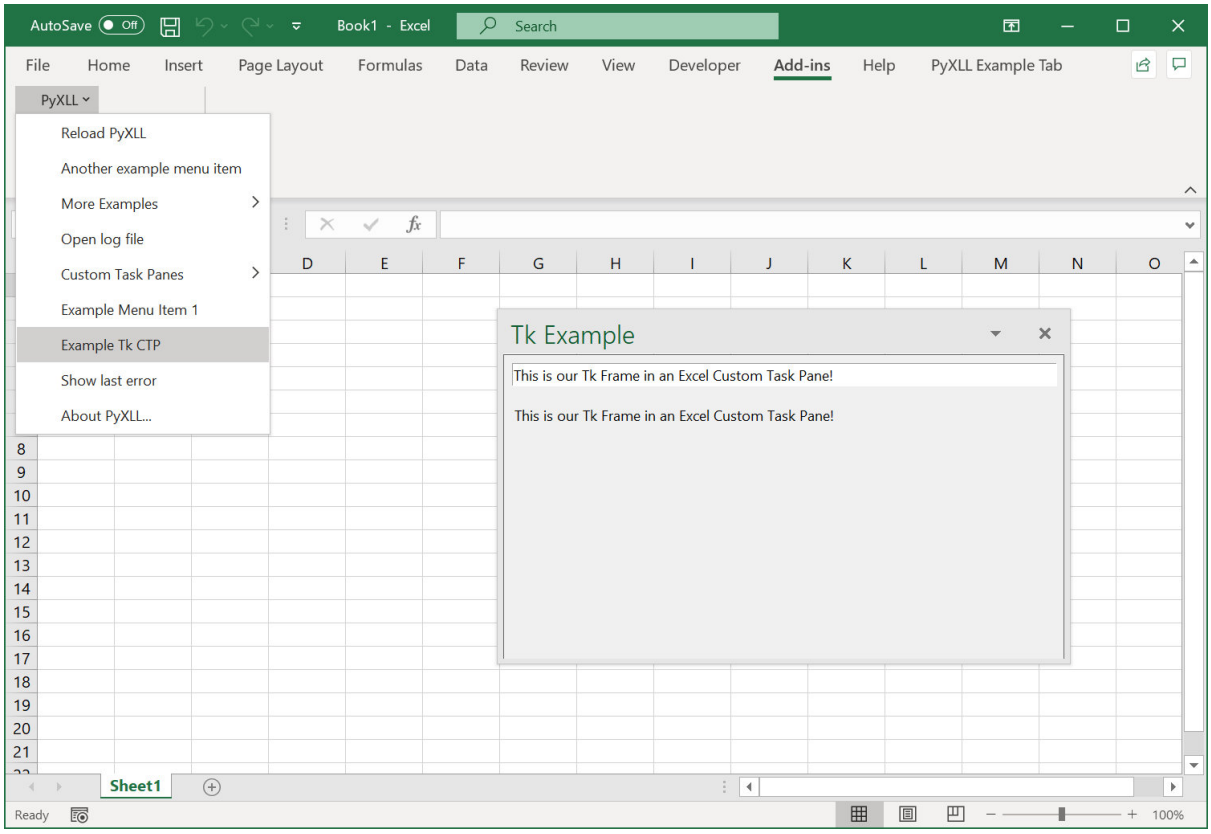
As above, before we can create the frame we have to create a root object to add it to. Unlike the above script, our function may be called many times and so we don't want to use the `tk.Tk` root object. Instead we use a `tk.Toplevel` object.

We can create the Custom Task Pane from many different places, but usually it will be from a *ribbon function* or a *menu function*.

The following code shows how we would create a custom task pane from an Excel menu function, using the `ExampleFrame` control from the example above.

```
from pyxll import xl_menu, create_ctp, CTPDockPositionFloating
import tkinter as tk
```

(continues on next page)



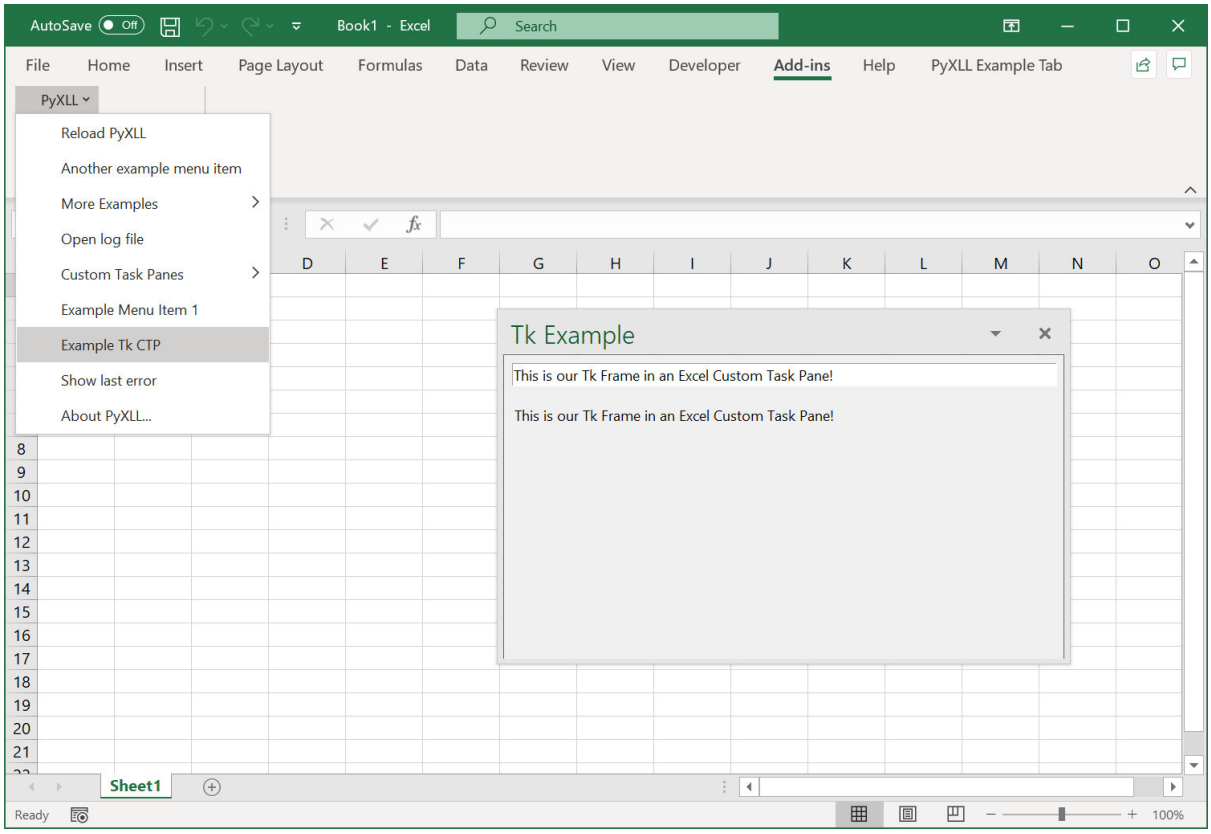
(continued from previous page)

```
@xl_menu("Example Tk CTP")
def example_tk_ctp():
    # Create the top level Tk window and give it a title
    window = tk.Toplevel()
    window.title("Tk Example")

    # Create our example frame from the code above and add
    # it to the top level window.
    frame = ExampleFrame(master=window)

    # Use PyXLL's 'create_ctp' function to create the custom task pane.
    # The width, height and position arguments are optional, but for this
    # example we'll create the CTP as a floating window rather than the
    # default of having it docked to the right.
    create_ctp(window,
               width=400,
               height=400,
               position=CTPDockPositionFloating)
```

When we add this code to PyXLL and reload the new menu function "Example Tk CTP" will be available, and when that menu function is run the ExampleFrame is opened as a Custom Task Pane in Excel.

Unlike a modal dialog, a Custom Task Pane does not block Excel from functioning. It can be moved and resized, and even docked into the current Excel window in exactly the same way as the native Excel tools.

See the API reference for `create_ctp` for more details.



3.10.4 Other UI Toolkits

PyXLL provides support for *PySide and PyQt*, *wxPython*, and *Tkinter*.

If you want to use another Python UI toolkit that's not already supported then you still may be able to. To do so you need to provide you own implementation of PyXLL's `CTPBridgeBase` class.

The *CTP Bridge* is what PyXLL uses to manage getting certain properties of the Python UI toolkit's window or frame objects in a consistent way and passing events from Excel to Python.

See the API reference for `CTPBridgeBase` for details of the methods you need to implement.

Once you have implemented your CTP Bridge you pass it to `create_ctp` as the `bridge_cls` keyword argument. Whatever object you pass as the widget to `create_ctp` will be used to construct your CTP Bridge object. PyXLL will take care of the rest of embedding your widget into Excel.

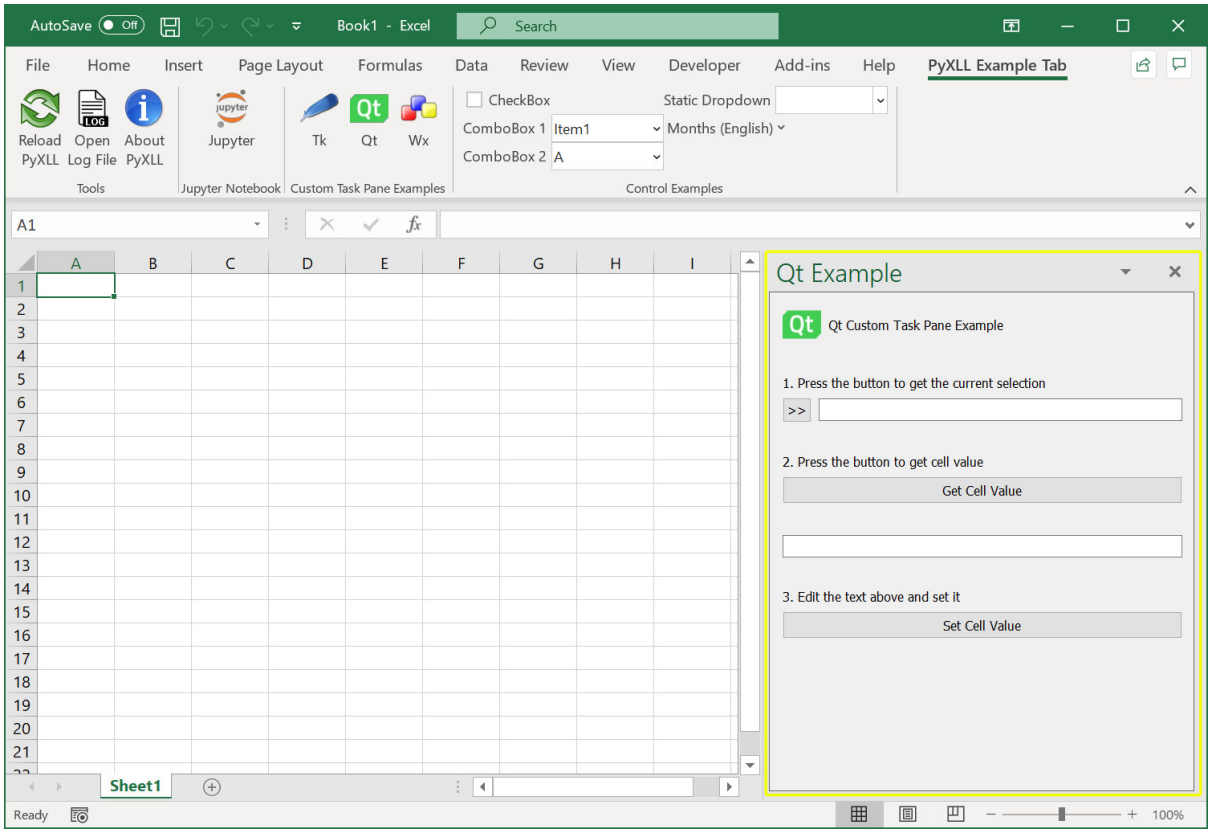


Fig. 1: A Python user interface in Excel

> **Warning**
>
> Writing a CTP Bridge requires detailed knowledge of the UI toolkit you are working with.
>
> This is an expert topic and PyXLL can only offer support limited to the functionality of PyXLL, and not third party packages.

Custom Task Panes (*CTPs*) are created using a control or widget from any of the supported Python UI toolkits by calling the PyXLL function `create_ctp`. All CTPs can be docked into the main Excel window and the initial position and size can be set when calling `create_ctp`.

For specific details of creating a custom task pane with any of the supported Python UI toolkits see the links above. Examples are provided in the *examples/custom_task_panes* folder in the PyXLL download.

3.11 ActiveX Controls

Similar to *Custom Task Panes*, Python UI controls can also be embedded as a control as part of the Excel worksheet.

These can be used for custom user interfaces that form part of the worksheet. For example, controls for a dashboard or for running more complex tasks that don't naturally fit into Excel's grid.

> **Note**
>
> For user interface controls that are hosted in a panel that can be floating or docked alongside the Excel worksheet see *Custom Task Panes*.

PyXLL has support for the following Python UI tookits.

3.11.1 PySide and PyQt

PySide and *PyQt* are both Python packages wrapping the popular *Qt* UI toolkit. They are quite similar but have different licenses and so which one you choose will be down to your own preference. Both work equally well with PyXLL.

User interfaces developed using Qt can be embedded as ActiveX controls directly on the Excel worksheet. If instead you want your controls to appear in a docked or floating window, see *Custom Task Panes*.

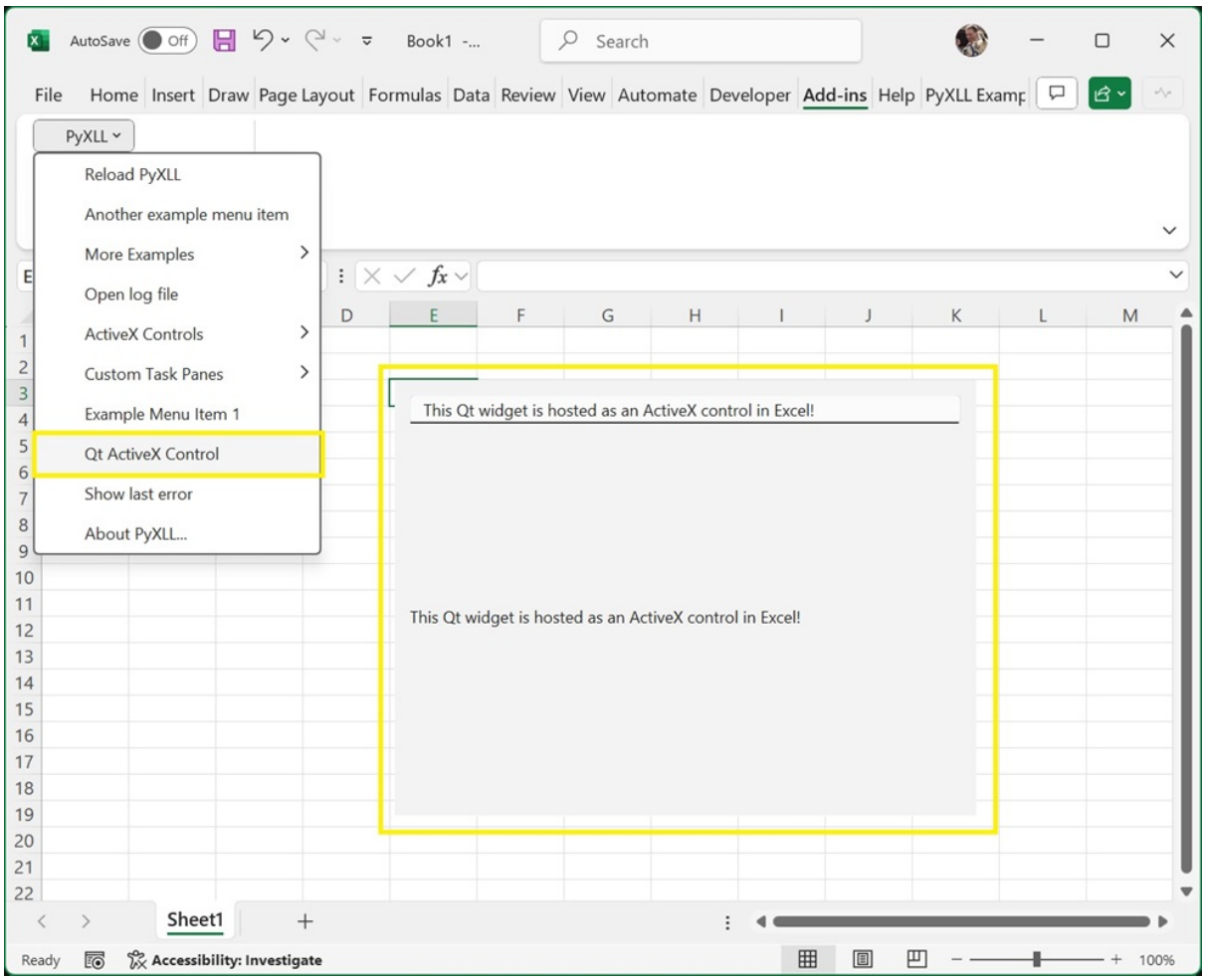
This document is not a guide to use PySide or PyQt. It is only intended to instruct you on how to use PySide and PyQt with the ActiveX feature of PyXLL. You should refer to the relevant package documentation for details of how to use each package.

Both PySide and PyQt can be installed using pip or conda, for example:

```
> pip install pyside6
# or
> pip install pyqt6
# or
> conda install pyside6
# or
> conda install "pyqt>=6"
```

Typically you will only want to install one or the other, and you should install it using pip *or* conda and not both.

You can find more information about PySide and PyQt on the websites, https://wiki.qt.io/Qt_for_Python and https://www.riverbankcomputing.com/software/pyqt/ respectively.



> **Tip**
>
> Any of PySide2, PySide6, PyQt5 and PyQt6 can be used with PyXLL.

Creating a Qt Widget

One of the main classes in Qt is the `QWidget` class. To create your own user interface it is this `QWidget` class that you will use, and it's what PyXLL will embed into Excel as an ActiveX control.

The following code demonstrates how to create simple Qt widget. If you run this code as a Python script then you will see the widget being shown.

```
from PySide6 import QtWidgets
# or from PyQt6 import QtWidgets
import sys

class ExampleWidget(QtWidgets.QWidget):

    def __init__(self):
        super().__init__()
        self.initUI()
```

(continues on next page)

(continued from previous page)

```
    def initUI(self):
        """Initialize the layout and child controls for this widget."""
        # Create a "Layout" object to help layout the child controls.
        # A QVBoxLayout lays out controls vertically.
        vbox = QtWidgets.QVBoxLayout(self)

        # Create a QLineEdit control and add it to the layout
        self.line_edit = QtWidgets.QLineEdit(self)
        vbox.addWidget(self.line_edit)

        # Create a QLabel control and add it to the layout
        self.label = QtWidgets.QLabel(self)
        vbox.addWidget(self.label)

        # Connect the 'textChanged' event to our 'onChanged' method
        self.line_edit.textChanged.connect(self.onChanged)

        # Set the layout for this widget
        self.setLayout(vbox)

    def onChanged(self, text):
        """Called when the QLineEdit's text is changed"""
        # Set the text from the QLineEdit control onto the label control
        self.label.setText(text)
        self.label.adjustSize()

if __name__ == "__main__":
    # Create the Qt Application
    app = QtWidgets.QApplication(sys.argv)

    # Create our example widget and show it
    widget = ExampleWidget()
    widget.show()

    # Run the Qt app
    sys.exit(app.exec_())
```

When you run this code you will see our example widget being display, and as you enter text into the line edit control the label below will be updated.

Next we'll see how we can use this widget in Excel.

Creating an ActiveX control from a Qt Widget

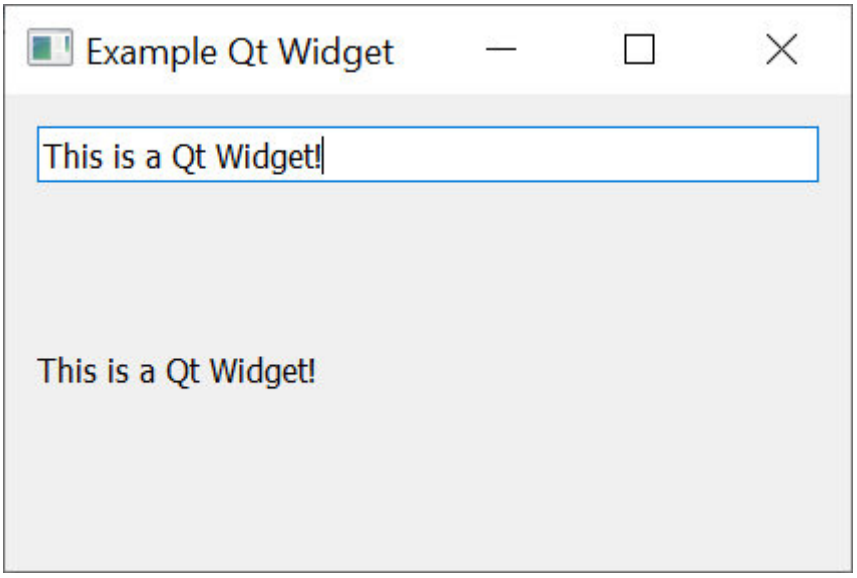
To show a QWidget in Excel as an ActiveX control we use the `create_activex_control` function.

As above, before we can create the widget we have to make sure the QApplication has been initialized. Unlike the above script, our function may be called many times and so we don't want to create a new application each time and so we check to see if one already exists.

The QApplication object must still exist when we call `create_activex_control`. If it has gone out of scope and been released then it will cause problems later so always make sure to keep a reference to it.

We can create the ActiveX control from many different places, but usually it will be from a *ribbon function*, or a *menu function*, or a *worksheet function*

The following code shows how we would create an ActiveX control from an Excel menu function, using the `ExampleWidget` control from the example above.




```
from pyxll import xl_menu, create_activex_control
from PySide6 import QtWidgets
# or from PyQt6 import QtWidgets

@xl_menu("Qt ActiveX Control")
def qt_activex_control():
    # Before we can create a Qt widget the Qt App must have been initialized.
    # Make sure we keep a reference to this until create_activex_control is called.
    app = QtWidgets.QApplication.instance()
    if app is None:
        app = QtWidgets.QApplication([])

    # Create our example Qt widget from the code above
    widget = ExampleWidget()

    # Use PyXLL's 'create_activex_control' function to create the ActiveX control.
    create_activex_control(widget)
```

When we add this code to PyXLL and reload the new menu function "Qt ActiveX Control" will be available, and when that menu function is run the ExampleWidget is created as an ActiveX control directly in the current Excel worksheet.

The optional parameters to `create_activex_control` can be used to set the size, position, and what sheet the control is created on.

To move or resize the ActiveX control you first need to enable `Design Mode`.

To enable Design Mode, go to the `Developer` tab in the Excel ribbon and select `Design Mode`. Whilst in Design Mode a bitmap preview will be displayed instead of the web control. You can now move and resize this shape. There may be some lag between resizing the preview image and the preview image updating.

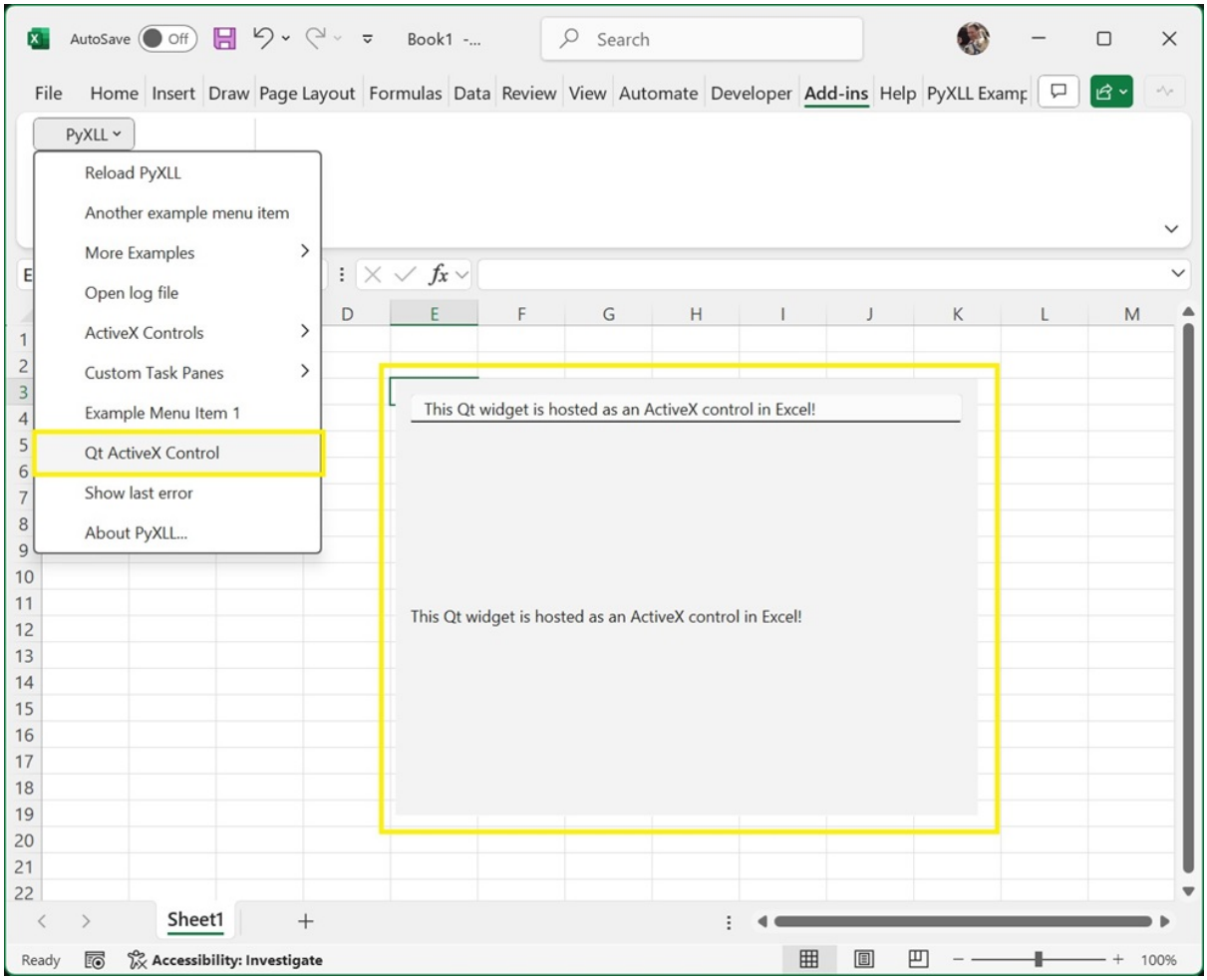
To return to the interactive web widget, unselect Design Mode.

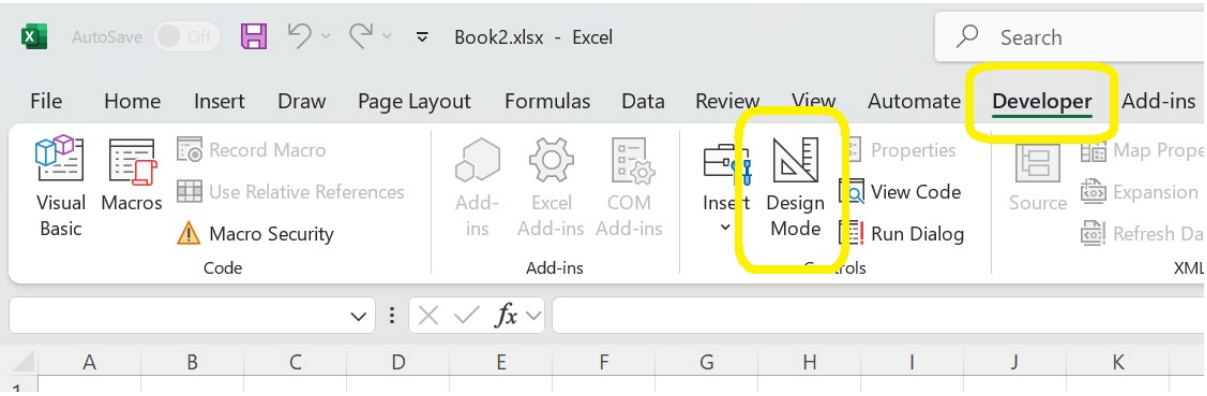
See the API reference for `create_activex_control` for more details.

3.11.2 wxPython

wxPython is a Python packages that wraps the UI toolkit *wxWindows*.

This document is not a guide to use wxPython or wxWindows. It is only intended to instruct you on how to use wxPython with the Custom Task Pane feature of PyXLL. You should refer to the relevant package documentation


AutoSave
Off
Book1 -...
Search
File
Home
Insert
Draw
Page Layout
Formulas
Data
Review
View
Automate
Developer
Add-ins
Help
PyXLL Examp
PyXLL
Reload PyXLL
Another example menu item
More Examples
Open log file
ActiveX Controls
Custom Task Panes
Example Menu Item 1
Qt ActiveX Control
Show last error
About PyXLL...
This Qt widget is hosted as an ActiveX control in Excel!
This Qt widget is hosted as an ActiveX control in Excel!
Sheet1
Ready
Accessibility: Investigate
100%


AutoSave
Off
Book2.xlsx - Excel
Search
File
Home
Insert
Draw
Page Layout
Formulas
Data
Review
View
Automate
Developer
Add-ins
Visual Basic
Macros
Record Macro
Use Relative References
Macro Security
Code
Add-ins
Excel Add-ins
COM Add-ins
Add-ins
Insert
Design Mode
Properties
View Code
Run Dialog
Source
Map Prope
Expansion
Refresh Da
XML

for details of how to use wxPython and wxWindows.

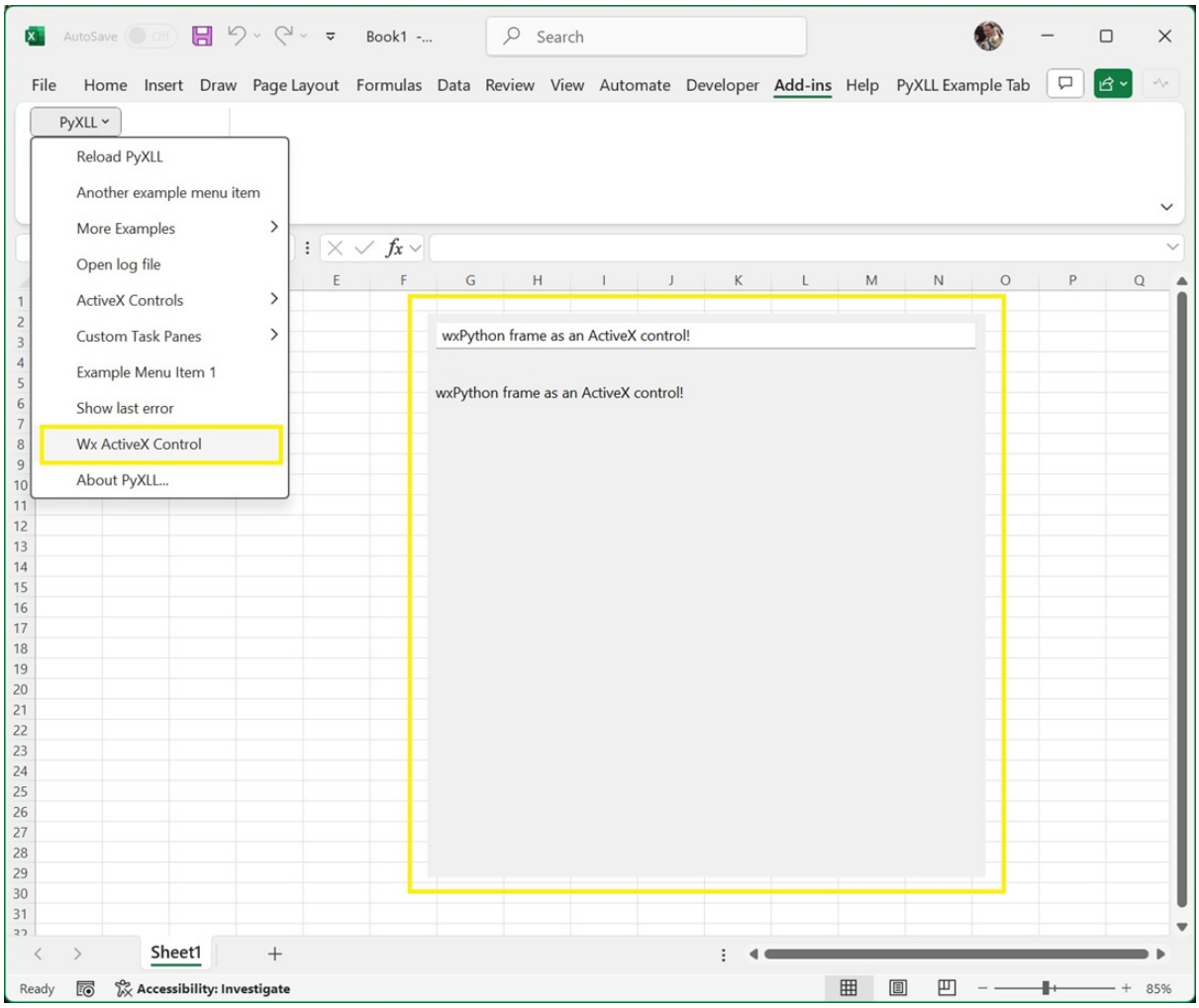
User interfaces developed using wxWindows can be embedded as ActiveX controls directly on the Excel worksheet. If instead you want your controls to appear in a docked or floating window, see *Custom Task Panes*.

Both wxWindows can be installed using pip or conda, for example:

```
> pip install wxpython
# or
> conda install wxpython
```

You should install it using pip *or* conda and not both.

You can find more information about wxPython on the website https://www.wxpython.org/.



Creating a wx Frame

Two of the main classes we'll use in wxPython are the `wx.Frame` and `wx.Panel` classes.

A `wx.Frame` is the main window type, and it's this that you'll create to contain your user interface that will be embedded into Excel as an ActiveX control. Frames typically host a single `wx.Panel` which is where all the controls that make up your user interface will be placed.

The following code demonstrates how to create simple wx.Frame and corresponding wx.Panel. If you run this code as a Python script then you will see the frame being shown.

```
import wx
```

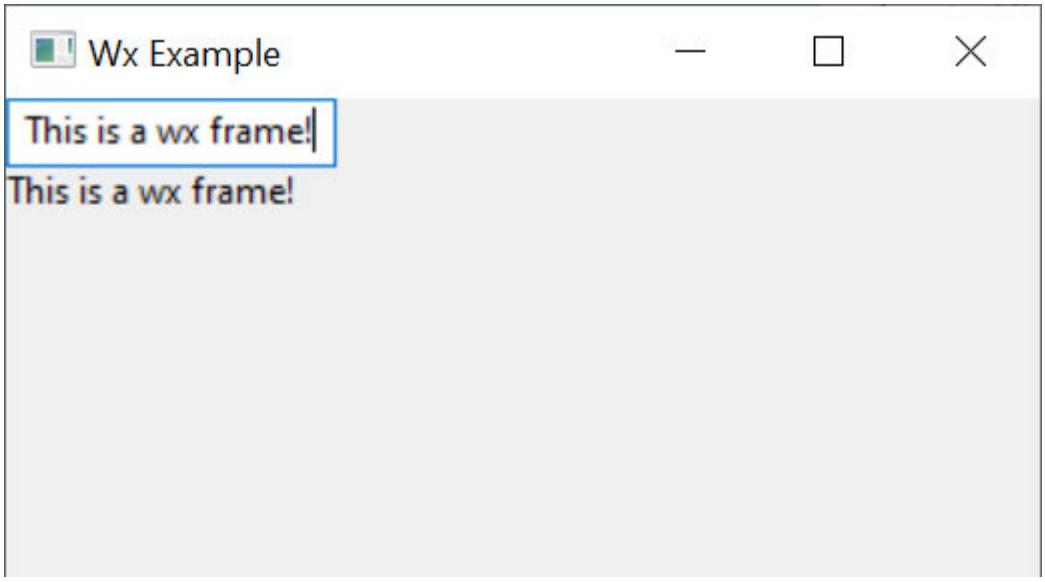
(continues on next page)

(continued from previous page)

```
class ExamplePanel(wx.Panel):

    def __init__(self, parent):
        super().__init__(parent=parent)

        # Create a sizer that will lay everything out in the panel.
        # A BoxSizer can arrange controls horizontally or vertically.
        sizer = wx.BoxSizer(orient=wx.VERTICAL)

        # Create a TextCtrl control and add it to the layout
        self.text_ctrl = wx.TextCtrl(self)
        sizer.Add(self.text_ctrl, flag=wx.ALL | wx.EXPAND, border=10)
        sizer.AddSpacer(20)

        # Create a StaticText control and add it to the layout
        self.static_text = wx.StaticText(self)
        sizer.Add(self.static_text, flag=wx.ALL | wx.EXPAND, border=10)

        # Connect the 'EVT_TEXT' event to our 'onText' method
        self.text_ctrl.Bind(wx.EVT_TEXT, self.onText)

        # Set the sizer for this panel and layout the controls
        self.SetSizer(sizer)
        self.Layout()

    def onText(self, event):
        """Called when the TextCtrl's text is changed"""
        # Set the text from the event onto the static_text control
        text = event.GetString()
        self.static_text.SetLabel(text)

class ExampleFrame(wx.Frame):

    def __init__(self):
        super().__init__(parent=None)

        # Create the panel that contains the controls for this frame
        self.panel = ExamplePanel(parent=self)


if __name__ == "__main__":
    # Create the wx Application object
    app = wx.App()

    # Construct our example Frame and show it
    frame = ExampleFrame()
    frame.Show()

    # Run the application's main event loop
    app.MainLoop()
```

When you run this code you will see our example frame being display, and as you enter text into the text control the static text below will be updated.

Next we'll see how we can use this frame in Excel.



Creating an ActiveX control from a wx.Frame

To show a wx.Frame in Excel using PyXLL as an ActiveX control we use the `create_activex_control` function.

As above, before we can create the frame we have to make sure the wx.App application object has been initialized. Unlike the above script, our function may be called many times and so we don't want to create a new application each time and so we check to see if one already exists.

The wx.App object must still exist when we call `create_activex_control`. If it has gone out of scope and been released then it will cause problems later so always make sure to keep a reference to it.

We can create the ActiveX control from many different places, but usually it will be from a *ribbon function*, or a *menu function*, or a *worksheet function*

The following code shows how we would create a ActiveX control from an Excel menu function, using the `ExampleFrame` control from the example above.

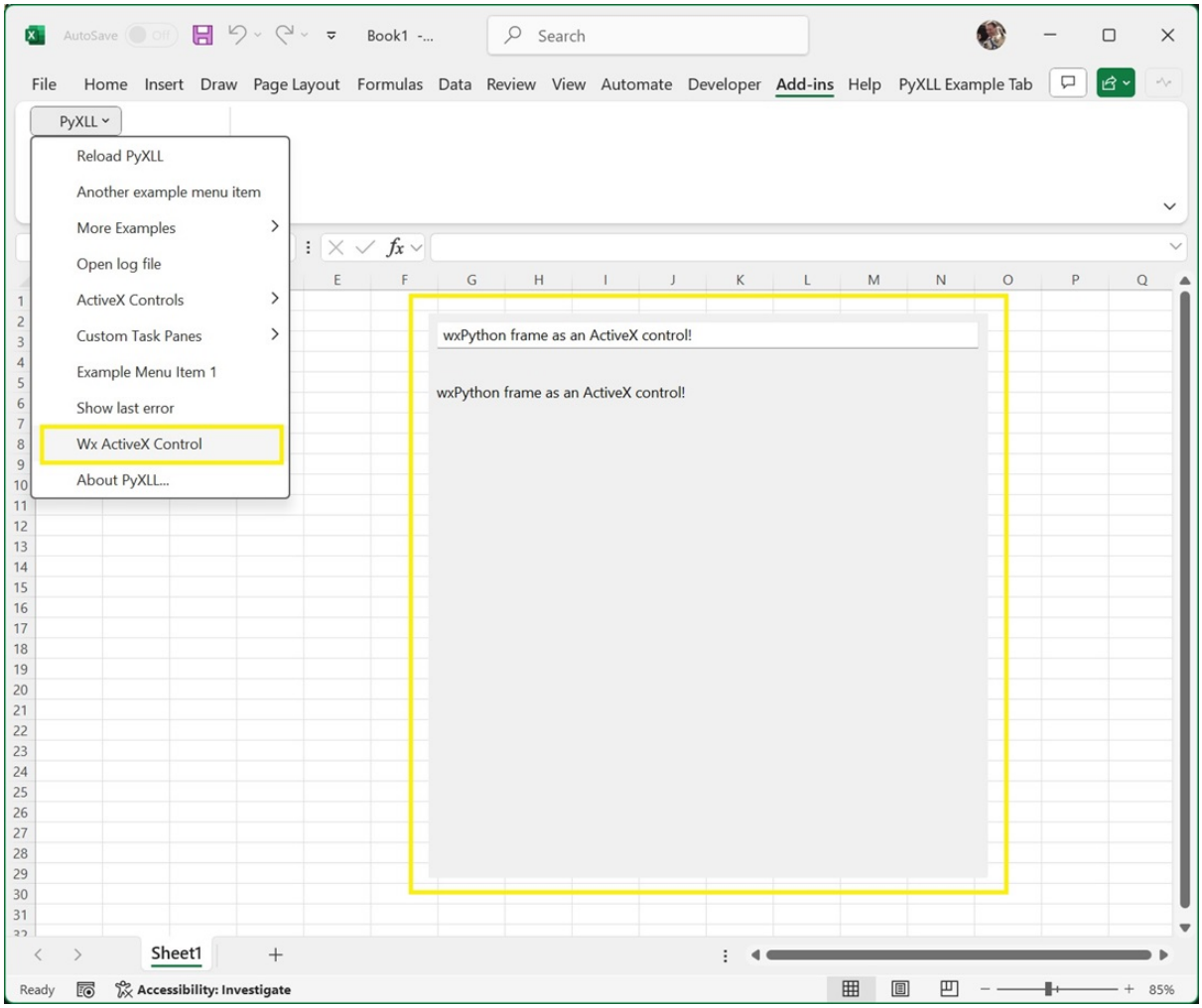
```
from pyxll import xl_menu, create_activex_control
import wx

@xl_menu("Wx ActiveX Control")
def wx_activex_control():
    # Before we can create a wx.Frame the wx.App must have been initialized.
    # Make sure we keep a reference to this until create_activex_control is called.
    app = wx.App.Get()
    if app is None:
        app = wx.App()

    # Create our example frame from the code above
    frame = ExampleFrame()

    # Use PyXLL's 'create_activex_control' function to create the ActiveX control.
    create_activex_control(frame, width=400, height=400)
```

When we add this code to PyXLL and reload the new menu function "Wx ActiveX Control" will be available, and when that menu function is run the ExampleFrame is opened as an ActiveX control in Excel.

The optional parameters to `create_activex_control` can be used to set the size, position, and what sheet the control is created on.

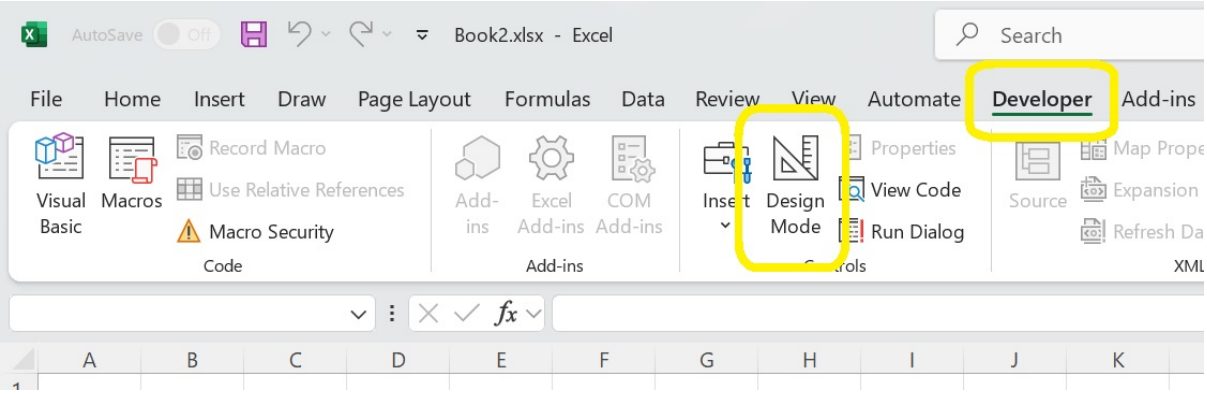
To move or resize the ActiveX control you first need to enable `Design Mode`.

To enable Design Mode, go to the `Developer` tab in the Excel ribbon and select `Design Mode`. Whilst in Design


AutoSave
Book1 -...
Search
File
Home
Insert
Draw
Page Layout
Formulas
Data
Review
View
Automate
Developer
Add-ins
Help
PyXLL Example Tab
PyXLL
Reload PyXLL
Another example menu item
More Examples
Open log file
ActiveX Controls
Custom Task Panes
Example Menu Item 1
Show last error
Wx ActiveX Control
About PyXLL...
wxPython frame as an ActiveX control!
wxPython frame as an ActiveX control!
Sheet1
Ready
Accessibility: Investigate
85%

Mode a bitmap preview will be displayed instead of the web control. You can now move and resize this shape. There may be some lag between resizing the preview image and the preview image updating.

To return to the interactive web widget, unselect Design Mode.



See the API reference for `create_activex_control` for more details.

3.11.3 Tkinter

tkinter is a Python packages that wraps the *Tk GUI toolkit*.

tkinter is included with Python and so is available to use without needing to install any additional packages.

User interfaces developed using Qt can be embedded as ActiveX controls directly on the Excel worksheet. If instead you want your controls to appear in a docked or floating window, see *Custom Task Panes*.

This document is not a guide to use tkinter. It is only intended to instruct you on how to use Tkinter with the Custom Task Pane feature of PyXLL. You should refer to the tkinter documentation for details of how to use tkinter.

You can find more information about tkinter in the Python docs website https://docs.python.org/3/library/tkinter.html.

Creating a Tk Frame

One of the main classes in tkinter is the `Frame` class. To create your own user interface it is this `Frame` class that you will use, and it's what PyXLL will embed into Excel as a Custom Task Pane.

The following code demonstrates how to create simple `tkinter.Frame`. If you run this code as a Python script then you will see the frame being shown.

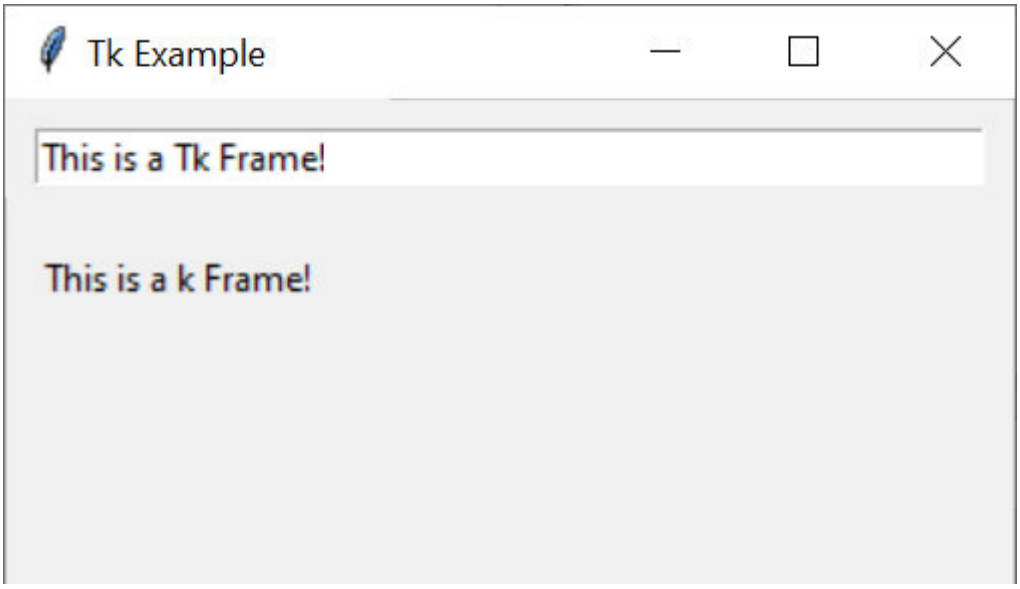
```
import tkinter as tk

class ExampleFrame(tk.Frame):

    def __init__(self, master):
        super().__init__(master)
        self.initUI()

    def initUI(self):
        # allow the widget to take the full space of the root window
        self.pack(fill=tk.BOTH, expand=True)

        # Create a tk.Entry control and place it using the 'grid' method
        self.entry_value = tk.StringVar()
        self.entry = tk.Entry(self, textvar=self.entry_value)
        self.entry.grid(column=0, row=0, padx=10, pady=10, sticky="ew")

        # Create a tk.Label control and place it using the 'grid' method
```

(continues on next page)

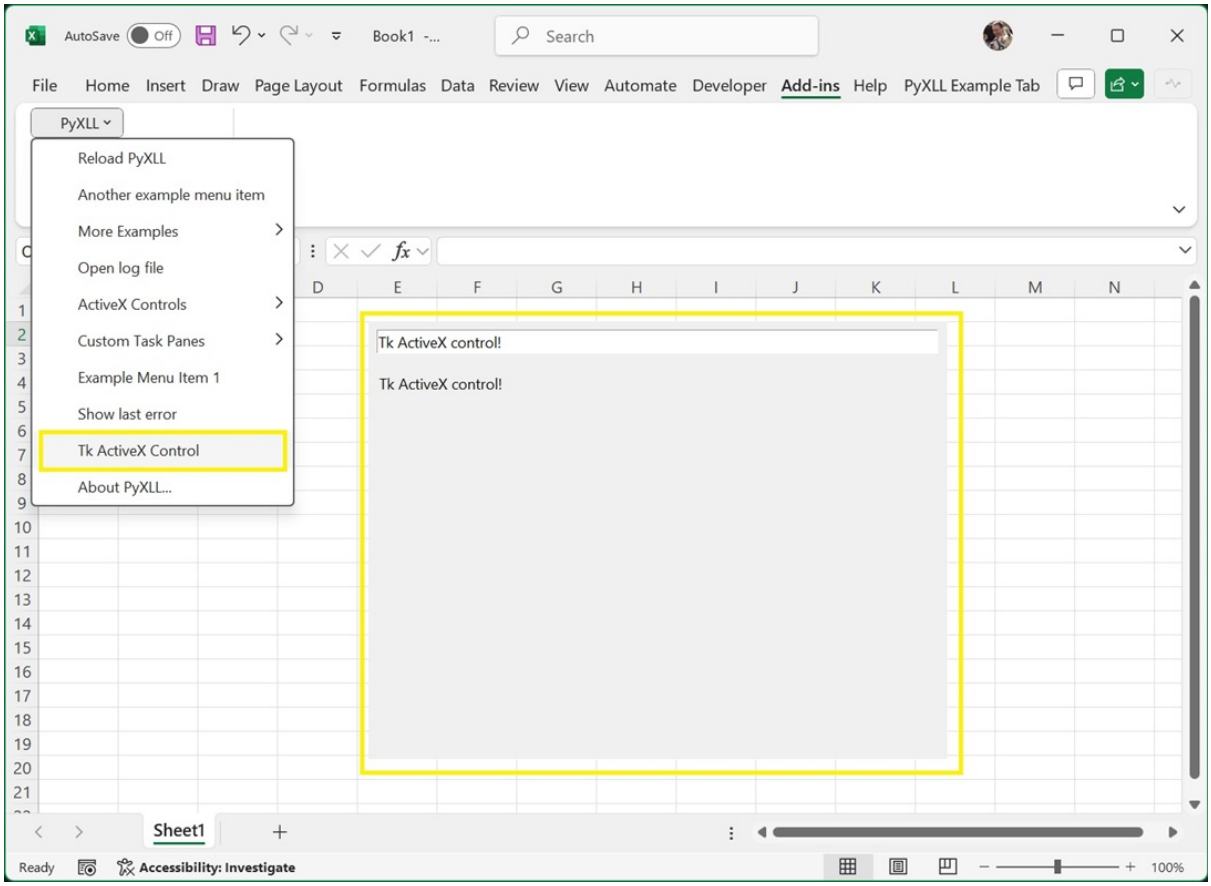
(continued from previous page)

```
        self.label_value = tk.StringVar()
        self.label = tk.Label(self, textvar=self.label_value)
        self.label.grid(column=0, row=1, padx=10, pady=10, sticky="w")

        # Bind write events on the 'entry_value' to our 'onWrite' method
        self.entry_value.trace("w", self.onWrite)

        # Allow the first column in the grid to stretch horizontally
        self.columnconfigure(0, weight=1)

    def onWrite(self, *args):
        """Called when the tk.Entry's text is changed"""
        # Update the label's value to be the same as the entry value
        self.label_value.set(self.entry_value.get())

if __name__ == "__main__":
    # Create the root Tk object
    root = tk.Tk()

    # Give the root window a title
    root.title("Tk Example")

    # Construct our frame object
    ExampleFrame(master=root)

    # Run the tk main loop
    root.mainloop()
```

When you run this code you will see our example frame being display, and as you enter text into the text entry control the static text label below will be updated.



Next we'll see how we can use this frame in Excel.



Creating an ActiveX control from a tkinter.Frame

To show a `tkinter.Frame` in Excel using PyXLL we use the *`create_activex_control`* function.

As above, before we can create the frame we have to create a root object to add it to. Unlike the above script, our function may be called many times and so we don't want to use the `tk.Tk` root object. Instead we use a `tk.Toplevel` object.

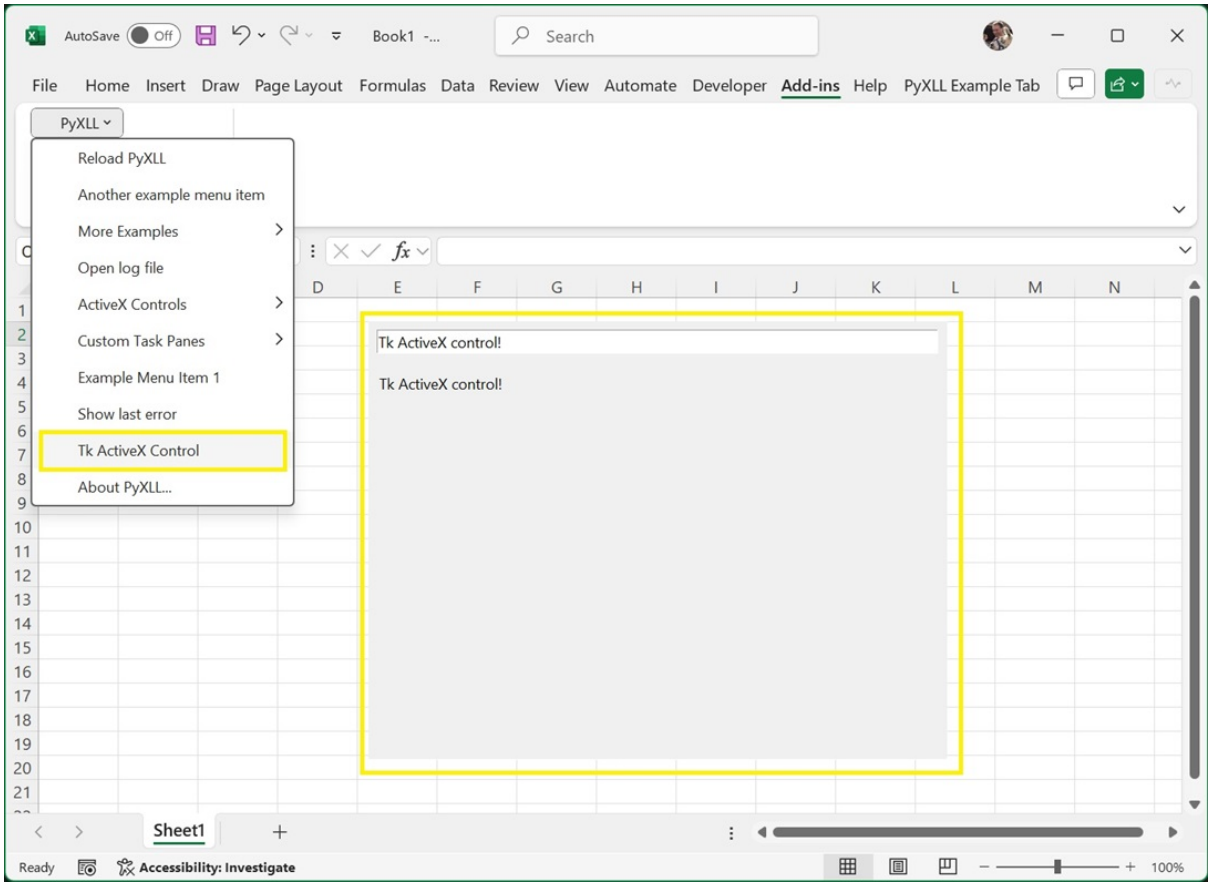
We can create the ActiveX control from many different places, but usually it will be from a *ribbon function*, or a *menu function*, or a *worksheet function*

The following code shows how we would create an ActiveX control from an Excel menu function, using the `ExampleFrame` control from the example above.

```
from pyxll import xl_menu, create_activex_control
import tkinter as tk

@xl_menu("Tk ActiveX Control")
def tk_activex_control():
    # Create the top level Tk window
    window = tk.Toplevel()

    # Create our example frame from the code above and add
    # it to the top level window.
    frame = ExampleFrame(master=window)

    # Use PyXLL's 'create_activex_control' function to create the ActiveX control.
    create_activex_control(window, width=400, height=400)
```

When we add this code to PyXLL and reload the new menu function "Tk ActiveX Control" will be available, and when that menu function is run the ExampleFrame is opened as an ActiveX control in Excel.

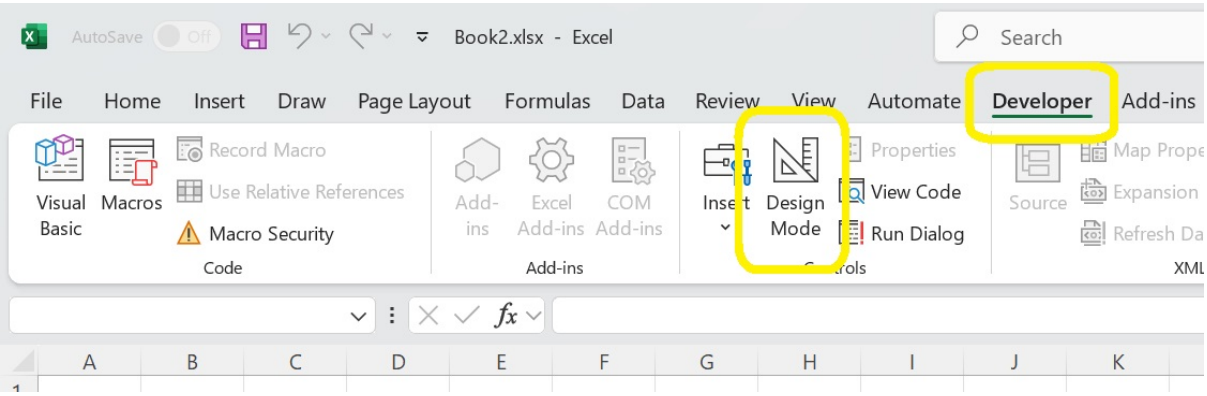
The optional parameters to `create_activex_control` can be used to set the size, position, and what sheet the control is created on.



To move or resize the ActiveX control you first need to enable `Design Mode`.

To enable Design Mode, go to the `Developer` tab in the Excel ribbon and select `Design Mode`. Whilst in Design Mode a bitmap preview will be displayed instead of the web control. You can now move and resize this shape. There may be some lag between resizing the preview image and the preview image updating.

To return to the interactive web widget, unselect Design Mode.



See the API reference for `create_activex_control` for more details.

3.11.4 Other UI Toolkits

PyXLL provides support for *PySide and PyQt*, *wxPython*, and *Tkinter*.

If you want to use another Python UI toolkit that's not already supported then you still may be able to. To do so you need to provide you own implementation of PyXLL's `AtxBridgeBase` class.

The *ActiveX Bridge* is what PyXLL uses to manage getting certain properties of the Python UI toolkit's window or frame objects in a consistent way and passing events from Excel to Python.

See the API reference for `AtxBridgeBase` for details of the methods you need to implement.

Once you have implemented your ActiveX Bridge you pass it to `create_activex_control` as the `bridge_cls` keyword argument. Whatever object you pass as the widget to `create_activex_control` will be used to construct your ActiveX Bridge object. PyXLL will take care of the rest of embedding your widget into Excel.

Warning

Writing an ActiveX Bridge requires detailed knowledge of the UI toolkit you are working with.

This is an expert topic and PyXLL can only offer support limited to the functionality of PyXLL, and not third party packages.

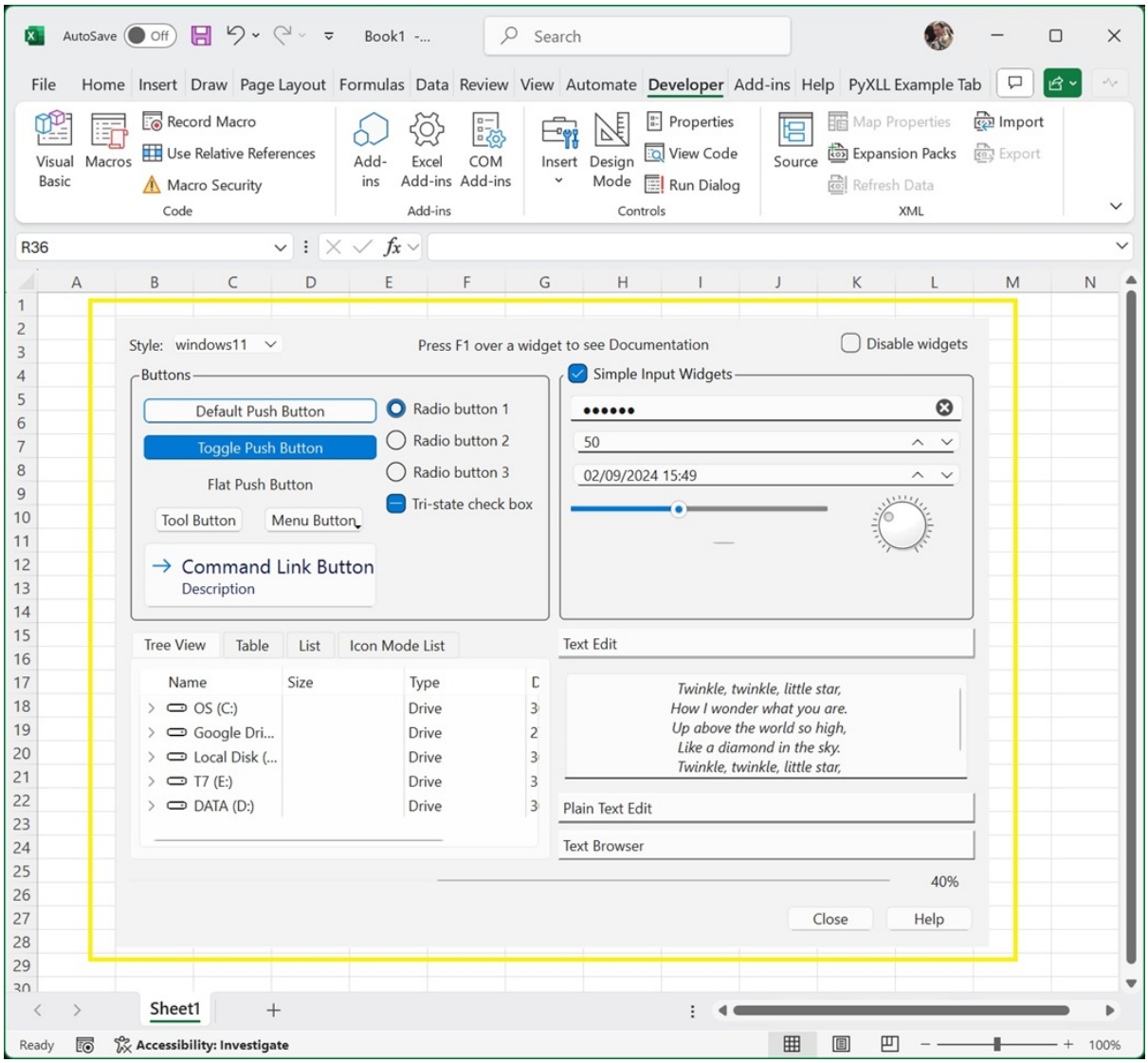


Fig. 2: A Python user interface in Excel

ActiveX controls can be created using a control or widget from any of the supported Python UI toolkits by calling the PyXLL function `create_activex_control`. ActiveX controls are objects embedded in the Excel worksheet. The initial position and size can be set when calling `create_activex_control`.

For specific details of creating an ActiveX control with any of the supported Python UI toolkits see the links above.

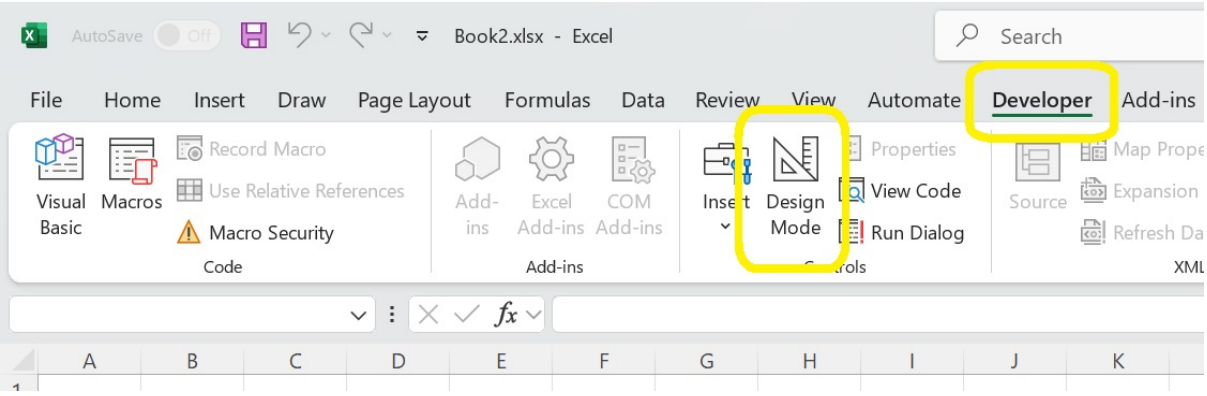
3.11.5 Resizing ActiveX Controls

To move or resize the ActiveX control you first need to enable `Design Mode`.

To enable Design Mode, go to the `Developer` tab in the Excel ribbon and select `Design Mode`.

Whilst in Design Mode a bitmap preview will be displayed instead of the web control. You can now move and resize this shape. There may be some lag between resizing the preview image and the preview image updating.

To return to the interactive web widget, unselect Design Mode.



See the API reference for `create_activex_control` for more details.

3.11.6 Other Considerations

1. The Python control hosted in an ActiveX control is not saved with the workbook.

   When opening a workbook that was saved with ActiveX controls added, by default you will see a placeholder instead of Python control.

   To recreate the Python control when the workbook is opened, one solution is to use a worksheet function that recalculates when the workbook is opened. By passing `name` to `create_activex_control` you can update an existing control with the newly created Python control.

   See *Creating an ActiveX Control from a Worksheet Function* for more details about how to do this.

2. ActiveX controls don't zoom in or out when zooming in or out of the Excel worksheet.

   The size of the control will change as you zoom, but the size of the contents will not automatically be scaled.

   You will need to bear this in mind when designing your user interface to ensure that it handles being resized correctly, if you want the Excel user to be able to change the zoom in Excel and still be able to use the control.

3.12 Using Pandas in Excel

Pandas DataFrames are Excel are a great match as both deal with 2d tables of data. With PyXLL, you can pass data as pandas DataFrames between Python and Excel easily.

rows	10
columns	5

=random_dataframe(C22,C23)		C	D	E
0	0.641194	0.260211	0.805942	0.419147	0.199523
1	0.17142	0.620258	0.371597	0.11461	0.418762
2	0.316006	0.418837	0.938002	0.465505	0.854013
3	0.908746	0.04335	0.11045	0.808443	0.693135
4	0.377972	0.630045	0.668993	0.90478	0.165449
5	0.326053	0.780202	0.031809	0.542983	0.365033
6	0.705675	0.547457	0.632721	0.368196	0.756802
7	0.329699	0.766581	0.86182	0.571743	0.240837
8	0.601057	0.225453	0.170927	0.142215	0.222863
9	0.902797	0.162996	0.620218	0.97348	0.726023

- *Returning a DataFrame from a Function*
- *Taking a DataFrame as a Function Argument*
- *Trimming DataFrame Arguments*
- *Returning Multiple DataFrames*
- *Formatting DataFrames*
- *Plotting DataFrames*
- *Reading and Writing DataFrames in Macros*
- *Passing DataFrames as Objects Instead of Arrays*
- *Pandas Series*
- *Advanced Usage of the Pandas Type Converters*

Note

You can also use `polars` DataFrames as well as pandas.

To distinguish between polars and pandas, use the `polars.dataframe` and `pandas.dataframe` types instead of just `dataframe` in your function signatures.

If you use just `dataframe` PyXLL will default to using pandas.

See *Polars DataFrames* for more information about polars types in PyXLL.

3.12.1 Returning a DataFrame from a Function

By far one of the most common ways to work with pandas data in Excel is with a worksheet function that returns a DataFrame.

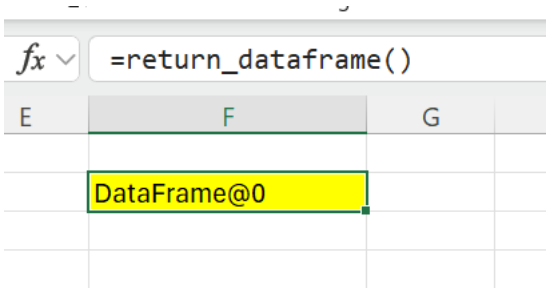
If you are not already familiar with writing Excel worksheet functions in Python using PyXLL, please see *Worksheet Functions* first.

This video demonstrates how to write worksheet functions in Python, using PyXLL. It also covers returning pandas DataFrames from functions towards the end of the video.

Consider the following function that returns a panda DataFrame:

```
import pandas as pd
from pyxll import xl_func

@xl_func
def return_dataframe():
     """A function that returns a DataFrame."""
    df = pd.DataFrame({
       "A": [1, 2, 3],
       "B": [4, 5, 6]
    })

    return df
```

The *@xl_func* decorator exposes the `return_dataframe` function to Excel as a worksheet function, but when we call this function from Excel the result might not be exactly what you expected:



The DataFrame object here is being returned as an *object handle*. This object handle can be passed to other PyXLL functions and the Python function will be called with the actual pandas DataFrame object. This is really useful (and very fast) as it allows us to pass large, complex objects between Python functions easily.

However, it is not what we want in this case!

We need to tell the PyXLL add-in to convert the returned DataFrame object into a range of values so that all of the DataFrames values get returned to the Excel grid.

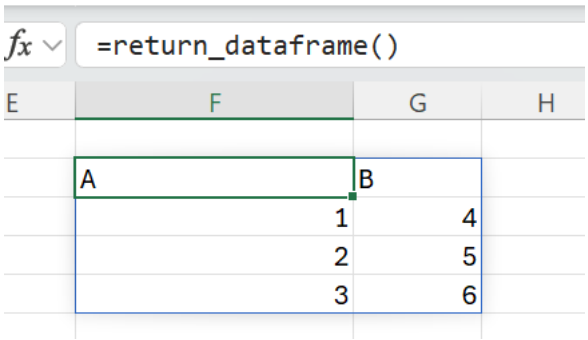
To do that, we need to specify the return type when declaring the function. That is done either by adding a *function signature* to the *@xl_func* decorator; using the *@xl_return* decorator; or by using a Python type annotation.

The simplest method is to add a *function signature* to our function as follows:

```
import pandas as pd
from pyxll import xl_func

@xl_func(": dataframe")
def return_dataframe():
     """A function that returns a DataFrame."""
    df = pd.DataFrame({
       "A": [1, 2, 3],
       "B": [4, 5, 6]
    })

    return df
```

Note that we don't have any arguments. The function signature is of the form `argument types: return type` and so in our signature above we don't have anything before the `:` as there are no arguments to the function.

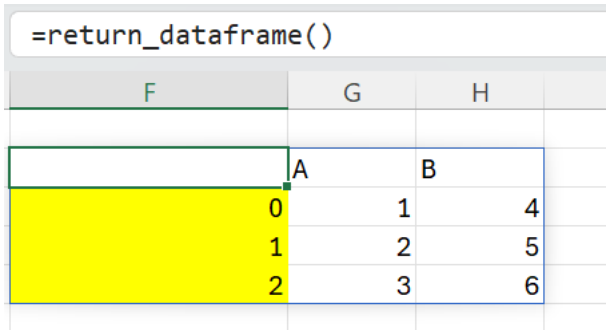
With the return type now specified, when we call the function in Excel the PyXLL add-in will convert the pandas DataFrame object to an array of values:



The **dataframe** return type can be parameterized using various options. These options control how the pandas DataFrame object is converted to the array that you see in Excel.

For example, if you want the index to be included in the array shown in Excel, you can set the `index` option to `True` (by default, the index is only included if it is a named index):

```
import pandas as pd
from pyxll import xl_func

@xl_func(": dataframe<index=True>")
def return_dataframe():
     """A function that returns a DataFrame."""
    df = pd.DataFrame({
       "A": [1, 2, 3],
       "B": [4, 5, 6]
    })

    return df
```



For more details of how to specify return types and their type parameters, please see *Argument and Return Types*.

You can find the documentation for all of the available parameters for the `dataframe` return type on the *Pandas Types* page.

3.12.2 Taking a DataFrame as a Function Argument

In the above section we saw how by specifying the function return type, PyXLL can convert the returned pandas DataFrame to an array of values on the Excel grid.

Similarly, PyXLL can also convert a range of values from the Excel grid passed to a function into a pandas DataFrame.

Let's look at an example function that takes a DataFrame argument, as well as a string and returns a number:

```
import pandas as pd
from pyxll import xl_func

@xl_func("dataframe df, str col: float")
```

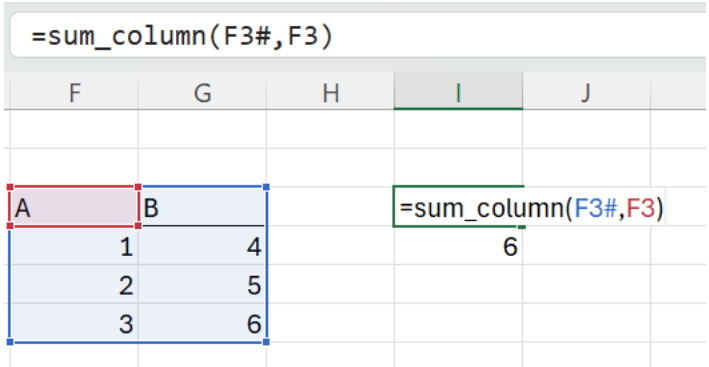
(continues on next page)

(continued from previous page)

```
def sum_column(df, col):
    """Sum the values in a column."""
    column_sum = df[col].sum()
    return column_sum
```

The *function signature* passed to `@xl_func` specifies that the argument `df` should be converted from a range of Excel values to a pandas DataFrame.

To call this from Excel we enter it as a formula, passing in our data range including the column headers, and the column we want to sum.



PyXLL converts the input range argument to a pandas DataFrame for us and calls our *sum_column* function with that converted DataFrame.

We can also specify various type parameters to the `dataframe` argument type, as with the return type. For example, the `index` option can be used to specify the number of columns to use as the index for the DataFrame (if supplying an index).

Instead of using the *function signature* as shown above, another option is to use the `@xl_arg` decorator. This works in the same way, but lets us specify the argument type parameters as keyword arguments to the `@xl_arg` decorator.

For example, the function above could be re-written as follows:

```
import pandas as pd
from pyxll import xl_func, xl_arg, xl_return

@xl_func
@xl_arg("df", "dataframe")
@xl_arg("col", "str")
@xl_return("float")
def sum_column(df, col):
    """Sum the values in a column."""
    column_sum = df[col].sum()
    return column_sum
```

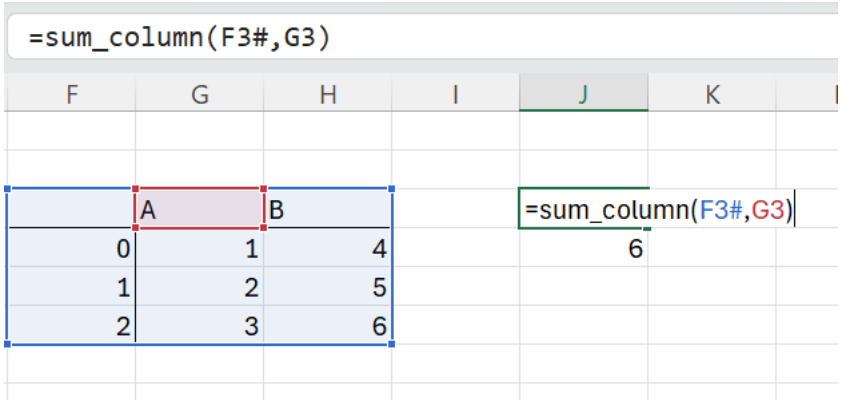
And to specify that the first column is to be used as the DataFrame index, we would set the `index` type parameter like so:

```
import pandas as pd
from pyxll import xl_func, xl_arg, xl_return

@xl_func
@xl_arg("df", "dataframe", index=1)  # equivalent to "dataframe<index=1>"
@xl_arg("col", "str")
@xl_return("float")
def sum_column(df, col):
    """Sum the values in a column."""
```

(continues on next page)

(continued from previous page)

```
    column_sum = df[col].sum()
    return column_sum
```



This time, the DataFrame passed to our `sum_column` function has its index constructed from the first column in the Excel range. Setting `index` to greater than one results in a DataFrame with a MultiIndex.

For more details of how to specify arguments types and their type parameters, please see *Argument and Return Types*.

You can find the documentation for all of the available parameters for the `dataframe` argument type on the *Pandas Types* page.

3.12.3 Trimming DataFrame Arguments

New in PyXLL 5.11

When passing a DataFrame as an argument to an Excel function sometimes you might want to pass a larger range than the actual data.

For example, when creating a sheet you might allow some extra rows in case the data grows, or to allow space for the user of the sheet to add more rows. You might even select a range of entire columns.

Converting a larger range than necessary takes additional time when constructing the DataFrame. Instead of constructing the entire DataFrame and then removing empty rows in your Python code, it is much more efficient (and easier!) for PyXLL to trim any trailing empty rows or columns from the Excel data before constructing the DataFrame.

To tell PyXLL to trim empty space use the `trim` type parameter to the `dataframe` type.

For example:

```
import pandas as pd
from pyxll import xl_func

@xl_func("dataframe<trim=True> df, str col: float")
def sum_column(df, col):
    """Sum the values in a column."""
    column_sum = df[col].sum()
    return column_sum
```

The function above can now accept a large range including empty rows and columns around the actual data. Before the DataFrame is constructed the data is trimmed to remove empty rows and columns from the top, bottom, left and right of the data.

When specifying `trim=True` any empty rows from the top or bottom of the range, or empty columns from the left or right of the range, will be removed.

If you only want to trim rows or columns from certain directions instead of using `trim=True` you can use a string to specify top, left, bottom and/or right by using the characters `t`, `l`, `b`, `r`, respectively.

For example, to only trim rows from the bottom and columns from the right you would use `trim='br'`.

3.12.4 Returning Multiple DataFrames

Sometimes you might have a function that returns more than one DataFrames.

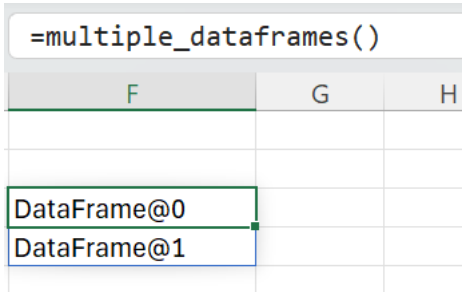
When returning values to Excel, the result has to fit into a grid and so returning multiple DataFrames doesn't fit as well as a single DataFrame.

If it makes sense to break your function into multiple functions, with each one returning a different DataFrame, that is a good solution. But, if that's not feasible or results in unwanted recomputation and you need to keep it all in one function, you can return multiple DataFrames from a single function.

Rather than returning multiple DataFrames as multiple grids of data from a single function, instead we have to return them as a list of objects. By returning them as a list of objects this then fits into Excel's grid.

```
import pandas as pd
from pyxll import xl_func

@xl_func(": object[]")
def multiple_dataframes():
    """A function that returns multiple DataFrames."""
    df1 = pd.DataFrame({
       "A": [1, 2, 3],
       "B": [4, 5, 6]
    })

    df2 = pd.DataFrame({
       "X": [-1, -2, -3],
       "Y": [-4, -5, -6]
    })

    return [
        df1,
        df2
    ]
```

Above we're using the `object[]` return type. The list of DataFrames returned by the function are returned to Excel as an array of object handles:



Our function returns multiple DataFrames, but as an array of object handles.

Next we need to convert those object handles into Excel arrays, which we can do with another function. This next function doesn't do anything, but by specifying `dataframe` as the return type PyXLL will do the conversion for us:

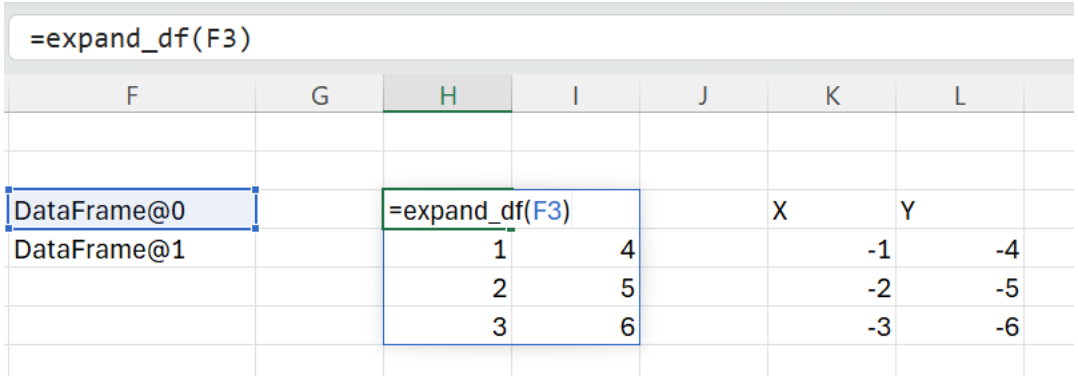
```
import pandas as pd
from pyxll import xl_func

@xl_func("object df: dataframe")
def expand_df(df):
```

(continues on next page)

(continued from previous page)

```
    if not isinstance(df, pd.DataFrame):
        raise TypeError("Expected a DataFrame object")

    # Will be converted to an Excel array because of the return type
    return df
```

We can now call this new function on the objects returned previously to get the contents of the DataFrames:



3.12.5 Formatting DataFrames

PyXLL has a feature for applying cell formatting to the results of a worksheet function. This can be very useful for giving your spreadsheets a consistent look and feel, as well as saving time applying formatting manually.

For complete details about cell formatting, please see the *Cell Formatting* and *Pandas DataFrame Formatting* sections of the user guide.

The following video also explains how the cell formatting functionality can be used:

3.12.6 Plotting DataFrames

Pandas DataFrames can be plotted using the `DataFrame.plot` method. This is a convenient way to quickly produce plots from DataFrames.

The `DataFrame.plot` method uses the plotting package `matplotlib`. The PyXLL add-in can display matplotlib figures directly in Excel, and so it is also possible to display plots of pandas DataFrames in Excel.

The function *`plot`* takes a matplotlib figure and displays it in Excel. To display a pandas plot we first create a matplotlib `Figure` and `Axes`. Then we call `DataFrame.plot`, passing in the `Axex` object as we want it to plot to the matplotlib figure we created. Finally, we call *`plot`* to display the figure in Excel.

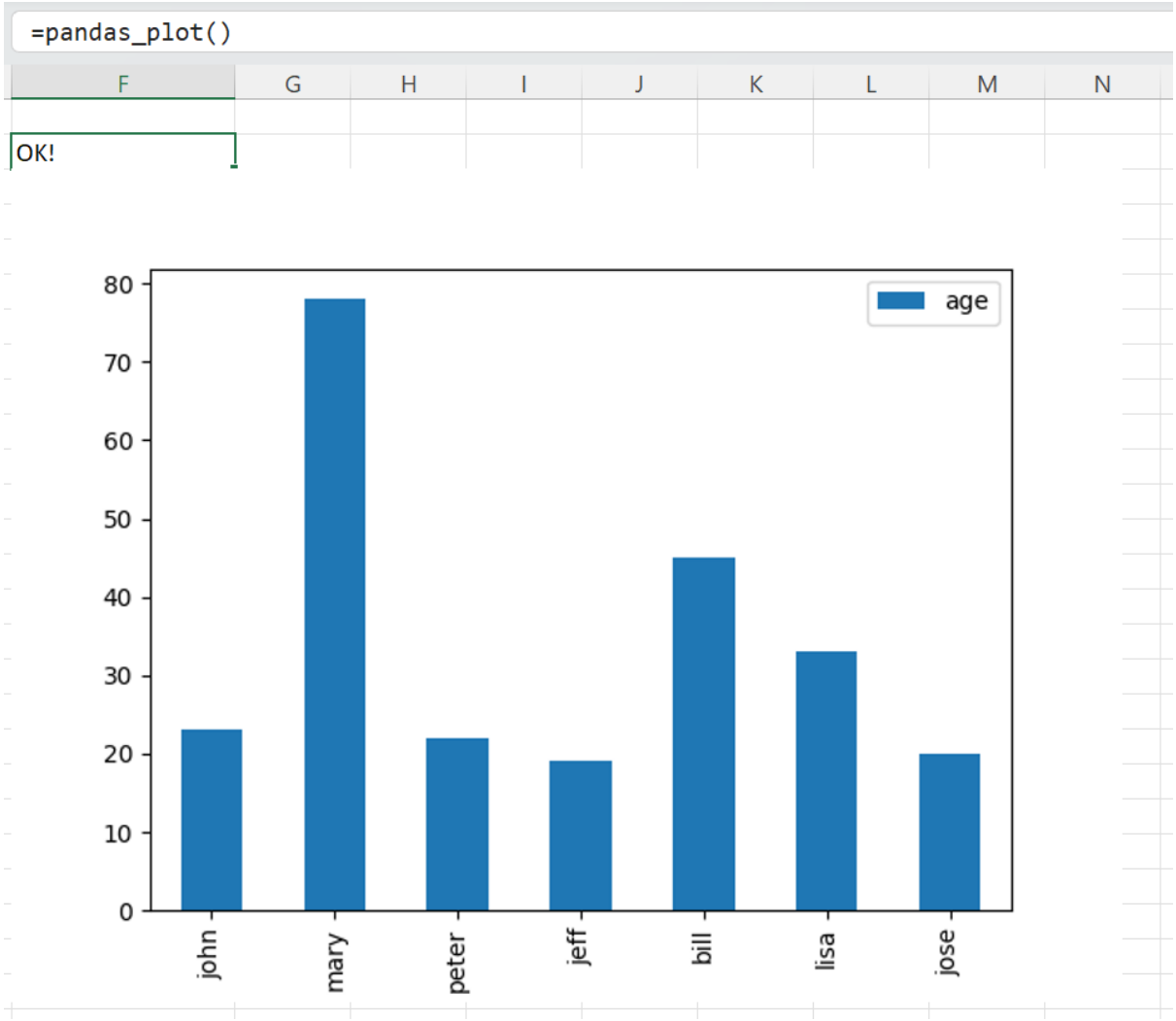
```
from pyxll import xl_func, plot
import matplotlib.pyplot as plt
import pandas as pd

    @xl_func
    def pandas_plot():
        # Create a DataFrame to plot
        df = pd.DataFrame({
            'name':['john','mary','peter','jeff','bill','lisa','jose'],
            'age':[23,78,22,19,45,33,20],
            'gender':['M','F','M','M','M','F','M'],
            'state':['california','dc','california','dc','california','texas','texas
↪'],
            'num_children':[2,0,0,3,2,1,4],
            'num_pets':[5,1,0,5,2,2,3]
        })
```

(continues on next page)

(continued from previous page)

```
    # Create the matplotlib Figure and Axes objects
    fig, ax = plt.subplots()

    # Plot a bar chart to the Axes we just created
    df.plot(kind='bar',x='name',y='age', ax=ax)

    # Show the matplotlib Figure created above
    plot(fig)

    return "OK!"
```

The above code creates a DataFrame. It then creates a matplotlib figure and plots the DataFrame onto that figure. Finally it displays the plot in Excel using the `plot` function.



For more details on plotting with PyXLL, and plotting pandas DataFrames in Excel, please see *Plotting with Pandas*.

3.12.7 Reading and Writing DataFrames in Macros

Macro functions can be called from Excel buttons and other controls, as well as from VBA.

Excel macros can automate Excel in the same way as VBA. They can also read and write values in the Excel workbooks.

If you're not already familiar with writing macro functions using PyXLL, please see *Introduction*.

With PyXLL, the entire Excel object model is exposed (see *Python as a VBA Replacement*), but for dealing with pandas DataFrames (and other types) the Python values need to converted to something that Excel understands.

This can be done using PyXLL's `XLCell` class.

In a macro function we can get the Excel `Application` object using `xl_app`. This is the same as the VBA Application object. From that, we can get the `Range` object for which we want to either read or write the DataFrame.

Once we have the `Range` object, we use `XLCell.from_range` to get a PyXLL `XLCell` object. The XLCell object is similar to the Range object. It's simpler in most ways, and can only be used for a contigous range, but it allows us to use PyXLL's type conversion when getting or setting its `value` property.

To read or write the cells' value as a DataFrame we pass `type="dataframe"` to the `XLCell.options` method before accessing the `value` property.

```
from pyxll import xl_macro, xl_app, XLCell

@xl_macro
def read_value_from_excel():
    # Get the Excel.Application COM object
    xl = xl_app()

    # Get a Range in the current active sheet
    xl_range = xl.ActiveSheet.Range("A1:D10")

    # Get an XLCell object from the Range object
    cell = XLCell.from_range(xl_range)

    # Get the value as a DataFrame
    df = cell.options(type="dataframe").value
```

The above macro reads cells `A1:D10` of the currently active sheet as a pandas DataFrame.

The `type` argument to `XLCell.options` is the same as the type used in the *function signature* in the previous sections. It can be parameterized in the same way to control exactly how the conversion is done between the Excel values and the pandas DataFrame.

> **Tip**
>
> You can pass `auto_resize=True` to `XLCell.options` when reading and writing DataFrames and the cell with automatically resize to fit the data, so you don't need to specify the full range.

3.12.8 Passing DataFrames as Objects Instead of Arrays

When passing large DataFrames between Python functions, it is not always necessary to return the full DataFrame to Excel and it can be expensive reconstructing the DataFrame from the Excel range each time. In those cases you can use the `object` return type to return a handle to the Python object. Functions taking the `dataframe` and `series` types can accept object handles.

The following returns a random DataFrame as a Python object, so will appear in Excel as a single cell with a handle to that object:

```
from pyxll import xl_func
import pandas as pd
import numpy as np

@xl_func("int rows, int columns: object")
def random_dataframe(rows, columns):
```

(continues on next page)

(continued from previous page)

```
    data = np.random.rand(rows, columns)
    column_names = [chr(ord('A') + x) for x in range(columns)]
    return pd.DataFrame(data, columns=column_names)
```

The result of a function like this can be passed to another function that expects a DataFrame:

```
@xl_func("dataframe, int: dataframe<index=True>", auto_resize=True)
def dataframe_head(df, num_rows):
    return df.head(num_rows)
```

This allows for large datasets to be used in Excel efficiently, especially where the data set would be cumbersome to deal with in Excel when unpacked.

> **Note**
>
> DataFrames returned from Excel to Python as cached objects are **not copied**. When the object handle is passed to another function, the object retrieved from the cache is the same object that was previously returned.
>
> You should be careful not to modify DataFrames passed this way (i.e. don't make changes *inplace*). Instead create a copy first and modify the copy, or use *inplace=False* if the pandas method you're using supports that.

3.12.9 Pandas Series

pandas Series objects can be used in a similar way to DataFrames.

Instead of using the `dataframe` type as described in the sections below, the `series` type is also available.

Please see *Pandas Types* for details of all of the pandas types available, including the `series` type.

3.12.10 Advanced Usage of the Pandas Type Converters

Sometimes it's useful to be able to convert a range of data into a DataFrame, or a DataFrame into a range of data for Excel, in a context other than function decorated with *`@xl_func`*.

You might have a function that takes the `var` type, which could be a DataFrame depending on other arguments.

Or, you might want to return a DataFrame but want to decide in your function what type parameters to use to control the conversion. For example, you might want to leave it to use the user to decide whether to include the index or not and give that as an argument to your function.

In these cases the function *`get_type_converter`* can be used. For example:

```
from pyxll import get_type_converter

to_dataframe = get_type_converter("var", "dataframe<index=True>")
df = to_dataframe(data)
```

Or the other way:

```
to_array = get_type_converter("dataframe", "var")
data = to_array(df)
```

Combined with PyXLL's `var` type you can pass in values from Excel as a plain array and perform the conversion in your function. Or, you can specify the return type of your function as `var` and convert from the DataFrame before returning the final converted value.

For example:

```
import pandas as pd
from pyxll import xl_func, get_type_converter
```

```
import pandas as pd
from pyxll import xl_func

@xl_func("bool include_index: var")
def return_dataframe(include_index=False):
    """A function that returns a DataFrame."""
    # Create the dataframe
    df = pd.DataFrame({
       "A": [1, 2, 3],
       "B": [4, 5, 6]
    })

    # Get the function to convert from a DataFrame to the PyXLL 'var'
    # return type, specifying the 'index' type parameter.
    to_var = get_type_converter("dataframe", "var", src_kwargs={
        "index": include_index
    })

    # Convert the DataFrame and return the result
    converted_df = to_var(df)
    return converted_df
```

The above function returns a DataFrame. But, rather than using the `dataframe` return type to do the conversion implicitly, it uses the `var` return type and performs the conversion inside the function.

3.13 Customizing the Ribbon

- *Introduction*
- *Creating a Custom Tab*
- *Action Functions*
- *Using Images*
- *Modifying the Ribbon*
- *Merging Ribbon Files*

3.13.1 Introduction

The Excel Ribbon interface can be customized using PyXLL. This enables you to add features to Excel in Python that are properly integrated with Excel for an intuitive user experience.

The ribbon customization is defined using an XML file, referenced in the *config* with the *ribbon* setting. This can be set to a filename relative to the config file, or as as absolute path. If multiple files are listed they will all be read and merged.

The ribbon XML file uses the standard Microsoft *CustomUI* schema. This is the same schema you would use if you were customizing the ribbon using COM, VBA or VSTO and there are various online resources from Microsoft that document it[1].

[1] Microsoft Ribbon Resources

- Ribbon XML
- Walkthrough: Creating a Custom Tab by Using Ribbon XML
- XML Schema Reference

Actions referred to in the ribbon XML file are resolved to Python functions. The full path to the function must be included (e.g. *"module.function"*) and the module must be on the python path so it can be imported. Often it's useful to include the modules used by the ribbon in the *modules* list in the *config* so that when PyXLL is reloaded those modules are also reloaded, but that is not strictly necessary.

3.13.2 Creating a Custom Tab

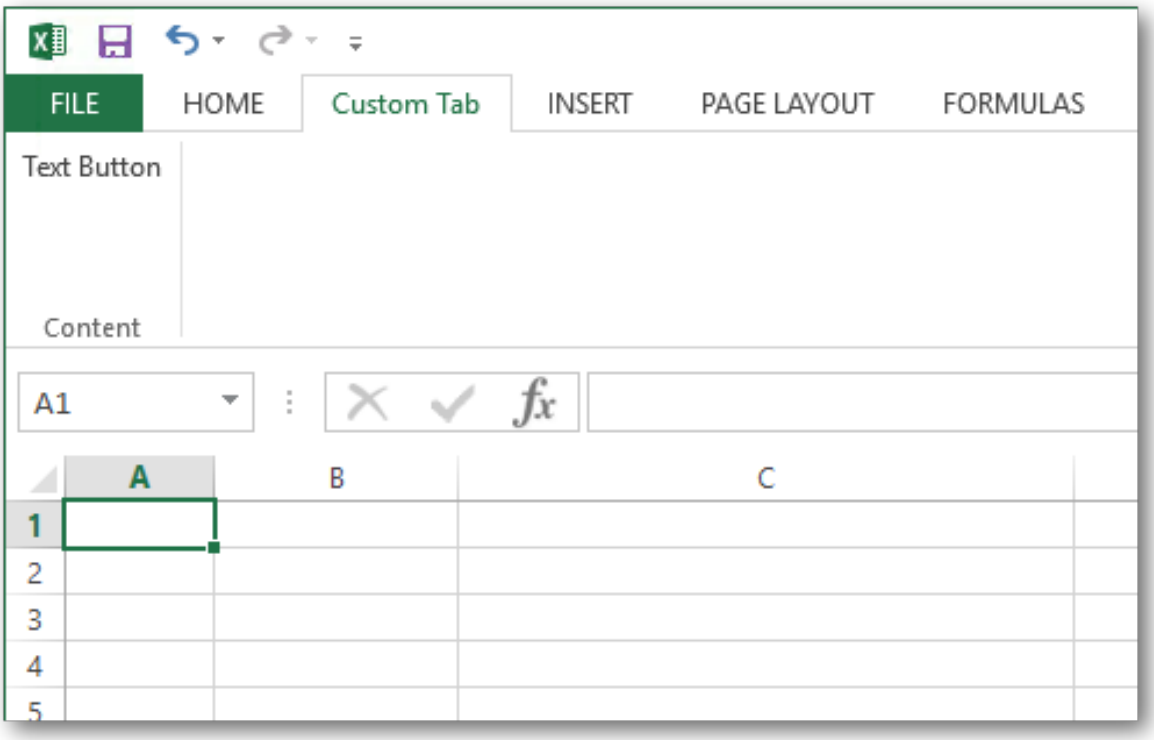
- Create a new ribbon xml file. The one below contains a single tab *Custom Tab* and a single button.

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
    <ribbon>
        <tabs>
            <tab id="CustomTab" label="Custom Tab">
                <group id="ContentGroup" label="Content">
                    <button id="textButton" label="Text Button"/>
                </group>
            </tab>
        </tabs>
    </ribbon>
</customUI>
```

- Set *ribbon* in the config file to the filename of the newly created ribbon XML file.

```
[PYXLL]
ribbon = <full path to xml file>
```

- Start Excel (or reload PyXLL if Excel is already started).



The tab appears in the ribbon with a single text button as specified in the XML file. Clicking on the button doesn't do anything yet.

3.13.3 Action Functions

Anywhere a callback method is expected in the ribbon XML you can use the name of a Python function.

Many of the controls used in the ribbon have an *onAction* attribute. This should be set to the name of a Python function that will handle the action.

- To add an action handler to the example above first modify the XML file to add the *onAction* attribute to the text button

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
    <ribbon>
        <tabs>
            <tab id="CustomTab" label="Custom Tab">
                <group id="ContentGroup" label="Content">
                    <button id="textButton" label="Text Button"
                        onAction="ribbon_functions.on_text_button"/>
                </group>
            </tab>
        </tabs>
    </ribbon>
</customUI>
```

- Create the *ribbon_functions* module with the filename *ribbon_functions.py* and add the *on_text_button* function[2]. Note that the module name isn't important, only that it matches the one used in the xml file.

```
from pyxll import xl_app

def on_text_button(control):
    xl = xl_app()
    xl.Selection.Value = "This text was added by the Ribbon."
```

- Add the module to the pyxll config[3].

```
[PYXLL]
modules = ribbon_functions
```

- Reload PyXLL. The custom tab looks the same but now clicking on the text button calls the Python function.

3.13.4 Using Images

Some controls can use an image to give the ribbon whatever look you like. These controls have an *image* attribute and a *getImage* attribute.

The *image* attribute is set to the filename of an image you want to load. The *getImage* attribute is a function that will return a COM object that implements the *IPicture* interface.

PyXLL provides a function, `load_image`, that loads an image from disk and returns a COM Picture object. This can be used instead of having to do any COM programming in Python to load images.

When images are referenced by filename using the *image* attribute Excel will load them using a basic image handler. This basic image handler is rather limited and doesn't handle PNG files with transparency, so it's recommended to use `load_image` instead. The image handler can be set as the *loadImage* attribute on the *customUI* element.

The following shows the example above with a new button added and the *loadImage* handler set.

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui"
            loadImage="pyxll.load_image">
    <ribbon>
```

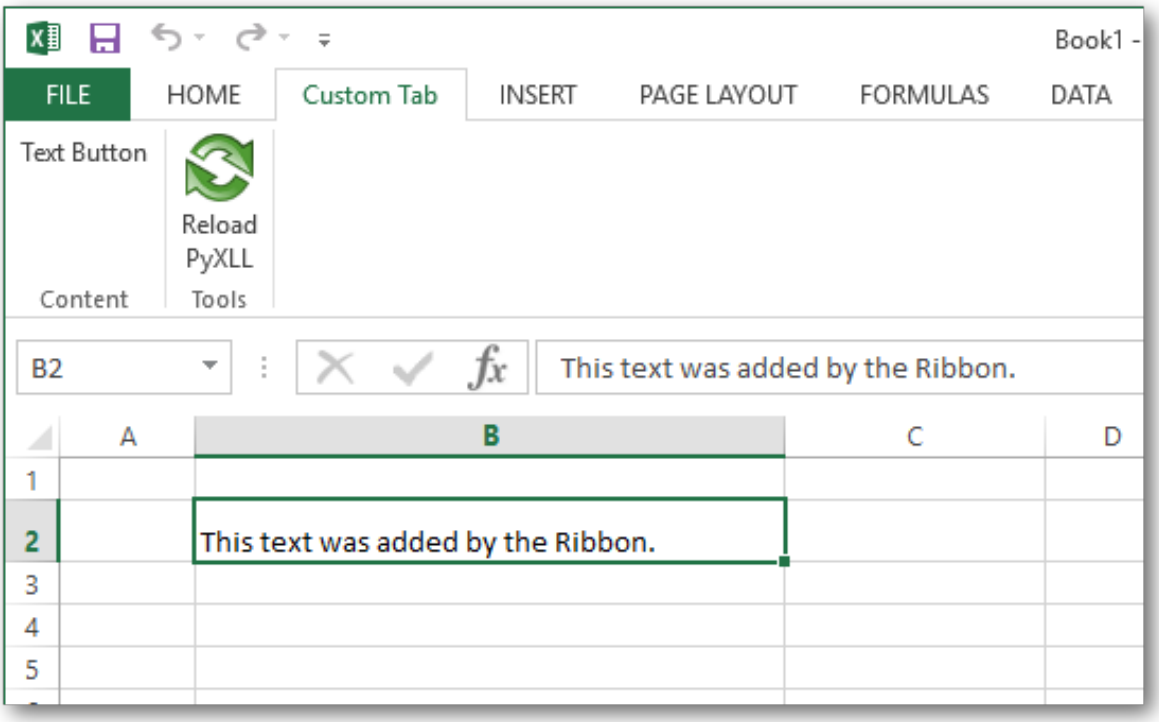
(continues on next page)

[2] The name of the module and function is unimportant, it just has to match the *onAction* attribute in the XML and be on the pythonpath so it can be imported.

[3] This isn't strictly necessary but is helpful as it means the module will be reloaded when PyXLL is reloaded.

(continued from previous page)

```
        <tabs>
            <tab id="CustomTab" label="Custom Tab">
                <group id="ContentGroup" label="Content">
                    <button id="textButton" label="Text Button"
                        onAction="ribbon_functions.on_text_button"/>
                </group>
                <group id="Tools" label="Tools">
                    <button id="Reload"
                            size="large"
                            label="Reload PyXLL"
                            onAction="pyxll.reload"
                            image="reload.png"/>
                </group>
            </tab>
        </tabs>
    </ribbon>
</customUI>
```



If using the *load_image* image loader package resources can also be used as well as filenames. To specify a package resource use for the format `module:resource`.

3.13.5 Modifying the Ribbon

Sometimes its convenient to be able to update the ribbon after Excel has started, without having to change the pyxll.cfg config file.

For example, if your addin is used by multiple users with different roles then one single ribbon may not be applicable for each user. Or, you may want to allow the user to switch between different ribbons depending on what they're working on.

There are some Python functions you can use from your code to update the ribbon:

- *get_ribbon_xml*
- *set_ribbon_xml*

- `set_ribbon_tab`
- `remove_ribbon_tab`

These functions can be used to completely replace the current ribbon (`set_ribbon_xml`) or just to add, replace or remove tabs (`set_ribbon_tab`, `remove_ribbon_tab`).

The ribbon can be updated anywhere from Python code running in PyXLL. Typically this would be when Excel starts up using the `xl_on_open` and `xl_on_reload` event handlers, or from an action function from the current ribbon.

3.13.6 Merging Ribbon Files

If multiple ribbon files are found, either because there are multiple listed using the `ribbon` setting in the pyxll.cfg file or because additional ones have been found via some *entry points* they will be merged automatically.

When merging, any tabs with the same id will be merged into a single tab. Similarly, any groups within those tabs with the same ids will also be merged. You should be careful to use unique ids for all elements so that they do not conflict with any other ribbon elements that might get merged.

The order in which tabs, groups and other elements in groups are merged can be influenced by setting the attributes `insertBefore` and `insertAfter`. These attributes are not part of the ribbon schema but PyXLL will use them when merging the ribbon files. They can be set on `tab` and `group` elements, or any child element of a `group` element. One or the other may be set, but not both. Use these to have elements inserted before or after other elements by their ids.

3.14 Context Menu Functions

- *Introduction*
- *Adding a Python Function to the Context Menu*
- *Creating Sub-Menus*
- *Dynamic Menus*
- *References*

3.14.1 Introduction

Context menus are the menus that appear in Excel when your right-click on something, most usually a cell in the current workbook.

These context menus have become a standard way for users to interact with their spreadsheets and are an efficient way to get to often used functions.

With PyXLL you can add your own Python functions to the context menus.

The context menu customizations are defined using the same XML file used when customizing the Excel ribbon (see *Customizing the Ribbon*). The XML file is referenced in the *config* with the *ribbon* setting. This can be set to a filename relative to the config file, or as an absolute path.

The ribbon XML file uses the standard Microsoft *CustomUI* schema. This is the same schema you would use if you were customizing the ribbon using COM, VBA or VSTO and there are various online resources from Microsoft that document it[1]. For adding context menus, you must use the 2009 version of the schema or later.

Actions referred to in the ribbon XML file are resolved to Python functions. The full path to the function must be included (e.g. *"module.function"*) and the module must be on the python path so it can be imported. Often it's useful to include the modules used by the ribbon in the *modules* list in the *config* so that when PyXLL is reloaded those modules are also reloaded, but that is not strictly necessary.

[1] XML Schema Reference

3.14.2 Adding a Python Function to the Context Menu

- Create a new ribbon xml file, or add the `contextMenus` section from below to your existing ribbon xml file.

  Note that you must use the 2009 version of the schema in the `customUI` element, and the `contextMenus` element must be placed after the `ribbon` element.

```
<?xml version="1.0" encoding="UTF-8"?>
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
    <ribbon>
        <!-- The ribbon and context menus can be specified in the same file -->
    </ribbon>
    <contextMenus>
        <contextMenu idMso="ContextMenuCell">
            <button id="MyButton" label="Toggle Case Upper/Lower/Proper"
                insertBeforeMso="Cut"
                onAction="context_menus.toggle_case"
                imageMso="HappyFace"/>
        </contextMenu>
   </contextMenus>
</customUI>
```

In the xml above, `insertBeforeMso` is used to insert the menu item before the existing "Cut" menu item. This may be removed if you want the item placed at the end of the menu. Also, `imageMso` may be replaced with `image` and set to the path of an image file rather than using one of Excel's built in bitmaps (see `load_image`).

- If you've not already done so, set *ribbon* in the config file to the filename of the ribbon XML file.

```
[PYXLL]
ribbon = <full path to xml file>
```

- Create the *context_menus* module with the filename *context_menus.py* and add the *toggle_case* function. Note that the module name isn't important, only that it matches the one referenced in the `onAction` handler in the xml file above.

```
from pyxll import xl_app

def toggle_case(control):
    """Toggle the case of the currently selected cells"""
    # get the Excel Application object
    xl = xl_app()

    # iterate over the currently selected cells
    for cell in xl.Selection:
        # get the cell value
        value = cell.Value

        # skip any cells that don't contain text
        if not isinstance(value, str):
            continue

        # toggle between upper, lower and proper case
        if value.isupper():
            value = value.lower()
        elif value.islower():
            value = value.title()
        else:
            value = value.upper()
```

(continues on next page)

(continued from previous page)

```
        # set the modified value on the cell
        cell.Value = value
```

- Add the module to the pyxll config[2].

```
[PYXLL]
modules = context_menus
```

- Start Excel (or reload PyXLL if Excel is already started).

  If everything has worked, you will now see the "Toggle Case" item in the context menu when you right click on a cell.

3.14.3 Creating Sub-Menus

Sub-menus can be added to the context menu using the `menu` tag.

The following adds a sub-menu after the "Toggle Case" button added above.

```
<?xml version="1.0" encoding="UTF-8"?>
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
    <ribbon>
        <!-- The ribbon and context menus can be specified in the same file -->
    </ribbon>
    <contextMenus>
        <contextMenu idMso="ContextMenuCell">
            <button id="MyButton" label="Toggle Case Upper/Lower/Proper"
                insertBeforeMso="Cut"
                onAction="context_menus.toggle_case"
                imageMso="HappyFace"/>
            <menu id="MySubMenu" label="Case Menu" insertBeforeMso="Cut"  >
                <button id="Menu1Button1" label="Upper Case"
                    imageMso="U"
                    onAction="context_menus.toupper"/>
                <button id="Menu1Button2" label="Lower Case"
                    imageMso="L"
                    onAction="context_menus.tolower"/>
                <button id="Menu1Button3" label="Proper Case"
                    imageMso="P"
                    onAction="context_menus.toproper"/>
            </menu>
        </contextMenu>
   </contextMenus>
</customUI>
```

The additional buttons use the following code, which you can copy to your *context_menus.py* module.:

```
def tolower(control):
    """Set the currently selected cells to lower case"""
    # get the Excel Application object
    xl = xl_app()

    # iterate over the currently selected cells
    for cell in xl.Selection:
        # get the cell value
        value = cell.Value
```

(continues on next page)

[2] This isn't strictly necessary but is helpful as it means the module will be reloaded when PyXLL is reloaded.

(continued from previous page)

```
        # skip any cells that don't contain text
        if not isinstance(value, str):
            continue

        cell.Value = value.lower()


def toupper(control):
    """Set the currently selected cells to upper case"""
    # get the Excel Application object
    xl = xl_app()

    # iterate over the currently selected cells
    for cell in xl.Selection:
        # get the cell value
        value = cell.Value

        # skip any cells that don't contain text
        if not isinstance(value, str):
            continue

        cell.Value = value.upper()


def toproper(control):
    """Set the currently selected cells to 'proper' case"""
    # get the Excel Application object
    xl = xl_app()

    # iterate over the currently selected cells
    for cell in xl.Selection:
        # get the cell value
        value = cell.Value

        # skip any cells that don't contain text
        if not isinstance(value, str):
            continue

        cell.Value = value.title()
```

3.14.4 Dynamic Menus

As well as statically declaring menus as above, you can also generate menus on the fly in your Python code.

A dynamic menu calls a Python function to get a xml fragment that tells Excel how to display the menu. This can be useful when the items you want to appear in a menu might change.

The following shows how to declare a dynamic menu.

```
<?xml version="1.0" encoding="UTF-8"?>
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
    <ribbon>
        <!-- The ribbon and context menus can be specified in the same file -->
    </ribbon>
    <contextMenus>
        <contextMenu idMso="ContextMenuCell">
```

(continues on next page)

(continued from previous page)

```
            <dynamicMenu id="MyDynamicMenu"
                label= "My Dynamic Menu"
                imageMso="ChangeCase"
                insertBeforeMso="Cut"
                getContent="context_menus.dynamic_menu"/>
        </contextMenu>
    </contextMenus>
</customUI>
```

The *getContent* callback references the *dynamic_menu* function in the *context_menus* module.:

```
def dynamic_menu(control):
    """Return an xml fragment for the dynamic menu"""
    xml = """
         <menu xmlns="http://schemas.microsoft.com/office/2009/07/customui">
            <button id="Menu2Button1" label="Upper Case"
                imageMso="U"
                onAction="context_menus.toupper"/>

            <button id="Menu2Button2" label="Lower Case"
                imageMso="L"
                onAction="context_menus.tolower"/>

            <button id="Menu2Button3" label="Proper Case"
                imageMso="P"
                onAction="context_menus.toproper"/>
         </menu>
    """
    return xml
```

3.14.5 References

- XML Schema Reference
- https://msdn.microsoft.com/en-us/library/dd926324(v=office.12).aspx
- http://interoperability.blob.core.windows.net/files/MS-CUSTOMUI2/{[}MS-CUSTOMUI2{]}-150904.pdf

3.15 Working with Tables

New in PyXLL 5.8

In the section *Introduction* it is explained how Python functions can be exposed a Excel macros, and how these macros can read and write Excel worksheet values.

Excel Tables can be used to make managing and analyzing a group of related data easier. In Excel, you can turn a range of cells into an Excel Table (previously known as an Excel list).

PyXLL can read and write Excel Tables in a similar way to how ranges can be read and written. Tables can be created and updated from Python data using macro functions.

- *Writing a Table*
- *Reading a Table*
- *Updating a Table*

- *Preserving Table Columns*
- *Tables and Worksheet Functions*
- *Advanced Features*
  - *Naming Tables*
  - *Advanced Customization*

3.15.1 Writing a Table

Writing an Excel Table from a macro works in a very similar way to writing an array of data as a range.

If you're not already familiar with writing a range of Python data from a macro function please see *Writing Python Values to Excel* for a full explaination.

To write an Excel Table we use the `XLCell` class, and set the `XLCell.value` property.

As previously, we specify the data type using `XLCell.options`, but this time we use the special `table` type to tell PyXLL to write the data as a Table instead of a range.

The `table` type takes one type parameter, which is the data type we want PyXLL to use when converting the Python type to Excel values. To write a `DataFrame` we would use `table<dataframe>`. The inner type can also be parameterized, for example, to include the index of the `DataFrame` we would use `table<dataframe<index=True>>`.

```
from pyxll import xl_macro, XLCell

@xl_macro
def write_excel_table():
    # Get an XLCell object for the cell 'A1' in the active sheet.
    # We could fully specify the range, for example "[Book1]Sheet1!A1" if
    # needed, or use a COM Range object instead of the address string.
    # The table will be written with this cell as the top left of the table.
    cell = XLCell.from_range("A1")

    # Create the DataFrame we want to write to Excel as a table
    df = your_code_to_construct_the_dataframe()

    # Write the DataFrame to Excel as a Table
    cell.options(type="table<dataframe>").value = df
```

The cell we use when writing the Table is the top left cell where we want the table to be written to. If there is not enough space to write the table, a `#SPILL!` error will be written to Excel and a `SpillError` Python exception will be raised.

When writing a table, we do not need to pass `auto_resize=True` to `XLCell.options`, the size of the table written will always match the same of the data even if this is not specified.

3.15.2 Reading a Table

Reading the data from an Excel Table is no different from reading it from a Range. A table is really just a range with some formatting added!

If you're not already familiar with reading an Excel range into Python from a macro function please see macros-read-from-excel for a full explaination.

When getting the `XLCell` instance in order to read the data, you can pass the Range corresponding to the entire table. Or, to make things simpler, you can pass any cell from within the table and use the `auto_resize=True` option to `XLCell.options`. When this is used on a table, the entire table will be used automatically.

```
from pyxll import xl_macro, XLCell

@xl_macro
def read_excel_table():
    # Get the XLCell object for the top left of the existing table
    cell = XLCell.from_range("A1")

    # Read the entire table into a DataFrame by using the `auto_resize=True`
    # option to XLCell.options.
    df = cell.options(auto_resize=True, type="dataframe").value
```

3.15.3 Updating a Table

To update a table written previously all that is required is to write to one of the cells in the table using the same method explained above. The existing table will be updated instead of creating a new table.

If the new data is larger than the current table the table will be expanded, or if the new data is smaller the table will be contracted. If there is not enough space to expand the table a `#SPILL!` error will be written to Excel and a *`SpillError`* Python exception will be raised.

```
from pyxll import xl_macro, XLCell

@xl_macro
def update_excel_table():
    # Get the XLCell object for the top left of the existing table
    cell = XLCell.from_range("A1")

    # Read the entire table into a DataFrame
    df = cell.options(auto_resize=True, type="dataframe").value

    # Make some changes to the DataFrame
    new_df = your_code_to_update_the_dataframe(df)

    # Update the table in Excel
    cell.options(type="table<dataframe>").value = new_df
```

3.15.4 Preserving Table Columns

New in PyXLL 5.11

When updating an existing table, by default, the table columns in Excel will be updated to match the columns of the DataFrame.

If you want to keep the columns as they are in Excel and only update the columns where the headers match the Python DataFrame columns you can specify `preserve_columns=True` as a type paramter to the `table` type.

This is useful where you wish to allow the Excel user to add calculated columns to the table, or remove columns they don't wish to include. Without specifying `preserve_columns=True` changes to the columns in the Excel table would be lost when updating, but with `preserve_columns=True` the columns are preserved.

```
from pyxll import xl_macro, XLCell

@xl_macro
def update_excel_table():
    # Get the XLCell object for the top left of the existing table
    cell = XLCell.from_range("A1")

    # Get the updated DataFrame
```

(continues on next page)

(continued from previous page)

```
    df = ...

    # Update the table in Excel without modifying the columns
    cell.options(type="table<dataframe, preserve_columns=True>").value = new_df
```

3.15.5 Tables and Worksheet Functions

In all of the code above we have used `@xl_macro` to read and write Excel Tables.

This is because reading and writing values from Excel using the `XLCell` class must always be done from an Excel macro.

Sometimes, it is desirable to be able to *return* data from a worksheet function. We can return DataFrames from worksheet functions using `@xl_func` as dynamic arrays, but what about using tables?

To do that we have to use `schedule_call`. This schedules a Python function to be run in such a way that it is safe to do what we can otherwise only do in a macro.

Using `schedule_call` we can schedule a function that will write Python data as a table in Excel. We would like to write the table near to where the worksheet function was called from, and to do that we use `xlfCaller`.

`xlfCaller` returns the `XLCell` of the function's calling cell. Using `XLCell.offset` we can get one cell below the calling cell and use that as the top left corner of our table.

For example:

```
from pyxll import xl_func, schedule_call, xlfCaller

@xl_func
def table_from_function():
    """A worksheet function that writes to a table."""
    # Get the XLCell this function was called from
    cell = xlfCaller()

    # Create the DataFrame we want to write to Excel as a table
    df = your_code_to_construct_the_dataframe()

    # Get the top left cell of the table we're going to write,
    # one row below the calling cell.
    top_left = cell.offset(rows=1, columns=0)

    # An inner function that will be called in the future and will
    # write the DataFrame to a table below the calling cell.
    def write_table():
        top_left.options(type="table<dataframe>").value = df

    # Schedule the call to 'write_table' that could otherwise
    # only be called as part of an Excel macro.
    schedule_call(write_table)

    return "[OK]"
```

The above method using `schedule_call` works around the fact that we can only write tables from a macro. It schedules a function to write the table after Excel has finished calculating, when it is possible to do so.

3.15.6 Advanced Features

Above we've seen how the `table` type can be used with *XLCell.options* to write Python data to an Excel table.

The same can be acheived using the *Table* class, and using the `table` type is really just shorthand for this.

The below example shows how to construct a *Table* instance and set that as the *XLCell.value*. This has the same effect as using the `table<dataframe>` type.

```
from pyxll import xl_macro, XLCell, Table

@xl_macro
def write_excel_table():
    # Get an XLCell object for the cell 'A1' in the active sheet.
    # We could fully specify the range, for example "[Book1]Sheet1!A1" if
    # needed, or use a COM Range object instead of the address string.
    # The table will be written with this cell as the top left of the table.
    cell = XLCell.from_range("A1")

    # Create the DataFrame we want to write to Excel as a table
    df = your_code_to_construct_the_dataframe()

    # Construct a Table instance, wrapping our DataFrame
    table = Table(df, type="dataframe")

    # Write the table to Excel
    cell.value = table
```

This is a small change but it allows us to access some of the more advanced features of PyXLL's table capabilites.

Naming Tables

When setting *XLCell.value* using an instance of the *Table* class, you can provide the name to use when creating the Excel Table.

The *Table* class constructor takes a kwargs `name`. It also has the `Table.name` attribute, allowing you to query the table named used after writing the table to Excel.

```
from pyxll import xl_macro, XLCell, Table

@xl_macro
def write_named_excel_table():
    # Get an XLCell object for the cell 'A1' in the active sheet.
    # We could fully specify the range, for example "[Book1]Sheet1!A1" if
    # needed, or use a COM Range object instead of the address string.
    # The table will be written with this cell as the top left of the table.
    cell = XLCell.from_range("A1")

    # Create the DataFrame we want to write to Excel as a table
    df = your_code_to_construct_the_dataframe()

    # Construct a named Table instance, wrapping our DataFrame
    table = Table(df, type="dataframe", name="MyNamedTable")

    # Write the table to Excel.
    # When creating a new table the table name will be used.
    cell.value = table

    # The table name is accessible from the table.name attribute.
    # You can use this to get the auto-generated name if no name
```

(continues on next page)

(continued from previous page)

```
    # was specified.
    name = table.name
    print(f"Table name = {name}")
```

Advanced Customization

Full control over how tables are written to Excel is possible by implementing a class derived from `Table` or `TableBase`.

The `TableBase` class defines the methods required for tables to be written to Excel. These can be used as escape hatches, allowing your own code to function differently to the default `Table` class.

Any class implemented with `TableBase` as a base class can be used when setting `XLCell.value`.

> **Note**
>
> Knowledge of the *Excel Object Model* is required to write an class derived from `Table` or `TableBase`.

When writing a table to Excel, the following happens:

1. `TableBase.find_table` is called to see if there is an existing `ListObject` object.
2. If no existing `ListObject` is found, `TableBase.create_table` is called.
3. If the `ListObject` size is different from that returned by `TableBase.rows` and `TableBase.columns`, `TableBase.resize_table` is called.
4. `TableBase.update_table` is called to update the data in the `ListObject` table object.
5. Finally, `TableBase.apply_filters` and `TableBase.apply_sorting` are called to apply any filtering and sorting required to the table.

`Table` provides the default implementation for these methods.

See the *Tables API Reference* for details of the `Table` and `TableBase` classes, including the methods that need to be implemented.

3.16 Python as a VBA Replacement

- *The Excel Object Model*
- *Accessing the Excel Object Model in Python*
- *Differences between VBA and Python*
  - *Case Sensitivity*
  - *Calling Methods*
  - *Named Arguments*
  - *Properties*
  - *Properties with Arguments*
  - *Implicit Objects and 'With'*
  - *Indexing Collections*
- *Enums and Constant Values*
- *Excel and Threading*

- *Notes on Debugging*

Everything you can write in VBA can be done in Python. This page contains information that will help you translate your VBA code into Python.

Please note that the *Excel Object Model* is part of Excel and documented by Microsoft. The classes and methods from that API used in this documentation are not part of PyXLL, and so please refer to the Excel Object Model documentation for more details about their use.

See also *Introduction*.

3.16.1 The Excel Object Model

When programming in VBA you interact with the *Excel Object Model*. For example, when writing

```
Sub Macro1()
    Range("B11:K11").Select
EndSub
```

what you are doing is constructing a Range object and calling the Select method on it. The Range object is part of the *Excel Object Model*.

Most of what people talk about in reference to VBA in Excel is actually the Excel Object Model, rather than the VBA language itself. Once you understand how to interact with the Excel Object Model from Python then replacing your VBA code with Python code becomes straightforward.

The Excel Object Model is well documented by Microsoft as part of the Office VBA Reference.

The first hurdle people often face when starting to write Excel macros in Python is finding documentation for the Excel Python classes. Once you realise that the Object Model is the same across Python and VBA you will see that the classes documented in the Office VBA Reference are the exact same classes that you use from Python, and so you can use the same documentation even though the example code may be written in VBA.

3.16.2 Accessing the Excel Object Model in Python

The Excel Object Model is made available to all languages using COM. Python has a couple of packages that make calling COM interfaces very easy. If you know nothing about COM then there's no need to worry as you don't need to in order to call the Excel COM API from Python.

The top-level object in the Excel Object Model is the Application object. This represents the Excel application, and all other objects are accessed via this object.

PyXLL provides a helper function, *`xl_app`*, for retrieving the Excel Application object. By default, it uses the Python package `win32com`, which is part of the `pywin32` package[1].

If you don't already have the `pywin32` package installed you can do so using `pip`:

```
pip install pywin32
```

Or if you are using Anaconda you can use `conda`:

```
conda install pywin32
```

You can use *`xl_app`* to access the Excel Application object from an Excel macro. The following example shows how to re-write the `Macro1` VBA code sample from the section above.

Note that in VBA there is an implicit object, which related to where the VBA Sub (macro) was written. Commonly, VBA code is written directly on a sheet, and the sheet is implied in various calls. In the `Macro1` example above, the `Range` is actually a method on the sheet that macro was written on. In Python, we need to explicitly get the current active sheet instead.

[1] If you prefer to use `comtypes` instead of `win32com` you can still use *`xl_app`* by passing `com_package='comtypes'`.

```
from pyxll import xl_macro, xl_app


@xl_macro
def macro1():
    xl = xl_app()

    # 'xl' is an instance of the Excel.Application object

    # Get the current ActiveSheet (same as in VBA)
    sheet = xl.ActiveSheet

    # Call the 'Range' method on the Sheet
    xl_range = sheet.Range('B11:K11')

    # Call the 'Select' method on the Range.
    # Note the parentheses which are not required in VBA but are in Python.
    xl_range.Select()
```

You can call into Excel using the Excel Object Model from macros and menu functions, and use a sub-set of the Excel functionality from worksheet functions, where more care must be taken because the functions are called during Excel's calculation process.

You can remove these restrictions by calling the PyXLL *schedule_call* function to schedule a Python function to be called in a way that lets you use the Excel Object Model safely. For example, it's not possible to update worksheet cell values from a worksheet function, but it is possible to schedule a call using *schedule_call* and have that call update the worksheet after Excel has finished calculating.

For testing, it can also be helpful to call into Excel from a Python prompt (or a Jupyter notebook). This can also be done using *xl_app*, and in that case the first open Excel instance found will be returned.

You might try this using `win32com` directly rather than *xl_app*. We do not advise this when calling your Python code from Excel however, as it may return an Excel instance other than the one you expect.

```
from win32com.client.gencache import EnsureDispatch

# Get the first open Excel.Application found, or launch a new one
xl = EnsureDispatch('Excel.Application')
```

3.16.3 Differences between VBA and Python

Case Sensitivity

Python is case sensitive. This means that code fragments like `r.Value` and `r.value` are different (note the capital `V` in the first case. In VBA they would be treated the same, but in Python you have to pay attention to the case you use in your code.

If something is not working as expected, check the PyXLL log file. Any uncaught exceptions will be logged there, and if you have attempted to access a property using the wrong case then you will probably see an `AttributeError` exception.

Calling Methods

In Python, parentheses (`()`) are **always** used when calling a method. In VBA, they may be omitted. Neglecting to add parentheses in Python will result in the method not being called, so it's important to be aware of which class attributes are methods (and must therefore be called) and which are properties (whose values are available by reference).

For example, the method `Select` on the `Range` type is a method and so must be called with parentheses in Python, but in VBA they can be, and usually are, omitted.

```
' Select is a method and is called without parentheses in VBA
Range("B11:K11").Select
```

```
from pyxll import xl_app
xl = xl_app()

# In Python, the parentheses are necessary to call the method
xl.Range('B11:K11').Select()
```

Keyword arguments may be passed in both VBA and Python, but in Python keyword arguments use = instead of the `:=` used in VBA.

Accessing properties does not require parentheses, and doing so will give unexpected results! For example, the `range.Value` property will return the value of the range. Adding `()` to it will attempt to call that value, and as the value will not be callable it will result in an error.

```
from pyxll import xl_app
xl = xl_app()

# Value is a property and so no parentheses are used
value = xl.Range('B11:K11').Value
```

Named Arguments

In VBA, named arguments are passed using `Name := Value`. In Python, the syntax is slightly different and only the equals sign is used. One other important difference is that VBA is *not* case-sensitive but Python is. This applies to argument names as well as method and property names.

In VBA, you might write

```
Set myRange = Application.InputBox(prompt := "Sample", type := 8)
```

If you look at the documentation for Application.InputBox you will see that the argument names are cased different from this, and are actually 'Prompt' and 'Type'. In Python, you can't get away with getting the case wrong like you can in VBA.

In Python, this same method would be called as

```
from pyxll import xl_app
xl = xl_app()

my_range = xl.InputBox(Prompt='Sample', Type=8)
```

Properties

Both VBA and Python support properties. Accessing a property from an object is similar in both languages. For example, to fetch *ActiveSheet* property from the *Application* object you would do the following in VBA:

```
Set mySheet = Application.ActiveSheet
```

In Python, the syntax used is identical:

```
from pyxll import xl_app
xl = xl_app()

my_sheet = xl.ActiveSheet
```

Properties with Arguments

In VBA, the distinction between methods and properties is somewhat blurred as properties in VBA can take arguments. In Python, a property never takes arguments. To get around this difference, the `win32com` Excel classes have *Get* and *Set* methods for properties that take arguments, in addition to the property.

The Range.Offset property is an example of a property that takes optional arguments. If called with no arguments it simply returns the same *Range* object. To call it with arguments in Python, the *GetOffset* method must be used instead of the *Offset* property.

The following code activates the cell three columns to the right of and three rows down from the active cell on *Sheet1*:

```
Worksheets("Sheet1").Activate
ActiveCell.Offset(rowOffset:=3, columnOffset:=3).Activate
```

To convert this to Python we must make the following changes:

- Replace the *Offset* property with the *GetOffset* method in order to pass the arguemnts.
- Replace *rowOffset* and *columnOffset* *RowOffset* and *ColumnOffset* as specified in the Range.Offset documentation.
- Call the *Activate* method by adding parentheses in both places it's used.

```
from pyxll import xl_app
xl = xl_app()

xl.Worksheets('Sheet1').Activate()
xl.ActiveCell.GetOffset(RowOffset=3, ColumnOffset=3).Activate()
```

Note

You may wonder, what would happen if you were to use the *Offset* property in Python? As you may by now expect, it would fail - but not perhaps in the way you might think.

If you were to call `xl.ActiveCell.Offset(RowOffset=3, ColumnOffset=3)` the the result would be that the parameter *RowOffset* is invalid. What's actually happening is that when `xl.ActiveCell.Offset` is evaluated, the *Offset* property returns a *Range* equivalent to *ActiveCell*, and that *Range* is then called.

Range has a *default method*. In Python this translates to the *Range* class being *callable*, and calling it calls the default method.

The default method for *Range* is *Item*, and so this bit of code is actually equivalent to `xl.ActiveCell.Offset.Item(RowOffset=3, ColumnOffset=3)`. The *Item* method doesn't expect a *RowOffset* argument, and so that's why it fails in this way.

Implicit Objects and 'With'

When writing VBA code, the code is usually written 'on' an object like a WorkBook or a Sheet. That object is used implicitly when writing VBA code.

If using a 'With..End' statement in VBA, the target of the 'With' statement becomes the implicit object.

If a property is not found on the current implicit object (e.g. the one specified in a 'With..End' statement) then the next one is tried (e.g. the Worksheet the Sub routine is associated with). Finally, the Excel Application object is implicitly used.

In Python there is no implicit object and the object you want to reference must be specified explicitly.

For example, the following VBA code selects a range and alters the column width.

```
Sub Macro2()
    ' ActiveSheet is a property of the Application
    Set ws = ActiveSheet

    With ws
        ' Range is a method of the Sheet
        Set r = Range("A1:B10")

        ' Call Select on the Range
        r.Select
    End With

    ' Selection is a property of the Application
    Selection.ColumnWidth = 4
End Sub
```

To write the same code in Python each object has to be referenced explicitly.

```
from pyxll import xl_macro, xl_app

@xl_macro
def macro2():
    # Get the Excel.Application instance
    xl = xl_app()

    # Get the active sheet
    ws = xl.ActiveSheet

    # Get the range from the sheet
    r = ws.Range('A1:B10')

    # Call Select on the Range
    r.Select()

    # Change the ColumnWidth property on the selection
    xl.Selection.ColumnWidth = 4
```

Indexing Collections

VBA uses parentheses (`()`) for calling methods and for indexing into collections.

In Python, square braces (`[]`) are used for indexing into collections.

Care should be taken when indexing into Excel collections, as Excel uses an index offset of 1 whereas Python uses 0. This means that to get the first item in a normal Python collection you would use index 0, but when accessing collections from the Excel Object Model you would use 1.

3.16.4 Enums and Constant Values

When writing VBA enum values are directly accessible in the global scope. For example, you can write

```
Set cell = Range("A1")
Set cell2 = cell.End(Direction:=xlDown)
```

In Python, these enum values are available as constants in the `win32com.client.constants` package. The code above would be re-written in Python as follows

```
from pyxll import xl_app
from win32com.client import constants

xl = xl_app()

cell = xl.Range('A1')
cell2 = cell.End(Direction=constants.xlDown)
```

3.16.5 Excel and Threading

In VBA everything always runs on Excel's main thread. In Python we have multi-threading support and sometimes to perform a long running task you may want to run code on a background thread.

The standard Python `threading` module is a convenient way to run code on a background thread in Python. However, we have to be careful about how we call back into Excel from a background thread. As VBA has no ability to use threads the Excel objects are not written in a such a way that they can be used across different threads. Attempting to do so may result in serious problems and even cause Excel to crash!

In order to be able to work with multiple threads and still call back into Excel PyXLL has the *schedule_call* function. This is used to schedule a Python function to run on Excel's main thread in such a way that the Excel objects can be used safely. Whenever you are working with threads and need to use the Excel API you should use *schedule_call*.

For example, you might use an Excel macro to start a long running task and when that task is complete write the result back to Excel. Instead of writing the result back to Excel from the background thread, use *schedule_call* instead.

```
from pyxll import xl_macro, xl_app, schedule_call
import threading

@xl_macro
def start_task():
    # Here we're being called from a macro on the main thread
    # so it's safe to use pyxll.xl_app.
    xl = xl_app()
    value = float(xl.Selection.Value)

    # Use a background thread for a long running task.
    # Be careful not to pass any Excel objects to the background thread!
    thread = threading.Thread(target=long_running_task, args=(value,))
    thread.start()

# This runs on a background thread
def long_running_task(value):
    # Do some work that takes some time
    result = ...

    # We shouldn't write the result back to Excel here as we are on
    # a background thread. Instead use pyxll.schedule_call to write
    # the result back to Excel.
    schedule_call(write_result, result, "A1")

# This is called via pyxll.schedule_call
def write_result(result, address):
    # Now we're back on the main thread and it's safe to use pyxll.xl_app
    xl = xl_app()
    cell = xl.Range(address)
    cell.Value = result
```

3.16.6 Notes on Debugging

The Excel VBA editor has integrating debugging so you can step through the code and see what's happening at each stage.

When writing Python code it is sometimes easier to write the code *outside* of Excel in your Python IDE before adapting it to be called from Excel as a macro or menu function etc.

When calling your code from Excel, remember that any uncaught exceptions will be printed to the PyXLL log file and so that should always be the first place you look to find what's going wrong.

If you find that you need to be able to step through your Python code as it is being executed in Excel you will need a Python IDE that supports remote debugging. Remote debugging is how debuggers connect to an external process that they didn't start themselves.

You can find instructions for debugging Python code running in Excel in this blog post Debugging Your Python Excel Add-In.

3.17 Menu Functions

- *Custom Menu Items*
- *New Menus*
- *Sub-Menus*

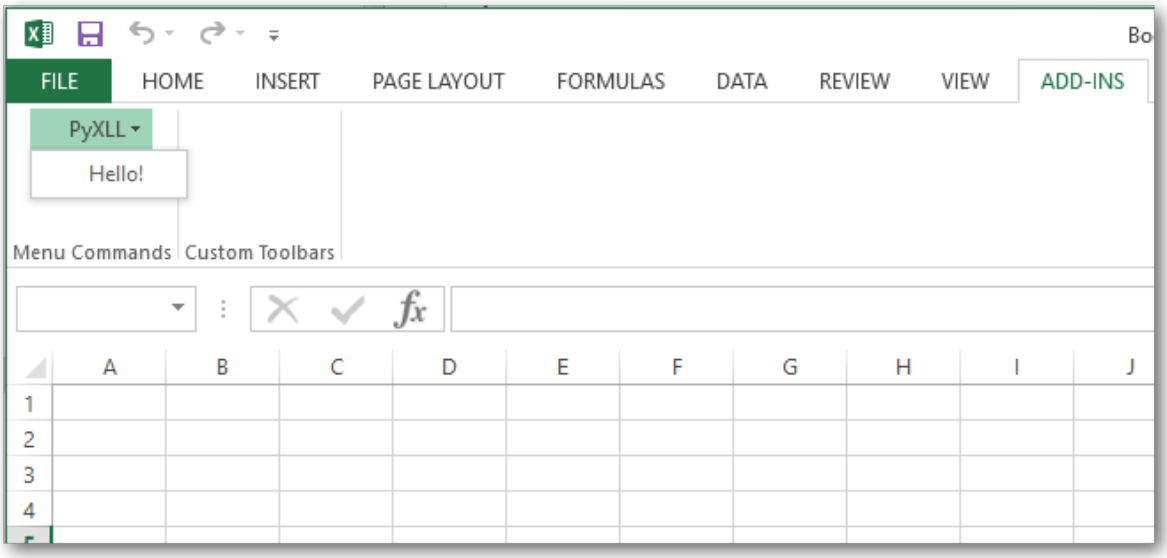
3.17.1 Custom Menu Items

The `@xl_menu` decorator is used to expose a python function as a menu callback. PyXLL creates the menu item for you, and when it's selected your python function is called. That python function can call back into Excel using `win32com` or `comtypes` to make changes to the current sheet or workbook.

Different menus can be created and you can also create submenus. The order in which the items appear is controlled by optional keyword arguments to the `@xl_menu` decorator.

Here's a very simple example that displays a message box when the user selects the menu item:

```
from pyxll import xl_menu, xlcAlert

@xl_menu("Hello!")
def on_hello():
    xlcAlert("Hello!")
```



Menu items may modify the current workbook, or in fact do anything that you can do via the Excel COM API. This allows you to do anything in Python that you previously would have had to have done in VBA.

Below is an example that uses *xl_app* to get the Excel Application COM object and modify the current selection. You will need to have `win32com` or `comtypes` installed for this.

```
from pyxll import xl_menu, xl_app

@xl_menu("win32com menu item")
def win32com_menu_item():
    # get the Excel Application object
    xl = xl_app()

    # get the current selected range
    selection = xl.Selection

    # set some text to the selection
    selection.Value = "Hello!"
```

3.17.2 New Menus

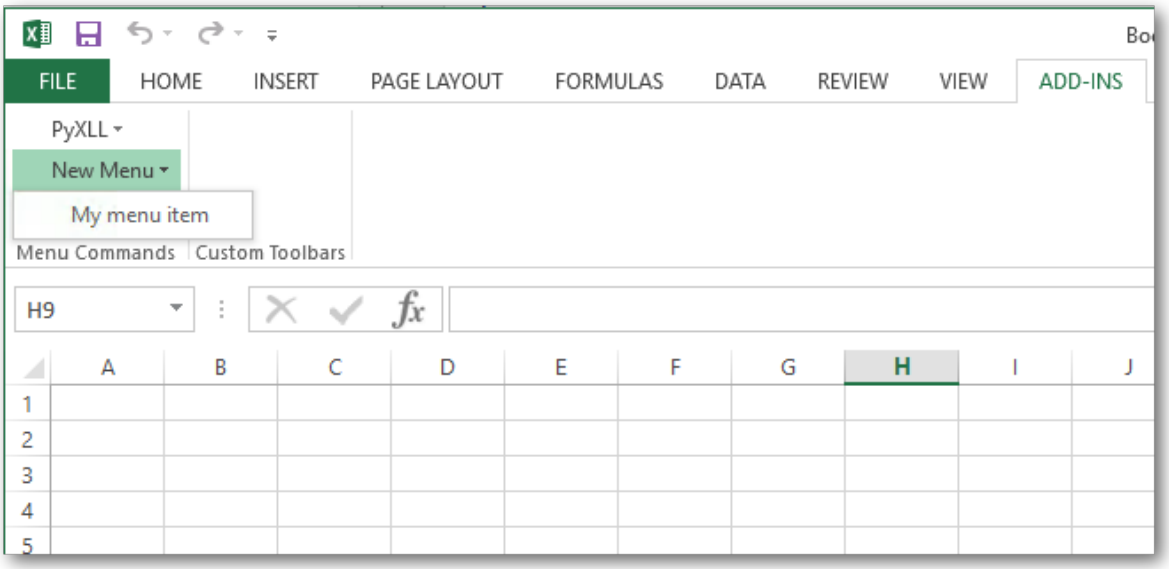
As well as adding menu items to the main PyXLL addin menu it's possible to create entirely new menus.

To create a new menu, use the *menu* keyword argument to the *@xl_menu* decorator.

In addition, if you want to control the order in which menus are added you may use the *menu_order* integer keyword argument. The higher the value, the later in the ordering the menu will be added. The menu order my also be set in the config (see configuration).

Below is a modification of an earlier menu example that puts the menu item in a new menu, called "New Menu":

```
from pyxll import xl_menu, xlcAlert

@xl_menu("My menu item", menu="New Menu")
def my_menu_item():
    xlcAlert("new menu example")
```



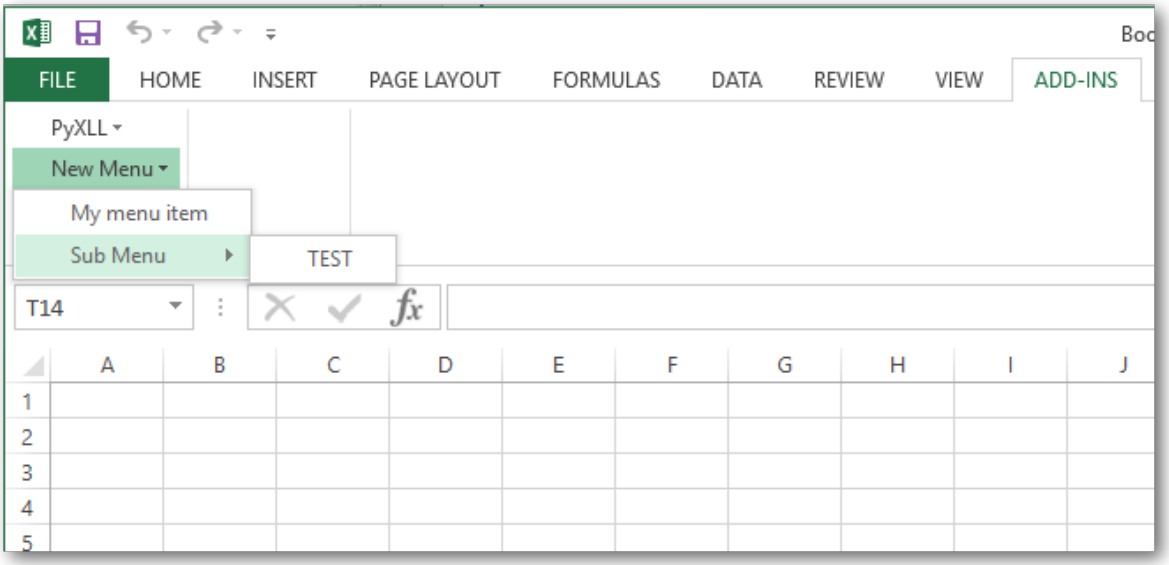
3.17.3 Sub-Menus

Sub-menus may also be created. To add an item to a sub-menu, use the *sub_menu* keyword argument to the `@xl_menu` decorator.

All sub-menu items share the same *sub_menu* argument. The ordering of the items within the submenu is controlled by the *sub_order* integer keyword argument. In the case of sub-menus, the *order* keyword argument controls the order of the sub-menu within the parent menu. The menu order my also be set in the config (see configuration).

For example, to add the sub-menu item "TEST" to the sub-menu "Sub Menu" of the main menu "My Menu", you would use a decorator as illustrated by the following code:

```
from pyxll import xl_menu, xlcAlert

@xl_menu("TEST", menu="New Menu", sub_menu="Sub Menu")
def my_submenu_item():
    xlcAlert("sub menu example")
```



3.18 Reloading and Rebinding

- *Introduction*
- *How to Reload PyXLL*
  - *Reload Manually*
  - *Automatic Reloading*
  - *Programmatic Reloading*
- *Deep Reloading*
- *Rebinding*

3.18.1 Introduction

When writing Python code to be used in Excel, there's no need to shut down Excel and restart it every time you make a change to your code.

Instead, you can simply tell PyXLL to reload your Python code so you can test it out immediately.

When reloading, the default behaviour is for PyXLL to only reload the Python modules listed in the `modules` list on your pyxll.cfg config file. Optionally, PyXLL can also reload *all* the modules that those modules depend on - this is called *deep reloading*. Deep reloading can take a bit longer than just reloading the modules listed in the config, but can be helpful when working on larger projects.

There are different options that affect how and when your Python code is reloaded, which are explained in this document. The different configuration options are also documented in the *Configuring PyXLL* section of the documentation.

3.18.2 How to Reload PyXLL

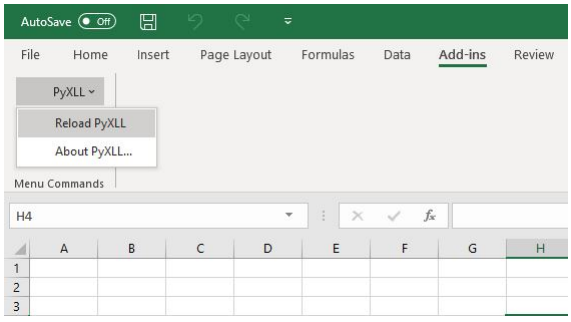
Before you can reload your Python modules with PyXLL, you need to make sure you have `developer_mode` enabled in your pyxll.cfg file.

```
[PYXLL]
developer_mode = 1
```

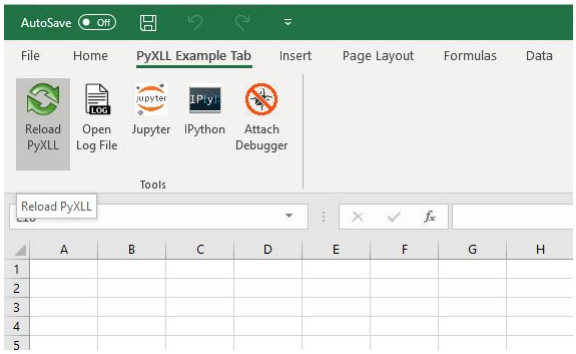
This setting enables reloading and adds the "Reload PyXLL" menu item to Excel. It is enabled by default.

Reload Manually

After working on some changes to your code you can tell PyXLL to reload your modules by selecting "Reload PyXLL" from the PyXLL menu in the Add-Ins tab.



You can also configure the Excel ribbon to have a "Reload" button. This is done for you in the example *ribbon.xml* file.



A simple ribbon file with just the "Reload" button would look like this

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
    <ribbon>
        <tabs>
            <tab id="PyXLL" label="PyXLL">
                <group id="Tools" label="Tools">
                    <button id="Reload"
                            size="large"
                            label="Reload PyXLL"
                            onAction="pyxll.reload" />
                </group>
            </tab>
    </ribbon>
</customUI>
```

Note the "onAction" attribute is set to "pyxll.reload". This binds that ribbon button to PyXLL's reload function.

You can read more about configuring the ribbon *here*.

Automatic Reloading

Rather than have to reload manually every time you make a change to your code, PyXLL can watch and reload automatically as soon as any of your files are saved.

To enable automatic reloading, set `auto_reload = 1` in the `[PYXLL]` section of your config file.

```
[PYXLL]
auto_reload = 1
```

When automatic reloading is enabled, changes to the following files will cause PyXLL to reload:

- Python modules
- PyXLL config files
- Ribbon XML files

Automatic reloading works with *deep reloading*. If deep reloading is enabled, then any change to a Python module that be reloaded will cause PyXLL to trigger a reload. If deep reloading is not enabled, then only the Python modules listed in the PyXLL config will trigger a reload.

> **Warning**
>
> Automatic reloading is only available from PyXLL 4.3 onwards.

Programmatic Reloading

It is possible to reload PyXLL programmatically via the Python function `reload` or by calling the Excel macro *pyxll_reload*.

Calling either the Python function or the Excel macro will cause PyXLL to reload shortly after. The reload does not happen immediately, but after the current function or macro has completed.

3.18.3 Deep Reloading

The default behaviour when reloading is that only the modules listed in the pyxll.cfg config file are reloaded.

When working on more complex projects it is normal to have Python code organized into packages, and to have PyXLL functions in many different Python modules. Instead of listing all of them in the config file they can be imported from a single module.

For example, you might have a directory structure something like the following

```
my_excel_addin
├── __init__.py
├── functions.py
└── macros.py
```

And in *my_excel_addin/__init__.py* you might import *functions* and *macros*.

```
from . import functions
from . import macros
```

In your pyxll.cfg file, you would only need to list *my_excel_addin*.

Listing 3: my_excel_addin/__init__.py

```
[PYXLL]
modules =
    my_excel_addin
```

When you reload PyXLL, only *my_excel_addin* would be reloaded, and so changes to *my_excel_addin.functions* or *my_excel_addin.macros* or any other imported modules wouldn't be discovered.

With **deep reloading**, PyXLL determines the dependencies between your imported modules and reloads *all* of the module dependencies, in the correct order.

To enable deep reloading, set `deep_reload = 1` in the `[PYXLL]` section of your config file.

```
[PYXLL]
deep_reload = 1
```

Not all modules can be reloaded. Sometimes because of the way some modules are written, they won't reload cleanly. Circular dependencies between modules is a common reason for packages to not reload cleanly, and Python cannot reload C extension modules.

If you are having trouble with a particular package or module not reloading cleanly, you can exclude it from being reloaded during the deep reload. To do so, list the modules you want excluded in the `deep_reload_exclude` list in your PyXLL config file.

As deep reloading can take longer than normal reloading, you can limit what modules and packages are included by setting `deep_reload_include` in your PyXLL config file. In the example above, because everything we're

interested in is contained in the *my_excel_addin* package, adding *my_excel_addin* to the `deep_reload_include` list would limit reloading to modules in that package.

> **Warning**
>
> Starting with PyXLL 4.3 onwards, packages in the *site-packages* folder are no longer included when deep reloading.
>
> To include modules in *site-packages*, set `deep_reload_include_site_packages = 1` in the `[PYXLL]` section of your config file.

3.18.4 Rebinding

As well as reloading, it is also possible to tell PyXLL to re-create its bindings between the imported Python code and Excel. This is referred to as *rebinding*.

Rebinding can be useful, for example, when importing modules dynamically and updating the Excel functions after the import is complete, without reloading.

By default rebinding occurs automatically whenever a new `@xl_func`, `@xl_macro` or `@xl_menu` decorator is called.

Automatic rebinding can be disabled by setting the following in your pyxll.cfg file:

```
[PYXLL]
auto_rebind = 0
```

If automatic rebinding has been disabled you can still tell PyXLL to rebind by calling the `rebind` function.

For example:

```
from pyxll import xl_macro, rebind

@xl_macro
def import_new_functions():
    """Import a new module and then call 'rebind' to tell PyXLL to update"""
    module = __import__("...")

    # Now the module has been imported and declared new UDFs using @xl_func
    # tell PyXLL to update it's Excel bindings.
    rebind()
```

PyXLL also declares an Excel macro `pyxll_rebind` that you can call from VBA to do the same as the Python `rebind` function.

3.19 Error Handling

- *Introduction*
- *Standard Error Handlers*
- *Custom Error Handlers*
- *Passing Errors to and from Worksheet Functions*
- *Retrieving Error Information*
- *Error Propagation*
  - *Custom Error Values*

– *Propagating Errors by Default*

3.19.1 Introduction

Any time a PyXLL function raises an uncaught Exception, it will be written to the log file as an error.

If you need to figure out what is going wrong, the log file should be your first piece of evidence. The location of the log file is set in the PyXLL config file, and by default it is in the *logs* folder alongside the PyXLL add-in.

In addition to the log file, PyXLL provides ways of handling errors to present them to the user directly when they occur. The full exception and stack trace are always written to the log file, but in many cases providing the user with some details of the error is sufficient to let them understand the problem without having to resort to the log file.

For example, if a worksheet function fails Excel's default behaviour is to show an error like `#NA`. Consider the following function:

```
@xl_func
def my_udf(x, y):
    if not 1 <= x <= 100:
        raise ValueError("Expected x to be between 1 and 100")
    return do_something(x, y)
```

If you call this from Excel with x outside of 1 and 100, without an error handler the user will see `#VALUE!`. They can look in the log file to see the full error, but an error handler can be used to return something more helpful. Using the standard error handler *pyxll.error_handler* `##ValueError: Expected x to be between 1 and 100` would be returned[1].

The configured error handler will be called for all types of functions when an uncaught Exception is raised, not simply worksheet functions.

3.19.2 Standard Error Handlers

PyXLL provides two standard error handlers to choose from.

- pyxll.error_handler
- pyxll.quiet_error_handler

These are configured by setting *error_handler* in the configuration file, e.g.:

```
[PYXLL]
error_handler = pyxll.error_handler
```

The following table shows how the two different error handlers behave for the different sources of errors:

Error Source	pyxll.error_handler	pyxll.quiet_error_handler	No Handler
Worksheet Function	Return error as string	Return error as string	Nothing (returns #NA! etc.)
Macro	Return error as string	Return error as string	Nothing (returns #NA! etc.)
Menu Item	Show error in message box	Do nothing	Do nothing
Ribbon Action	Show error in message box	Do nothing	Do nothing
Module Import	Show error in message box	Do nothing	Do nothing

[1] Sometimes it's useful to actually return an error code (eg `#VALUE!`) to Excel. For example, if using the *=ISERROR* Excel function. In those cases, you should not set an error handler, or use a custom error handler that returns a Python Exception.

3.19.3 Custom Error Handlers

For cases where the provided error handling isn't suitable, you can provide your own error handler.

An error handler is simply a Python function that you reference from your configuration file, including the module name, for example:

```
[PYXLL]
error_handler = my_error_handler.error_handler
```

The error handler takes four[2] arguments, *context* (`ErrorContext`), *exc_type*, *exc_value* and *exc_traceback*. *context* is a `ErrorContext` object that contains additional information about the error that has occurred, such as the type of function that was being called.

The following shows a custom error handler that returns a string if the function type was a worksheet function (UDF) or macro. For all other types, it calls `pyxll.error_handler`, delegating error handling to PyXLL's standard handler.

```
from pyxll import error_handler as standard_error_handler

def error_handler(context, exc_type, exc_value, exc_traceback):
    """Custom PyXLL error handler"""

    # For UDFs return a preview of the error as a single line
    if context.error_type in (ErrorContext.Type.UDF, ErrorContext.Type.MACRO):
        error = "##" + getattr(exc_type, "__name__", "Error")
        msg = str(exc_value)
        if msg:
            error += ": " + msg
        return error

    # For all other error types call the standard error handler
    return standard_error_handler(context, exc_type, exc_value, exc_traceback)
```

PyXLL will still log the exception, so there is no need to do that in your handler.

If you want your error handler to return an error code to Excel instead of a string, return the Exception value. Python Exceptions are converted to Excel errors as per the following table.

Excel error	Python Exception type
#NULL!	`LookupError`
#DIV/0!	`ZeroDivisionError`
#VALUE!	`ValueError`
#REF!	`ReferenceError`
#NAME!	`NameError`
#NUM!	`ArithmeticError`
#NA!	`RuntimeError`

3.19.4 Passing Errors to and from Worksheet Functions

Excel errors can be passed to, or returned from, Python functions.

Similarly, Python functions can return specific errors to Excel by returning Python Exception objects.

PyXLL maps Excel errors to Python Exception types as specified in the following table:

[2] Prior to PyXLL 4.3, error handlers only took three arguments and didn't have the context argument.
PyXLL is backwards compatible with older versions. If you have an old error handler that only takes three arguments, this will be handled automatically and that error handler will only be called for worksheet functions (UDFs) and macros.

Excel error	Python exception type
#NULL!	`LookupError`
#DIV/0!	`ZeroDivisionError`
#VALUE!	`ValueError`
#REF!	`ReferenceError`
#NAME!	`NameError`
#NUM!	`ArithmeticError`
#NA!	`RuntimeError`

For example, the following function `foo` will receive a `ValueError` object when called as `=foo(#VALUE!)`, and the function `bar` returns `#DIV/0!` when called as `=bar()`:

Listing 4: my_error_handler.py

```
from pyxll import xl_func
import typing

@xl_func
def foo(x: typing.Any) -> str:
    # x can take any Excel value, including errors.
    # When called with '#VALUE!', x will be 'ValueError()'
    return f"x = {repr(x)}"

@xl_func
def bar() -> Exception:
    # Return #DIV/0! to Excel
    return ZeroDivisionError()
```

For details about how worksheet functions can accept and return errors please see *Error Types* from the *Argument and Return Types* section of the user guide.

3.19.5 Retrieving Error Information

When a Python function is called an Excel as a worksheet function, if an uncaught exception is raised PyXLL caches the exception and traceback as well as logging it to the log file.

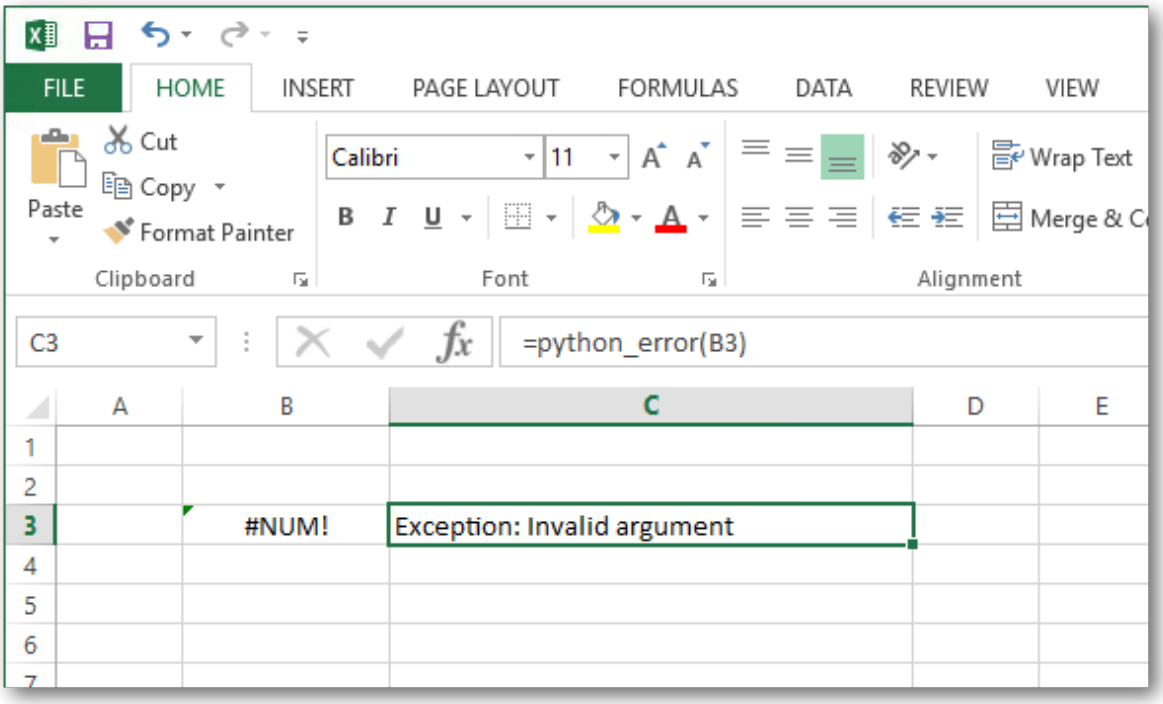
The last exception raised while evaluating a cell can be retrieved by calling PyXLL's `get_last_error` function.

`get_last_error` takes a cell reference and returns the last error for that cell as a tuple of *(exception type, exception value, traceback)*. The cell reference may either be a `XLCell` or a COM *Range* object (the exact type of which depend on the *com_package* setting in the *config*.

The cache used by PyXLL to store thrown exceptions is limited to a maximum size, and so if there are more cells with errors than the cache size the least recently thrown exceptions are discarded. The cache size may be set via the *error_cache_size* setting in the *config*.

The error cache can also be disabled entirely by setting *disable_error_cache* in the *config* (**new in PyXLL 5.13**), in which case `get_last_error` will not return anything for a failed function call.

When a cell returns a value and no exception is thrown any previous error is **not** discarded, because to do so would add additional performance overhead to every function call.




```
from pyxll import xl_func, xl_menu, xl_version, get_last_error
import traceback

@xl_func("xl_cell: string")
def python_error(cell):
    """Call with a cell reference to get the last Python error"""
    exc_type, exc_value, exc_traceback = get_last_error(cell)
    if exc_type is None:
        return "No error"

    return "".join(traceback.format_exception_only(exc_type, exc_value))

@xl_menu("Show last error")
def show_last_error():
    """Select a cell and then use this menu item to see the last error"""
    selection = xl_app().Selection
    exc_type, exc_value, exc_traceback = get_last_error(selection)

    if exc_type is None:
        xlcAlert("No error found for the selected cell")
        return

    msg = "".join(traceback.format_exception(exc_type, exc_value, exc_traceback))
    if xl_version() < 12:
        msg = msg[:254]

    xlcAlert(msg)
```

3.19.6 Error Propagation

New in PyXLL 5.12

When a worksheet function or macro is given an argument of the wrong type that often results in a `#VALUE!` error, can make it difficult to trace what the actual error is.

By setting `propagate_errors=True` in the *@xl_func* or *@xl_macro* decorators, PyXLL will check to see if any of the arguments are errors before calling the function.

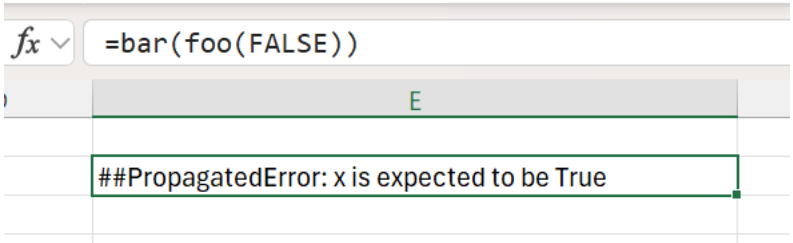
If `propagate_errors` is set, and if an argument is an error, then instead of calling the function a new `PropagatedError` exception is raised. This new `PropagatedError` exception wraps the original exception, and the standard error handler converts this to a string so the user can see the error that has been propagated.

For example, suppose you had the following Python functions:

```
from pyxll import xl_func

@xl_func
def bar(x: bool):
    if not x:
        raise RuntimeError("x is expected to be True")
    return True

@xl_func(propagate_errors=True)
def foo(x: bool):
    return f"Called with {x=}"
```

In Excel, if you called `=foo(bar(FALSE))` *without* any error propagation, this would result in a `#VALUE!` error since the argument to `foo` would be an error string, and not the expected boolean type.

With error propagation enabled, `=foo(bar(FALSE))` will result in a propagated error message. The inner function `bar` returns an error, but because `foo` has `propagate_errors=True` set the PyXLL add-in will detect that it is being called with an error and not a boolean value and propagate the error instead of calling the function.



Logging is suppressed for propagated errors and so only the original error is logged.

Custom Error Values

By default, error propagation works by testing each argument to see if it is either an Excel error type, or a string that matches the error strings converted by the Standard Error Handlers.

If you are using Custom Error Handlers you may need to use a different test to see if an argument should be considered an error or not, for the purpose of error propagation.

This is done by using the `propagate_errors_filter` kwarg to the *@xl_func* or *@xl_macro* decorators.

`propagate_errors_filter` can be set to a function that takes the argument value and returns `True` if the argument should be considered an error, or `False` otherwise.

A typical errors filter function might look as follows:

```
from pyxll import xl_func
import re

def custom_error_filter(arg):
    # Always treat exceptions as errors
    if isinstance(arg, Exception):
        return True

    # If the argument is a string, check if it looks like an error
```

(continues on next page)

(continued from previous page)

```
    # converted in your error handler
    if isinstance(arg, str):
        if re.match(r"<your error pattern>", arg):
            return True

    # Otherwise, it's not an error
    return False

@xl_func(propagate_errors=True,
         propagate_errors_filter=custom_error_filter)
def foo(x: bool):
    return f"Called with {x=}"
```

Propagating Errors by Default

Errors are not propagated by default. This is intentional because checking each argument to test if it's an error introduces a small additional overhead for each function that uses this feature.

The convenience of having error propagation enabled for all functions might outweigh the small performance hit, but that will depend on your specific use case and preferences. While it is not enabled by default it can (optionally) be enabled for all functions, or for all functions in specific modules or packages.

To enable error propagation for all functions use the *Default Decorator Parameters* configuration in the `pyxll.cfg` file.

Using the *Default Decorator Parameters*, you can provide default values for both `propagate_errors` and `propagate_errors_filter` instead of specifying these parameters on every function.

3.20 Deploying your add-in

Everything needed to run your Python code using the PyXLL add-in can be packaged together and deployed to other users of your organization.

PyXLL is licensed per-user so you should check your license covers all the users of the add-in. If you need to add more users to your license you can do so using the , or contact us if you are not sure.

> **Warning**
>
> Redistribution of the PyXLL add-in to unlicensed users is not permitted by the Software License Agreement.

For the add-in to work your end users will need to have the following. Each can be pre-configured and packaged so that the end user doesn't need to install each individually or do any configuration themselves:

1. Your Python code
2. A Python environment with any dependencies installed
3. The PyXLL add-in, configured and loaded in Excel

One of the benefits of using PyXLL is that the code is separated from the Excel workbooks so that updates to the code can be deployed without having to change each workbook that depends on it.

There are various ways to make the items listed above available to your end users. Which methods you choose will depend on your specific use case and requirements. It's quite usual to end up using a combination of the methods described below.

- *Sharing everything on a network drive*
- *Using a standalone zip file*
- *Building an installer*
- *Using a common pyxll.cfg file*
- *Using a startup script to install and update Python code*
- *Deploying the Python Environment*
- *Adding the PyXLL add-in to Excel*
- *Setuptools Entry Points*
  - *modules entry point*
  - *ribbon entry point*

3.20.1 Sharing everything on a network drive

This is the simplest option as it allows your Python code to be deployed centrally, ensuring that all users see the same code at all times.

As long as a user can read from the network drive, PyXLL can be configured to read Python modules from there. This ensures that you don't need to copy code around and that all users are always referencing the same version of your code.

The PyXLL configuration can be shared by using the `external_config` option in the pyxll.cfg file (see *Using a common pyxll.cfg file*). You can list multiple external configs which will get merged together when PyXLL loads.

The PyXLL config file itself can also be on a network drive. The environment variable `PYXLL_CONFIG_FILE` can be set to tell PyXLL to load a config file from somewhere other than the default location. You can use environment variables inside the config file so that logging gets written to the user's local storage.

When deploying code on a network drive you will want to make the folder read-only to your users to ensure no accidental updates occur.

To update the Python code, rather than updating in-place it is good practice to create a new folder with the updated code and then change the pythonpath in the shared config file to that new folder. This way if there are any problems then you can quickly revert to the previous folder, and it also avoids problems with certain files (e.g. dll and pyd files) becoming locked while users have then open.

A typical structure for this shared folder would be (with the folder names changed to suit your requirements):

- modules-{version}

  The folder containing your Python code

- python-{version}

  (Optional) Folder containing your Python environment

- shared_pyxll.cfg

  This would specify `pythonpath = ./modules-{version}`, your modules list, any other shared settings, and optionally `executable = .\python-{version}\pythonw.exe` if you are including the Python environment on the network drive.

In your main `pyxll.cfg` file you would include the shared_pyxll.cfg file using `external_config = X:\network\folders\pyxll\shared_pyxll.cfg`.

You would choose whether or not to include the Python environment on the shared folder or not depending on whether or not each user would already have a suitable local Python environment and how fast your network drive is. Loading Python from a slow network drive can slow down starting Excel, in which case one of the options below may be more suitable.

To install the PyXLL add-in you can either load the pyxll.xll add-in manually into Excel, or your could script it using the `pyxll activate` command (see *Adding the PyXLL add-in to Excel*).

3.20.2 Using a standalone zip file

Similarly to putting everything on a network drive, as above, you can combine everything into a single zip archive for distribution to your users. You can include everything, including a Python environment, and pre-configure PyXLL to reference it using a relative path.

To install the PyXLL add-in you would unzip the archive on the PC where you want it to be installed and then load the PyXLL add-in in Excel. This can be scripted, and you can script the step of adding PyXLL to Excel using the `pyxll activate` command (see *Adding the PyXLL add-in to Excel*).

Once installed, updates can be made using a *Startup Script*. See *Using a startup script to install and update Python code* for more details about that.

A typical structure for this zip file would be (changing the folder names to suit your requirements):

- modules

  The folder containing your Python code

- python

  Folder containing your Python environment

- pyxll.xll

  The PyXLL add-in

- pyxll.cfg

  This would specify everything relative to the location of this file. For example, `pythonpath = .\modules` and `executable = .\python\pythonw.exe`.

Once you have a zip file containing the above you could use a batch script to do the installation. The script would do the following:

- Copy and unzip `your_pyxll_archive.zip` to `C:\Your\Local\PyXLL\Install`
- Run `C:\Your\Local\PyXLL\Install\python\python.exe -m pyxll activate --non-interactive C:\Your\Local\PyXLL\Install\pyxll.xll`

This script could be run directly by your end users to install the add-in, or pushed out by your systems team or IT administrators. It could even be configured to run each time the user logs in.

3.20.3 Building an installer

If you prefer to build an installer or MSI instead of using a zip file and script as above then that is also possible. Some organizations prefer this method as they already have mechanisms for pushing out installers to users' PCs.

As with the zip file approach above, the Python runtime can be bundled alongside PyXLL and your Python code into a single standalone installer.

For detailed instructions and an example project for building an MSI installer, see the pyxll-installer project on GitHub.

3.20.4 Using a common pyxll.cfg file

The PyXLL config be shared so that each user gets the same configuration, and so updates to the config can be made once rather than on each PC. This is done by setting the *external_config* option in the pyxll.cfg file.

Each user still has their own pyxll.cfg file with any settings specific to them (if any), but they also use the *external_config* option to source in one or more shared configs.

The external config can be a file on a network drive or a URL.

```
[PYXLL]
external_config = https://intranet/pyxll/pyxll-shared.cfg
```

If more than one external config is required the external_config setting accepts a list of files and URLs.

If it is not desirable for each user to have their own pyxll.cfg file then the environment variable `PYXLL_CONFIG_FILE` can be set to tell PyXLL where to load the config from. This could be a path on a network drive or a URL.

When using a shared config typically you don't want the log file to be written to the same place for every user. You can use environment variables in the config file to avoid this, eg

```
[LOG]
path = %(USERPROFILE)s/pyxll/logs
```

See *Environment Variables* for more details.

3.20.5 Using a startup script to install and update Python code

Importing Python code from a network drive can have some disadvantages. It requires a fast network, and even then it can be slow to import the modules. It may also be against your coroprate IT policy to deploy code via a network drive because it lacks sufficient control, or it just may not suit your deployment needs.

Using a *startup script* you can check what version of your Python code is currently deployed and download the latest if necessary. Once downloaded the code is on the local PC and so importing it will be fast. When updates are needed the script will detect there's a newer version of the code available and download it.

Such a script might look something like this:

```
SET VERSION=v1
SET PYTHON_FOLDER=.\python-code-%VERSION%

REM No need to download anything if we already have the latest
IF EXIST %PYTHON_FOLDER% THEN GOTO END

REM Download and unzip the latest code
wget https://intranet/pyxll/python-code-%VERSION%.tar.gz
tar -xzf python-code-%VERSION%.tar.gz --directory %PYTHON_FOLDER%

ECHO Latest code has been downloaded to .\python-code-%VERSION%
:END
```

The above script is just an illustration and your script would be different depending on your needs. It could also be a Powershell script rather than a plain batch script.

To get this script to run when Excel starts we use the *startup_script* option in the pyxll.cfg file. This is set to the the path of the script to run, or it can be a URL. By using a URL (or a location on a network drive) whenever we want to deploy a different version of the code to all of our users we only have to update the version number in the script.

```
[PYXLL]
startup_script = https://intranet/pyxll/startup-script.cmd
```

Now the script runs when Excel starts, but the code downloaded isn't on our Python Path and so won't be able to be imported. Because we're using a different folder for each version of the code we can't hard-code the path in our pyxll.cfg file.

Within a startup script run by PyXLL you can run various commands, including getting and setting PyXLL options. There's a command `pyxll-set-option` that we can use to set the `pythonpath` option to the correct folder:

```
SET VERSION=v1
SET PYTHON_FOLDER=.\python-code-%VERSION%
ECHO pyxll-set-option PYTHON pythonpath %PYTHON_FOLDER%
```

The `pyxll-set-option` command is run by echoing it from the batch script. PyXLL sees this in the output from the script and updates the `pythonpath` option. Calling `pyxll-set-option` for a multi-line option like `pythonpath` appends to it rather than replacing it.

There are several other commands available from a startup script. See *Startup Script* for more details.

3.20.6 Deploying the Python Environment

The Python environment and many of the Python packages your code depends on are likely to change less often than your main Python code. They do still need to be available to PyXLL for it to work however.

This doesn't mean that Python actually needs to be installed on the local PC.

PyXLL can be configured to use any Python environment as long as it is accessible by the user. This means you can take a Python environment and copy it to a network drive and have PyXLL reference it from there. For example, where below X: is a mapped network drive:

```
[PYTHON]
executable = X:\PyXLL\Python\pythonw.exe
```

As long as the Python environment on the network drive is complete, this will work fine.

A very useful tool for creating a Python environment suitable for being relocated to a network drive is conda-pack.

Note, using a `venv` doesn't create a complete Python environment and still requires the base Python install and so cannot be used in this way.

Referencing the Python environment from a network drive will not be as fast to load as if it was installed on the local PC. Another option is to use the *startup_script* option and copy a Python environment locally on demand when Excel starts.

A startup script that downloads a Python environment would look something along the lines of the following:

```
SET VERSION=v1
SET PYTHON_ENV=.\python37-%VERSION%

REM No need to download anything if we already have the latest
IF EXIST %PYTHON_ENV% THEN GOTO DONE

REM Download and unzip the Python environment
wget https://intranet/pyxll/python37-%VERSION%.tar.gz
tar -xzf python37-%PYTHON_ENV%.tar.gz --directory %PYTHON_ENV%

ECHO Latest Python environment has been downloaded to .\python37-%VERSION%
:DONE

REM Set the PyXLL executable option
ECHO pyxll-set-option PYTHON executable %PYTHON_ENV%\pythonw.exe
```

3.20.7 Adding the PyXLL add-in to Excel

Once you have made your Python code and Python environment available to your end user, either by copying them to the local PC or by making them available on the network drive, you will need to add the PyXLL add-in so that it gets loaded each time Excel starts.

This can either be done manually, if the end user is comfortable managing their own Excel add-ins, or it can be scripted for them.

To install the PyXLL add-in from a script you can use the *pyxll activate* sub-command. The `activate` sub-command installs the PyXLL Excel add-in into Excel so that it is loaded whenever Excel starts.

When using the `pyxll` command from a specific Python environment, rather than the system default or currently active environment, it can be easier to use `python -m pyxll` instead of running the `pyxll` command directly. For example, if you have the PyXLL add-in copied locally to `C:\PyXLL\pyxll.xll` and Python copied locally as `C:\PyXLL\python\python.exe` you would run:

```
> C:\PyXLL\python\python.exe -m pyxll activate --non-interactive C:\PyXLL\pyxll.xll
```

The `--non-interactive` switch prevents the `pyxll activate` command from asking the user for input or confirmation which makes it suitable to be called from a script.

3.20.8 Setuptools Entry Points

When distributing Python code it is usual to package it up into a *wheel* file using `setuptools`. This allows consumers of your package to install it easily using pip (the Python package manager).

You can distribute a Python package containing PyXLL functionality in the same way. To avoid the end user of your package from having to manually configure their pyxll.cfg file, PyXLL looks for its entry points in any installed packages.

The entry points are configured in your setup.py file used to build your package. PyXLL supports two entry points, `pyxll.modules` and `pyxll.ribbon`.

A simple setup.py file to build a package called `your_package` might look as follows:

```
from setuptools import setup, find_packages

setup(
    name="your_package",
    description="Your package description",
    version="0.0.1",
    packages=find_packages(),
    entry_points={
        "pyxll": [
            "modules = your_package:pyxll_modules",
            "ribbon = your_package:pyxll_ribbon"
        ]
    }
)
```

To build a wheel using your setup.py file you run `python setup.py bdist_wheel`.

The user of your package would install the wheel by running `pip install <wheel file>`.

The entry points listed in this setup.py file are `your_package:pyxll_modules` for the `pyxll/modules` entry point and `your_package:pyxll_ribbon` for the `pyxll/ribbon` entry point.

Each entry point is a reference to a function. It's these functions that PyXLL will call to configure itself to load your package automatically without the consumer of your package having to modify their pyxll.cfg file.

modules entry point

The modules entry point is a function that returns a list of module names for PyXLL to import when it loads.

In the above `your_package` example, suppose `your_package` contained two sub-modules `your_package.xlfuncs` and `your_package.xlmacros` that you want to be loaded when PyXLL starts. To make that happen you would write the `your_package.pyxll_modules` entry point function return both packages.

Listing 5: setup.py

```
def pyxll_modules():
    """entry point referenced in setup.py"""
    return [
        "your_package.xlfuncs",
        "your_package.xlmacros",
    ]
```

This is of course just an example. The entry point function could be in any package (including a subpackage) that you configure in your `setup.py` file.

ribbon entry point

The ribbon entry point can be used to add ribbon controls to the Excel ribbon in addition to whatever ribbon controls are configured in the pyxll.cfg file.

The ribbon entry point function should return either a single ribbon xml resource or a list of ribbon xml resources. These will be merged with any other ribbon files loaded and combined to create the custom ribbon UI in Excel.

See *Customizing the Ribbon* for the specifics of how to create a ribbon xml file.

In the above `your_package` example, suppose you had also included a "ribbon.xml" resource in the wheel and you wanted to add that to the Excel ribbon. Your ribbon entry point would load the XML data from the resource (or it could load it from a file) and return that for PyXLL to use when building the ribbon.

Listing 6: your_package/__init__.py

```
import pkg_resources

def pyxll_ribbon():
    """entry point referenced in setup.py"""
    # Load the XML resource
    ribbon_xml = pkg_resources.resource_string(__name__, "ribbon.xml")

    # Return the ribbon XML resource for PyXLL to load
    return ribbon_xml
```

If you are using files instead of package resources then you can also tell PyXLL the filename of the XML file. If you have images referenced in your ribbon xml using relative paths then providing the filename will ensure that PyXLL can load the images relative to the correct path.

Listing 7: your_package/__init__.py

```
import os

def pyxll_ribbon():
    """entry point referenced in setup.py"""
    # Get the ribbon XML filename
    ribbon_file = os.path.join(os.path.dirname(__file__), "ribbon.xml")

    # Load the xml data
    with open(ribbon_file) as fh:
        ribbon_xml = fh.read()

    # Return the ribbon XML resource with its file name for PyXLL to load
    return (ribbon_file, ribbon_xml)
```

When using the `load_image` function as your image loaded in the ribbon xml file, images can be referenced either by filename or as a package resource name if you are building them into your package.

If you have multiple ribbon resources then the `pyxll.ribbon` entry point function may return a list of resources or a list of (filename, resource) tuples.

Warning

If your Python packages are on a network drive it can be slow to look for entry points, which may result in slow start times for Excel.

You can prevent PyXLL from looking for entry points by setting the following in your pyxll.cfg file:

Listing 8: your_package/__init__.py

```
[PYXLL]
ignore_entry_points = 1
```

3.21 Workbook Metadata

Some PyXLL features will add XML metadata to the Excel workbook when saving.

Features that use this metadata are:

- *Recalculating On Open*
- *Saving Objects in the Workbook*
- *Cell Formatting*[1]

Metadata used by PyXLL is added to the workbook as a *CustomXMLPart*, which is part of the workbook document.

The *CustomXMLPart* is saved in the workbook using an XML namespace specific to the PyXLL add-in so as not to conflict with data saved by other add-ins. If you have specified a name for your add-in using the `name` setting that will be used to avoid conflict with any other PyXLL add-ins you may have loaded.

If you prefer to specify the namespace to use instead of having PyXLL use it's own namespace you can do so by setting `metadata_custom_xml_namespace` in the `PYXLL` section of your pyxll.cfg file.

```
[PYXLL]
metadata_custom_xml_namespace = urn:your_name:metadata
```

To disable writing any metadata you can set `disable_saving_metadata = 1` in the `PYXLL` section of your pyxll.cfg file. Note that this will affect all PyXLL features that require metadata.

```
[PYXLL]
disable_saving_metadata = 1
```

[1] Custom formatting only requires metadata when a custom formatter is applied to a Dynamic Array function.

CHAPTER

FOUR

API REFERENCE

4.1 Worksheet Functions

These decorators, functions and classes are used to expose Python functions to Excel as worksheet functions (UDFs).

Please see *Worksheet Functions* for complete details on writing Excel worksheet functions in Python.

See also *Type Conversion* and *Real Time Data*.

- `@xl_func`
- `XLAsyncHandle`

@xl_func(*signature: str = None, category: str = 'PyXLL', help_topic: str = '', thread_safe: bool = False, macro: bool = False, allow_abort: bool = None, volatile: bool = None, disable_function_wizard_calc: bool = None, disable_replace_calc: bool = None, name: str = None, auto_resize: bool = None, hidden: bool = False, transpose: bool = False, recalc_on_open: bool = None, recalc_on_reload: bool = None, formatter: Formatter = None, nan_value: Any = undefined, posinf_value: Any = undefined, neginf_value: Any = undefined, none_value: Any = undefined, decimals_to_float: bool = None, lru_cache: int | bool = None, cache: Mapping = None*)

xl_func is decorator used to expose python functions to Excel. Functions exposed in this way can be called from formulas in an Excel worksheet and appear in the Excel function wizard.

Parameters

- **signature** (*string*) – string specifying the argument types and, optionally, their names and the return type. If the return type isn't specified the var type is assumed. eg:

  `"int x, string y: double"` for a function that takes two arguments, x and y and returns a double.

  `"float x"` or `"float x: var"` for a function that takes a float x and returns a variant type.

  If no signature is provided the argument and return types will be inferred from any type annotations, and if there are no type annotations then the types will be assumed to be `var`.

  See *Standard Types* for the built-in types that can be used in the signature.

- **category** (*string*) – String that sets the category in the Excel function wizard the exposed function will appear under.
- **help_topic** (*string*) – Path of the help file (.chm) or URL that will be available from the function wizard in Excel.
- **thread_safe** (*boolean*) – Indicates whether the function is thread-safe or not. If True the function may be called from multiple threads in Excel 2007 or later

- **macro** (*boolean*) – If True the function will be registered as a macro sheet equivalent function. Macro sheet equivalent functions are less restricted in what they can do, and in particular they can call Excel macro sheet functions such as `xlfCaller`.
- **allow_abort** (*boolean*) – If True the function may be cancelled by the user pressing Esc. A KeyboardInterrupt exception is raised when Esc is pressed. If not specified the behavior is determined by the *allow_abort* setting in the config (see *PyXLL Settings*).

  Enabling this option has performance implications. See *Interrupting Functions* for more details.
- **volatile** (*boolean*) – if True the function will be registered as a volatile function, which means it will be called every time Excel recalculates regardless of whether any of the parameters to the function have changed or not
- **disable_function_wizard_calc** (*boolean*) – Don't call from the Excel function wizard. This is useful for functions that take a long time to complete that would otherwise make the function wizard unresponsive

  The default value for this argument can be set in the *pyxll.cfg* file.
- **disable_replace_calc** (*boolean*) – Set to True to stop the function being called from Excel's find and replace dialog.

  The default value for this argument can be set in the *pyxll.cfg* file.
- **arg_descriptions** – dict of parameter names to help strings.
- **name** (*string*) – The Excel function name. If None, the Python function name is used.
- **auto_resize** (*boolean*) – When returining an array, PyXLL can automatically resize the range used by the formula to match the size of the result.
- **hidden** (*boolean*) – If True the UDF is hidden and will not appear in the Excel Function Wizard.

  @Since PyXLL 3.5.0
- **transpose** (*boolean*) – If true, if an array is returned it will be transposed before being returned to Excel. This can be used for returning 1d lists as rows.

  @Since PyXLL 4.2.0
- **recalc_on_open** (*boolean*) – If true, when saved and re-opened the cell calling this function will be recalculated. The default is True for functions returning cached objects and RTD functions, and False otherwise.

  See *Recalculating On Open*.

  @Since PyXLL 4.5.0
- **recalc_on_reload** (*boolean*) – If true, when the add-in is reloaded the cell calling this function will be recalculated.

  See *Recalculating On Reload*.

  @Since PyXLL 5.10.0
- **formatter** (`pyxll.Formatter`) – `Formatter` object to use to format the result of the function. For brevity a dict may be used, in which case a `Formatter` will be constructed from that dict.

  See *Cell Formatting*.

  @Since PyXLL 4.5.0
- **nan_value** – Value to use in the case that the return value is `NaN`.

  Defaults to the global setting `nan_value` set in the *config file*, or `#NUM!` if not set.

  Set to an Exception instance (e.g. `RuntimeError()`) to return an Excel error.

@Since PyXLL 5.5.0

- **posinf_value** – Value to use in the case that the return value is `+Inf`.

  Defaults to the global setting `posinf_value` set in the *config file*, or Excel's own numeric representation of `+Inf` if not set.

  Set to an Exception instance (e.g. `RuntimeError()`) to return an Excel error.

  @Since PyXLL 5.5.0

- **neginf_value** – Value to use in the case that the return value is `-Inf`.

  Defaults to the global setting `neginf_value` set in the *config file*, or Excel's own numeric representation of `-Inf` if not set.

  Set to an Exception instance (e.g. `RuntimeError()`) to return an Excel error.

  @Since PyXLL 5.5.0

- **none_value** – Value to use in the case that the return value is `None`.

  Defaults to the global setting `none_value` set in the *config file*. If not set, the Excel `NULL` value is used which gets displayed as `0` by default.

  Set to an empty string to return an empty looking cell instead of 0.

  @Since PyXLL 5.9.0

- **decimals_to_float** – If set, any returned Python `Decimal` objects will be converted automatically to a floating point number when the `var` type is used.

  Excel doesn't support decimal values and so if this is not set then a Python object handle will be returned if a `Decimal` is returned when using the `var` type.

  Defaults to the global setting `decimals_to_float` set in the config file. If not set, it is enabled by default.

  @Since PyXLL 5.11.0

- **lru_cache** – Enables the LRU cache for this function.

  See *Cached Functions* for details.

  @Since PyXLL 5.11.0

- **cache** – Enables a custom cache for this function.

  See *Alternative Caching Methods* for details.

  @Since PyXLL 5.12.0

Example usage:

```
from pyxll import xl_func

@xl_func
def hello(name):
    """return a familiar greeting"""
    return "Hello, %s" % name

# Python 3 using type annotations
@xl_func
def hello2(name: str) -> str:
    """return a familiar greeting"""
    return "Hello, %s" % name

# Or a signature may be provided as string
@xl_func("int n: int", category="Math", thread_safe=True)
```

(continues on next page)

(continued from previous page)

```
def fibonacci(n):
    """naive iterative implementation of fibonacci"""
    a, b = 0, 1
    for i in xrange(n):
        a, b = b, a + b
    return a
```

See *Worksheet Functions* for more details about using the xl_func decorator, and *Array Functions* for more details about array functions.

class XLAsyncHandle

XLAsyncHandle instances are passed to *Asynchronous Functions* as the *async_handle* argument.

They are passed to `xlAsyncReturn` to return the result from an asynchronous function.

set_value(*value*)

Set the value on the handle and return it to Excel.

Equivalent to `xlAsyncReturn`.

@Since PyXLL 4.2.0

set_error(*exc_type*, *exc_value*, *exc_traceback*)

Return an error to Excel.

@Since PyXLL 4.2.0

Example usage:

```
from pyxll import xl_func
import threading
import sys

@xl_func("async_handle h, int x")
def async_func(h, x):
    def thread_func(h, x):
        try:
            result = do_calculation(x)
            h.set_value(result)
        except:
            result.set_error(*sys.exc_info())

    thread = threading.Thread(target=thread_func, args=(h, x))
    thread.start()
```

New in PyXLL 4.2

For Python 3.5.1 and later, asynchronous UDFs can be simplified by simply using the *async* keyword on the function declaration and dropping the *async_handle* argument.

Async functions written in this way run in an asyncio event loop on a background thread.

4.2 Real Time Data

Please see *Real Time Data* in the User Guide for more details about writing RTD functions.

See also *Worksheet Functions*.

- *RTD*
- *IterRTD*
- *AsyncIterRTD*
- *RTDGenerator*
- *RTDAsyncGenerator*

class RTD

RTD is a base class that should be derived from for use by functions wishing to return real time ticking data instead of a static value.

Since PyXLL 5.12, when using Python 3.9 or later, the RTD type is a generic type and can be used in type hints as `RTD[T]`, where `T` denotes the type of the `value` property.

See *Real Time Data* for more information.

value

Current value. Setting the value notifies Excel that the value has been updated and the new value will be shown when Excel refreshes.

connect(*self*)

Called when Excel connects to this RTD instance, which occurs shortly after an Excel function has returned an RTD object.

May be overridden in the sub-class.

@Since PyXLL 4.2.0: May be an async method.

disconnect(*self*)

Called when Excel no longer needs the RTD instance. This is usually because there are no longer any cells that need it or because Excel is shutting down.

May be overridden in the sub-class.

@Since PyXLL 4.2.0: May be an async method.

detach(*self*)

Detatches the RTD instance from any Excel RTD functions.

After detaching, any subsequent calls to the Excel RTD function that created this object will result in the Python function be re-run.

See *Restarting RTD Functions* for more details.

@Since PyXLL 5.9.0

set_error(*self*, *exc_type*, *exc_value*, *exc_traceback*)

Update Excel with an error. E.g.:

```
def update(self):
    try:
        self.value = get_new_value()
    except:
        self.set_error(*sys.exc_info())
```

class IterRTD(*RTD*)

IterRTD implements the RTD base class and wraps a Python iterator.

This class creates a background thread and iterates over the iterator in that thread. Each value yielded by the iterator is sent to Excel.

In Python 3.7 and above, the background thread is run in the same context as the calling thread (see `contextvars` in the Python documentation for details about contexts and context variables).

When writing an RTD generator (*RTD Generators*) this class is what is used to wrap the generator object into an *RTD* object.

__init__(*iterator*, *auto_detach: bool = None*)

Parameters

- **iterator** – A Python iterable object that will be iterated over.
- **auto_detach** – If True, calls *RTD.detach* when then iterator is complete.

class AsyncIterRTD(*RTD*)

AsyncIterRTD implements the RTD base class and wraps a Python async iterator.

This class iterates over the async iterator in PyXLL's asyncio event loop. Each value yielded by the iterator is sent to Excel.

In Python 3.7 and above, the background thread is run in the same context as the calling thread (see `contextvars` in the Python documentation for details about contexts and context variables).

When writing an RTD async generator (*Async RTD Generators*) this class is what is used to wrap the async generator object into an *RTD* object.

__init__(*iterator*, *auto_detach: bool = None*)

Parameters

- **iterator** – A Python async iterable object that will be iterated over.
- **auto_detach** – If True, calls *RTD.detach* when then iterator is complete.

type RTDGenerator

New in PyXLL 5.12, requires Python 3.12+

RTDGenerator is a generic type alias that can be used as a type hint for RTD generators.

Using the RTDGenerator type avoids the need for a function signature to be used when registering an RTD generator with *@xl_func*.

See *Using Type Hints*.

```
from pyxll import xl_func, RTDGenerator
from typing import Any
import time

@xl_func
def my_rtd_generator(...) -> RTDGenerator[Any]:
    i = 0
    while True:
        i += 1
        yield i
        time.sleep(5)
```

type RTDAsyncGenerator

New in PyXLL 5.12, requires Python 3.12+

RTDAsyncGenerator is a generic type alias that can be used as a type hint for RTD async generators.

Using the RTDAsyncGenerator type avoids the need for a function signature to be used when registering an RTD async generator with *@xl_func*.

See *Using Type Hints*.

```
from pyxll import xl_func, RTDAsyncGenerator
from typing import Any
import asyncio

@xl_func
async def my_async_rtd_generator(...) -> AsyncRTDGenerator[Any]:
    i = 0
    while True:
        i += 1
        yield i
        await asyncio.sleep(5)
```

4.3 Macro Functions

These decorators, functions and classes are used to expose Python functions to Excel as macro functions (Subs).

Please see *Introduction* for complete details on writing Excel macro function in Python.

See also *Type Conversion*.

- *@xl_macro*
- *xl_app*
- *xl_disable*
- *XLCell*
- *XLRect*

@xl_macro(*signature: str = None, allow_abort: bool = None, name: str = None, shortcut: str = None, nan_value: Any = undefined, posinf_value: Any = undefined, neginf_value: Any = undefined, none_value: Any = undefined, decimals_to_float: bool = None, disable_calculation: bool = False, disable_screen_updating: bool = False, disable_alerts: bool = False, restore_selection: bool = False, restore_active_sheet: bool = False, lru_cache: int | bool = None, cache: Mapping = None*)

xl_macro is a decorator for exposing python functions to Excel as macros. Macros can be triggered from controls, from VBA or using COM.

Parameters

- **signature** (*str*) – An optional string that specifies the argument types and, optionally, their names and the return type.

  The format of the signature is identical to the one used by *@xl_func*.

  If no signature is provided the argument and return types will be inferred from any type annotations, and if there are no type annotations then the types will be assumed to be `var`.

- **allow_abort** (*bool*) – If True the function may be cancelled by the user pressing Esc. A KeyboardInterrupt exception is raised when Esc is pressed. If not specified the behavior is determined by the *allow_abort* setting in the config (see *PyXLL Settings*).

- **name** (*string*) – The Excel macro name. If None, the Python function name is used.

- **shortcut** (*string*) – Assigns a keyboard shortcut to the macro. Shortcuts should be one or more modifier key names (*Ctrl*, *Shift* or *Alt*) and a key, separated by the '+' symbol. For example, 'Ctrl+Shift+R'.

  If the same key combination is already in use by Excel it may not be possible to assign a macro to that combination.

Macros can also have keyboard shortcuts assigned in the config file (see *configuration*).

- **transpose** (*boolean*) – If true, if an array is returned it will be transposed before being returned to Excel.
- **nan_value** – Value to use in the case that the return value is `NaN`.

  Defaults to the global setting `nan_value` set in the *config file*, or `#NUM!` if not set.

  Set to an Exception instance (e.g. `RuntimeError()`) to return an Excel error.

  @Since PyXLL 5.5.0
- **posinf_value** – Value to use in the case that the return value is `+Inf`.

  Defaults to the global setting `posinf_value` set in the *config file*, or Excel's own numeric representation of `+Inf` if not set.

  Set to an Exception instance (e.g. `RuntimeError()`) to return an Excel error.

  @Since PyXLL 5.5.0
- **neginf_value** – Value to use in the case that the return value is `-Inf`.

  Defaults to the global setting `neginf_value` set in the *config file*, or Excel's own numeric representation of `-Inf` if not set.

  Set to an Exception instance (e.g. `RuntimeError()`) to return an Excel error.

  @Since PyXLL 5.5.0
- **none_value** – Value to use in the case that the return value is `None`.

  Defaults to the global setting `none_value` set in the *config file*. If not set, the Excel `NULL` value is used which gets displayed as `0` by default.

  @Since PyXLL 5.9.0
- **decimals_to_float** – If set, any returned Python `Decimal` objects will be converted automatically to a floating point number when the `var` type is used.

  Excel doesn't support decimal values and so if this is not set then a Python object handle will be returned if a `Decimal` is returned when using the `var` type.

  Defaults to the global setting `decimals_to_float` set in the config file. If not set, it is enabled by default.

  @Since PyXLL 5.11.0
- **automatic_calculations** – Disable automatic calculations until finished.

  @Since PyXLL 5.9.0
- **screen_updating** – Disable screen updating until finished.

  @Since PyXLL 5.9.0
- **disable_alerts** – Disable alerts until finished.

  @Since PyXLL 5.9.0
- **restore_selection** – Restore the current selection when finished.

  @Since PyXLL 5.9.0
- **restore_active_sheet** – Restore the current active sheet when finished.

  @Since PyXLL 5.9.0
- **lru_cache** – Enables the LRU cache for this function.

  See *Cached Functions* for details.

  @Since PyXLL 5.11.0

- **cache** – Enables a custom cache for this function.

  See *Alternative Caching Methods* for details.

  @Since PyXLL 5.12.0

Example usage:

```
from pyxll import xl_macro, xlcAlert

@xl_macro
def popup_messagebox():
    """pops up a message box"""
    xlcAlert("Hello")

@xl_macro
def py_strlen(s):
    """returns the length of s"""
    return len(s)
```

See ../userguide/macros for more details about using the xl_macro decorator.

xl_app(*com_package: str = None*)

Gets the Excel Application COM object and returns it as a `win32com.Dispach`, `comtypes.POINTER(IUknown)`, `pythoncom.PyIUnknown` or `xlwings.App` object, depending on which COM package is being used.

Many methods and properties Excel Application COM object will fail if called from outside of an *Excel macro context*.

xl_app must only be used from Python code called from an Excel macro[1], menu[1], worksheet function[1][2], or Excel Application event handler.

To use it from any other context, or from a background thread, schedule a call using *schedule_call*.

Parameters

com_package (*string*) – The Python package to use when returning the COM object. It should be None, 'win32com', 'comtypes', 'pythoncom' or 'xlwings'. If None the com package set in the configuration file will be used, or 'win32com' if nothing is set.

Returns

The Excel Application COM object using the requested COM package.

> **Warning**
>
> Excel COM objects should **never** be passed between threads. Only use a COM object in the same thread it was created in. Doing otherwise is likely to crash Excel! If you are using a background thread the safest thing to do is to only call into Excel using COM via functions scheduled using *schedule_call*.

xl_disable(*automatic_calculations: bool = True, screen_updating: bool = True, alerts: bool = True, restore_selection: bool = True, restore_active_sheet: bool = True*)

New in PyXLL 5.9

Context manager to disable Excel while performing operations that interact with Excel.

Can only be used from inside an Excel macro, or from a function scheduled using pyxll.schedule_call.

Example:

[1] Do not use async functions when using *xl_app* as async functions run in the asyncio event loop on a background thread. If you need to use *xl_app* from an async function, schedule it using *schedule_call*.

[2] Certain things will not work when trying to call back into Excel with COM from an Excel worksheet function as some operations are not allowed while Excel is calculating. For example, trying to set the value of a cell will fail. For these cases, use *schedule_call* to schedule a call *after* Excel has finished calculating.

```
@xl_macro
def macro_function():
    with xl_disable():
        # do some work here that updates Excel where we do not
        # want Excel to automatically recalculate or update.
        xl = xl_app()
        xl.Range("A1").Value = 1

    # After the with block, Excel reverts to its previous calculation mode.
    return
```

Parameters

- **automatic_calculations** – Disable automatic calculations until finished.
- **screen_updating** – Disable screen updating until finished.
- **alerts** – Disable alerts until finished.
- **restore_selection** – Restore the current selection when finished.
- **restore_active_sheet** – Restore the current active sheet when finished.

class XLCell

XLCell represents the data and metadata for a cell (or range of cells) in Excel.

XLCell instances are passed as an `xl_cell` argument to a function registered with *@xl_func*, or may be constructed using *from_range*.

Some of the properties of *XLCell* instances can only be accessed if the calling function has been registered as a macro sheet equivalent function[3].

Example usage:

```
from pyxll import xl_func

@xl_func("xl_cell cell: string", macro=True)
def xl_cell_test(cell):
    return "[value=%s, address=%s, formula=%s, note=%s]" % (
                cell.value,
                cell.address,
                cell.formula,
                cell.note)
```

from_range(*range*)

Static method to construct an XLCell from an Excel *Range* instance or address (e.g. 'A1').

The *Range* class is part of the *Excel Object Model*, and can be obtained via *xl_app*.

See *Python as a VBA Replacement*.

The XLCell instance returned can be used to get and set values in Excel using PyXLL type converters and object cache.

Parameters

range – An Excel *Range* object or cell address as a string.

Example usage:

[3] A macro sheet equivalent function is a function exposed using *@xl_func* with *macro=True*.

```
xl = xl_app()
range = xl.Selection
cell = XLCell.from_range(range)
cell.options(type='object').value = x
```

Must be called from a macro or macro sheet equivalent function[Page 236, 3]

value

Get or set the value of the cell.

The type conversion when getting or setting the cell content is determined by the type passed to `XLCell.options`. If no type is specified then the type conversion will be done using the `var` type.

Must be called from a macro or macro sheet equivalent function[Page 236, 3]

address

String representing the address of the cell, or `None` if a value was passed to the function and not a cell reference.

Must be called from a macro or macro sheet equivalent function[Page 236, 3]

formula

Formula of the cell as a `string`, or `None` if a value was passed to the function and not a cell reference or if the cell has no formula.

Must be called from a macro or macro sheet equivalent function[Page 236, 3]

note

Note on the cell as a `string`, or `None` if a value was passed to the function and not a cell reference or if the cell has no note.

Must be called from a macro or macro sheet equivalent function[Page 236, 3]

sheet_name

Name of the sheet this cell belongs to.

sheet_id

Integer id of the sheet this cell belongs to.

rect

`XLRect` instance with the coordinates of the cell.

is_calculated

True or False indicating whether the cell has been calculated or not. In almost all cases this will always be True as Excel will automatically have recalculated the cell before passing it to the function.

options(*self*, *type: Any = 'var'*, *auto_resize: bool = False*, *type_kwargs: dict = {}*, *nan_value: Any = undefined*, *posinf_value: Any = undefined*, *neginf_value: Any = undefined*, *decimals_to_float: bool = None*, *check_equals: bool = False*)

Sets the options on the XLCell instance.

Parameters

- **type** – Data type to use when converting values to or from Excel. The default type is `var`, but any recognized types may be used, including `object` for getting or setting cached objects.
- **auto_resize** – When setting the cell value in Excel, if auto_resize is set and the value is an array, the cell will be expanded automatically to fit the size of the Python array.

  When getting `XLCell.value`, if auto_resize is set then the returned value will also include adjacent cells according to these rules:

  – If the cell references a table then the entire table's contents be use used.

- Otherwise, if the cell is part of a block of non-blank cells, the values for the entire block are used.

Note: auto_resize was extended to work when getting values in PyXLL 5.8.0. In earlier versions it has no effect when getting the cell value.

- **type_kwargs** – If setting `type`, type_kwargs can also be set as the options for that type.
- **nan_value** – Value to use in the case that the value being set is `NaN`.

  Defaults to the global setting `nan_value` set in the *config file*, or `#NUM!` if not set.

  Set to an Exception instance (e.g. `RuntimeError()`) to return an Excel error.

  @Since PyXLL 5.5.0
- **posinf_value** – Value to use in the case that the value being set is `+Inf`.

  Defaults to the global setting `posinf_value` set in the *config file*, or Excel's own numeric representation of `+Inf` if not set.

  Set to an Exception instance (e.g. `RuntimeError()`) to return an Excel error.

  @Since PyXLL 5.5.0
- **neginf_value** – Value to use in the case that the value being set is `-Inf`.

  Defaults to the global setting `neginf_value` set in the *config file*, or Excel's own numeric representation of `-Inf` if not set.

  Set to an Exception instance (e.g. `RuntimeError()`) to return an Excel error.

  @Since PyXLL 5.5.0
- **decimals_to_float** – If set, Python `Decimal` objects will be converted automatically to a floating point number when the `var` type is used.

  Excel doesn't support decimal values and so if this is not set then a Python object handle will be used if a `Decimal` value is set when using the `var` type.

  Defaults to the global setting `decimals_to_float` set in the config file. If not set, it is enabled by default.

  @Since PyXLL 5.11.0
- **check_equals** – If set to `True` then the current Excel value will be compared with the value being set. If the values are already equal then the value in Excel will not be updated.

  This can be used to avoid circular updates where updating a cell triggers function, which updates the cell, triggering the function to be called again and so on.

  This option can be used when setting table values as well as plain cell values.

  Note that when setting objects as object handles, the object handle changes each time even if the object is the same and so the new object handle will be written, even if the underlying object has not changed.

  @Since PyXLL 5.12.0

Returns

self. The cell options are modified and the same instance is returned, for easier method chaining.

Example usage:

```
cell.options(type='dataframe', auto_resize=True).value = df
```

to_range(*self*, *com_wrapper: str = None*)

Return an Excel *Range* COM object using the COM package specified.

Parameters

com_package – COM package to use to return the COM *Range* object.

com_package may be any of:

- win32com (default)
- comtypes
- xlwings

@Since PyXLL 4.4.0

offset(*self*, *rows: int = 0*, *columns: int = 0*)

Return a clone of the *XLCell*, offset by *rows* and *columns*.

Parameters

- **rows** (*int*) – Number of rows to offset by.
- **columns** (*int*) – Number of columns to offset by.

@Since PyXLL 5.8.0

resize(*self*, *rows: int = None*, *columns: int = None*)

Return a clone of the *XLCell*, resized to *rows* and *columns*.

Parameters

- **rows** (*int*) – Number of rows after resizing (or kept the same if not specified).
- **columns** (*int*) – Number of columns after resizing (or kept the same if not specified).

@Since PyXLL 5.8.0

class XLRect

XLRect instances are accessed via *XLCell.rect* to get the coordinates of the cell.

first_row

First row of the range as an integer.

last_row

Last row of the range as an integer.

first_col

First column of the range as an integer.

last_col

Last column of the range as an integer.

4.4 Type Conversion

The following functions and decorators provide access to PyXLL's type conversion system.

When registering Python functions as Excel worksheet functions using *@xl_func*, or as Excel macros using *@xl_macro*, type information can be supplied that informs the PyXLL add-in how to convert between Python and Excel values.

Type information can be provided to PyXLL in a number of different ways, include Python type hints, a function signature string, and using the *@xl_arg* and *@xl_return* decorators documented below.

Custom type conversion methods can be registered using the *@xl_arg_type* and *@xl_return_type* decorators.

To access PyXLL's type converters (including any custom type converters you have registered) you can use the *get_type_converter* function.

See *Argument and Return Types* for more details about PyXLL's type conversion features.

- *get_type_converter*
- *@xl_arg*
- *@xl_return*
- *@xl_arg_type*
- *@xl_return_type*
- *TypeParameters*
- *Object*

get_type_converter(*src_type, dest_type [, src_kwargs=None] [, dest_kwargs=None]*)

Returns a function to convert objects of type src_type to dest_type.

Even if there is no function registered that converts exactly from `src_type` to `dest_type`, as long as there is a way to convert from `src_type` to `dest_type` using one or more intermediate types this function will create a function to do that.

Parameters

- **src_type** – Type or signature of type to convert from.
- **dest_type** – Type or signature of type to convert to.
- **src_kwargs** (*dict*) – Parameters for the source type (e.g. `{'dtype'=float}` for `numpy_array`).
- **dest_kwargs** (*dict*) – Parameters for the destination type (e.g. `{'index'=True}` for `dataframe`).

Returns

Function to convert from src_type to dest_type.

Example usage:

```
from pyxll import xl_func, get_type_converter

@xl_func("var x: var")
def py_function(x):
    # if x is a number, convert it to a date
    if isinstance(x, float):
        to_date = get_type_converter("var", "date")
        x = to_date(x)
    return "%s : %s" % (x, type(x))
```

@xl_arg(*_name*[, *_type*][, *_label*][, *_description*][, ***kwargs*])

Decorator for providing type information for a function argument. This can be used instead of providing a function signature to *@xl_func* and *@xl_macro*.

Parameters

- **_name** (*string*) – Argument name. This should match the argument name in the function definition.
- **_type** – Optional argument type. This should be a recognized type name or the name of a custom type.
- **_label** – Argument label that will be used in the Excel function wizard.

  Defaults to the argument name.

@Since PyXLL 5.5.0.

- **_description** – Argument description that will be used in the Excel function wizard.

  This can be used instead of documenting the parameter in the function's docstring.

  @Since PyXLL 5.5.0.

- **kwargs** – Type parameters for parameterized types (eg *NumPy arrays* and *Pandas types*).

`@xl_return`(*[_type=None] [, **kwargs]*)

Decorator for providing type information for a function's return value. This can be used instead of providing a function signature to `@xl_func` and `@xl_macro`.

Parameters

- **_type** – Optional argument type. This should be a recognized type name or the name of a custom type.
- **kwargs** – Type parameters for parameterized types (eg *NumPy arrays* and *Pandas types*).

`@xl_arg_type`(*name, base_type [, allow_arrays=True] [, macro=None] [, thread_safe=None] [, typing_type=None]*)

Returns a decorator for registering a function for converting from a base type to a custom type.

Parameters

- **name** (*string*) – custom type name.
- **base_type** (*string*) – base type.
- **allow_arrays** (*boolean*) – custom type may be passed in an array using the standard `[]` notation.
- **macro** (*boolean*) – If `True` all functions using this type will automatically be registered as a macro sheet equivalent function.
- **thread_safe** (*boolean*) – If `False` any function using this type will never be registered as thread safe.
- **typing_type** – An optional `typing` type that will be recognised as this type when used as a Python type hint.

`@xl_return_type`(*name, base_type [, allow_arrays=True] [, macro=None] [, thread_safe=None] [, typing_type=None]*)

Returns a decorator for registering a function for converting from a custom type to a base type.

Parameters

- **name** (*string*) – custom type name.
- **base_type** (*string*) – base type.
- **allow_arrays** (*boolean*) – custom type may be returned as an array using the standard `[]` notation.
- **macro** (*boolean*) – If `True` all functions using this type will automatically be registered as a macro sheet equivalent function.
- **thread_safe** (*boolean*) – If `False` any function using this type will never be registered as thread safe.
- **typing_type** – An optional `typing` type that will be recognised as this type when used as a Python type hint.

`class TypeParameters`

New in PyXLL 5.12

The `TypeParametes` class is used in conjunction with `typing.Annotated` to specify type parameters used by PyXLL when converting values between Python and Excel.

Using the `TypeParameters` class it is possible to specify options to PyXLL's type converters using only Python type hints, without having to use additional type specifier strings.

For example, to specify an argument should be treated as a `pandas.DataFrame` with the first column being the index, instead of using the type specifier string `dataframe<index=True>` we can use the Python type hint `Annotated[pd.DataFrame, TP(index=True)]` like this:

```
from pyxll import xl_func
from pyxll import TypeParameters as TP
from typing import Annotated
import pandas as pd

@xl_func
def df_head(
    df: Annotated[pd.DataFrame, TP(index=True)],
    n: int
) -> Annotated[pd.DataFrame, TP(index=True)]:
    return df.head(n)
```

type Object

New in PyXLL 5.12, requires Python 3.12+

`Object` is a generic type alias that can be used where a parameter or returned value should be passed as a cached object using an object handle, instead of directly by value.

The `Object[T]` type resolves to `T` when type checking, allowing function that use cached objects to be written with correct typing information.

See *Using Type Hints*.

```
from pyxll import xl_func, Object

class CustomObject:
    def __init__(self, name):
        self.name = name

# The 'Object[T]' generic type alias is used as we want PyXLL to return
# the object to Excel as a cached object, but we want type checks to see
# 'CustomObject' as the return type.
@xl_func
def create_object(name: str) -> Object[CustomObject]:
    return CustomObject(name)
```

4.5 Ribbon Functions

These functions can be used to manipulate the Excel ribbon.

The ribbon can be updated at any time, for example as PyXLL is loading via the *xl_on_open* and *xl_on_reload* event handlers, or from a menu using using *@xl_menu*.

See the section on *customizing the ribbon* for more details.

- *load_image*
- *get_ribbon_xml*
- *set_ribbon_xml*
- *set_ribbon_tab*

- *remove_ribbon_tab*
- *IRibbonControl*
- *IRibbonUI*

load_image(*name: str*)

Loads an image file and returns it as a COM *IPicture* object suitable for use when *customizing the ribbon*.

This function can be set at the Ribbon image handler by setting the *loadImage* attribute on the *customUI* element in the ribbon XML file.

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
            loadImage="pyxll.load_image">
    <ribbon>
        <tabs>
            <tab id="CustomTab" label="Custom Tab">
                <group id="Tools" label="Tools">
                    <button id="Reload"
                            size="large"
                            label="Reload PyXLL"
                            onAction="pyxll.reload"
                            image="reload.png"/>
                </group>
            </tab>
        </tabs>
    </ribbon>
</customUI>
```

Or it can be used when returning an image from a *getImage* callback.

Parameters

name (*string*) – Filename or resource location of the image file to load. This may be an absolute path or a resource location in the form `module:resource`.

Returns

A COM *IPicture* object (the exact type depends on the *com_package* setting in the *config*.

get_ribbon_xml()

Returns the XML used to customize the Excel ribbon bar, as a string.

See the section on *customizing the ribbon* for more details.

set_ribbon_xml(*xml*, *reload: bool = True*)

Sets the XML used to customize the Excel ribbon bar.

Parameters

- **xml** – XML to set, as a string.
- **reload** – If True, the ribbon bar will be reloaded using the new XML (does *not* reload PyXLL).

See the section on *customizing the ribbon* for more details.

set_ribbon_tab(*xml: str*, *tab_id: str = None*, *reload: bool = True*)

Sets a single tab in the ribbon using an XML fragment.

Instead of replacing the whole ribbon XML this function takes a tab element from the input XML and updates the ribbon XML with that tab.

If multiple tabs exist in the input XML, the first who's *id* attribute matches *tab_id* is used (or simply the first tab element if *tab_id* is None).

If a tab already exists in the ribbon XML with the same *id* attribute then it is replaced, otherwise the new tab is appended to the tabs element.

Parameters

- **xml** – XML document containing at least one *tab* element.
- **tab_id** – *id* of the tab element to set (or None to use the first tab element in the document).
- **reload** – If True, the ribbon bar will be reloaded using the new XML (does *not* reload PyXLL).

remove_ribbon_tab(*tab_id: str*, *reload: bool = True*)

Removes a single tab from the ribbon XML where the tab element's *id* attribute matches *tab_id*.

Parameters

- **tab_id** – id of the tab element to remove.
- **reload** – If True, the ribbon bar will be reloaded using the new XML (does *not* reload PyXLL).

Returns

True if a tab was removed, False otherwise.

class IRibbonControl

Many ribbon callbacks will be passed a control object as a *IRibbonControl* object.

The control object corresponds to the ribbon control item handling the action.

This class is a Python wrapper around the Excel COM type of the same name.

Id

Gets the ID of the control specified in the Ribbon XML markup customization file. Read-only.

Tag

Used to store arbitrary strings and fetch them at runtime. Read-only.

Context

Represents the active window containing the Ribbon user interface that triggers a callback procedure. Read-only.

class IRibbonUI

Certain ribbon callbacks such as `onLoad` will be passed an *IRibbonUI* object.

This class is a Python wrapper around the Excel COM type of the same name.

Invalidate(*self*)

Invalidates the cached values for all of the controls of the Ribbon user interface.

InvalidateControl(*self*, *control_id: str*)

Invalidates the cached value for a single control on the Ribbon user interface.

Parameters

control_id – Id of the control to invalidate.

InvalidateControlMso(*self*, *control_id: str*)

Invalidating a control, repaints the screen and causes any callback procedures associated with that control to execute.

Parameters

control_id – Mso id of the control to invalidate.

ActivateTab(*self*, *tab_id: str*)

Activates the specified custom tab.

Parameters
tab_id – Id of the tab to activate.

ActivateTabMso(*self*, *tab_id: str*)

Activates the specified custom tab.

Parameters
tab_id – Mso id of the tab to activate.

ActivateTabQ(*self*, *tab_id: str*)

Activates the specified custom tab.

Parameters
tab_id – Fully qualified id of the tab to activate including the namespace.

4.6 Menu Functions

These decorators are used to expose Python functions to Excel as menu items.

This is using the 'old style' Add-Ins menu in Excel. For ribbon toolbars, please see *Customizing the Ribbon*.

@xl_menu(*name*, *menu=None*, *sub_menu=None*, *order=0*, *menu_order=0*, *allow_abort=None*, *shortcut=None*)

xl_menu is a decorator for creating menu items that call Python functions. Menus appear in the 'Addins' section of the Excel ribbon from Excel 2007 onwards, or as a new menu in the main menu bar in earlier Excel versions.

Parameters

- **name** (*string*) – name of the menu item that the user will see in the menu
- **menu** (*string*) – name of the menu that the item will be added to. If a menu of that name doesn't already exist it will be created. By default the PyXLL menu is used
- **sub_menu** (*string*) – name of the submenu that this item belongs to. If a submenu of that name doesn't exist it will be created
- **order** (*int*) – influences where the item appears in the menu. The higher the number, the further down the list. Items with the same sort order are ordered lexographically. If the item is a sub-menu item, this order influences where the sub-menu will appear in the main menu. The menu order my also be set in the config (see *configuration*).
- **sub_order** (*int*) – similar to order but it is used to set the order of items within a sub-menu
- **menu_order** (*int*) – used when there are multiple menus and controls the order in which the menus are added
- **allow_abort** (*boolean*) – If True the function may be cancelled by the user pressing Esc. A KeyboardInterrupt exception is raised when Esc is pressed. If not specified the behavior is determined by the *allow_abort* setting in the config (see *PyXLL Settings*).
- **shortcut** (*string*) – Assigns a keyboard shortcut to the menu item. Shortcuts should be one or more modifier key names (*Ctrl*, *Shift* or *Alt*) and a key, separated by the '+' symbol. For example, 'Ctrl+Shift+R'.

  If the same key combination is already in use by Excel it may not be possible to assign a menu item to that combination.

Example usage:

```
from pyxll import xl_menu, xlcAlert

@xl_menu("My menu item")
def my_menu_item():
    xlcAlert("Menu button example")
```

See *Menu Functions* for more details about using the xl_menu decorator.

4.7 Plotting

See *Charts and Plotting* for more information about plotting Python charts in Excel.

- `plot`
- `PlotBridgeBase`

plot(*figure=None*, *name=None*, *width=None*, *height=None*, *top=None*, *left=None*, *sheet=None*, *allow_html=None*, *allow_svg=None*, *allow_resize=None*, *reset=False*, *bridge_cls=None*, ***kwargs*)

Plots a figure to Excel as an embedded image.

This can be called from an Excel worksheet function, or from anywhere else such as a macro or menu function. If called from a worksheet function the image will be placed below the calling cell, and repeated calls will update the image rather than create new ones. If called from elsewhere the image will be placed below the current selection, and each call will create a new image.

The figure can be any of the following:

- A matplotlib Figure, Subplot, Artist or Animation object
- A plotly Figure object
- A bokeh Plot object
- An altair Chart object
- A plotnine ggplot object (*New in PyXLL 5.13*)

If no figure is provided the current matplotlib.pyplot figure is used.

Parameters

- **figure** – Figure to plot. This can be an instance of any of the following:
  - matplotlib.figure.Figure
  - matplotlib.artist.Artist
  - matplotlib.axes._subplots.SubplotBase
  - matplotlib.animation.Animation
  - plotly.graph_objects.Figure
  - bokeh.models.plots.Plot
  - altair.vegalite.v4.api.Chart
  - plotnine.ggplot.ggplot (*New in PyXLL 5.13*)

  If None, the active matplotlib.pyplot figure is used.
- **name** – Name of Picture object in Excel. If this is None then a name will be chosen, and if called from a UDF then repeated calls will re-use the same name.
- **width** – Initial width of the picture in Excel, in points. If set then height must also be set. If None the width will be taken from the figure.
- **height** – Initial height of the picture in Excel, in points. If set then width must also be set. If None the height will be taken from the figure.
- **top** – Initial location of the top of the plot in Excel, in points. If set then left must also be set. If None, the picture will be placed below the current or selected cell.

- **left** – Initial location of the left of the plot in Excel, in points. If set then top must also be set. If None, the picture will be placed below the current or selected cell.
- **sheet** – Name of the sheet to add the picture to. If none, the current sheet is used.
- **allow_html** – Some figures may be rendered as HTML and displayed using a web control, if the plotting library and the version of Excel being used allows.

  This can be disabled by setting this option to False.

  The default value may be changed by setting `plot_allow_html` in the `[PYXLL]` section of the config file.
- **allow_svg** – Some figures may be rendered as SVG, if the plotting library and the version of Excel being used allows.

  This can be disabled by setting this option to False.

  The default value may be changed by setting `plot_allow_svg` in the `[PYXLL]` section of the config file.
- **allow_resize** – If enabled, the figure will be re-drawn after the image is resized in Excel, and when the selection changes.

  This is enabled by default and can be disabled by setting this option to False.

  The default value may be changed by setting `plot_allow_svg` in the `[PYXLL]` section of the config file.

  New in PyXLL 5.7
- **reset** – Reset the image size and position to the values specified. If `False` (default) the arguments `width`, `height`, `top` and `left` only affect the initial size and position of the image, allowing the user to resize and reposition it without it being reset each time it is updated.

  @Since PyXLL 5.4.0
- **bridge_cls** – Class to use for exporting the plot as an image. If None this will be selected automatically based on the type of figure.
- **kwargs** – Additional arguments will be called to the implementation specific method for exporting the figure to an image.

> **Note**
>
> The options width, height, top and left only affect the image when it is initially created. If a subsequent call to plot updates an existing image (for example, if called from a worksheet function or if a name is passed) then it will not be resized or moved from its current location.

class PlotBridgeBase

Base class for plotting bridges used by *plot*.

This can be used to add support for plotting libraries other than the standard ones supported by PyXLL.

All methods must be implemented by the derived class.

__init__(*self*, *figure*)

Construct the plot bridge for exporting a figure. The figure is the object passed to *plot*.

can_export(*self*, *format*)

Return True if the figure can be exported in a specific format.

Valid formats are 'html', 'svg', 'png' and 'gif'.

get_size_hint(*self*, *dpi*)

Return (width, height) tuple the figure should be exported as or None.

Width and height are in points (72th of an inch).

If no size hint is available return None.

export(*self*, *width*, *height*, *dpi*, *format*, *filename*, ***kwargs*)

Export the figure to a file as a given size and format.

Parameters

- **width** – Width of the image to export in points.
- **height** – Height of the image to export in points.
- **dpi** – DPI to use to export the image.
- **format** – Format to export the image to. Valid formats are 'html', 'svg', 'png' and 'gif'.
- **filename** – Filename to export the image to.
- **kwargs** – Additional kwargs passed to `plot`.

4.8 Custom Task Panes

See *Custom Task Panes* for more information about Custom Task Panels in PyXLL.

- `create_ctp`
- `CustomTaskPane`
- `CTPBridgeBase`

create_ctp(*control*, *title=None*, *width=None*, *height=None*, *position=CTPDockPositionRight*, *position_restrict=CTPDockPositionRestrictNone*, *top=None*, *left=None*, *timer_interval=0.1*, *bridge_cls=None*)

Creates a Custom Task Pane from a UI control object.

The control object can be any of the following:

- tkinter.Toplevel
- PySide2.QtWidgets.QWidget
- PySide6.QtWidgets.QWidget
- PyQt5.QtWidgets.QWidget
- PyQt6.QtWidgets.QWidget
- wx.Frame

Parameters

- **control** – UI control of one of the supported types.
- **title** – Title of the custom task pane to be created.
- **width** – Initial width of the custom task pane in points.
- **height** – Initial height of the custom task pane in points.
- **position** – Where to display the custom task pane. Can be any of:
  - `CTPDockPositionLeft`

- `CTPDockPositionTop`
- `CTPDockPositionRight`
- `CTPDockPositionBottom`
- `CTPDockPositionFloating`

- **position_restrict** – Restrict how the user can reposition the custom task pane. Can be any of:
  - `CTPDockPositionRestrictNone`
  - `CTPDockPositionRestrictNoChange`
  - `CTPDockPositionRestrictNoHorizontal`
  - `CTPDockPositionRestrictNoVertical`

  New in PyXLL 5.5
- **top** – Initial top position of custom task pane (only used if floating). *New in PyXLL 5.2*
- **left** – Initial left position of custom task pane (only used if floating). *New in PyXLL 5.2*
- **timer_interval** – Time in seconds between calls to `CTPBridgeBase.on_timer`.

  The CTP bridge classes are what integrate the Python UI toolkit with the Excel Windows message loop. They use on_timer to poll their own message queues. If you are finding the panel is not responsive enough you can reduce the timer interval with this setting.

  This can also be defaulted by setting `ctp_timer_interval` in the `PYXLL` section of the *pyxll.cfg* config file.

  New in PyXLL 5.1
- **bridge_cls** – Class to use for integrating the control into Excel. If None this will be selected automatically based on the type of control.

Returns
`CustomTaskPane` (*New in PyXLL 5.5*)

class CustomTaskPane

Wrapper around the Excel COM `_CustomTaskPane` type.

Returned by `create_ctp`.

New in PyXLL 5.5

Title

Gets the title of a CustomTaskPane object. Read-only.

Application

Gets the Application object of the host application. Read-only.

Window

Gets the parent window object of the _CustomTaskPane object. Read-only.

Visible

True if the specified _CustomTaskPane object is visible. Read-only.

ContentControl

Gets the Microsoft ActiveX® control instance displayed in the custom task pane frame. Read-only.

Height

Gets or sets the height of the CustomTaskPane object (in points). Read/write.

Width

Gets or sets the width of the task pane specified by the CustomTaskPane object. Read/write.

DockPosition

Gets or sets a value specifying the docked position of a _CustomTaskPane object. Read/write.

Permitted values are:

- `CTPDockPositionLeft`
- `CTPDockPositionTop`
- `CTPDockPositionRight`
- `CTPDockPositionBottom`
- `CTPDockPositionFloating`

DockPositionRestrict

Gets or sets a value specifying a restriction on the orientation of a _CustomTaskPane object. Read/write.

Permitted values are:

- `CTPDockPositionRestrictNone`
- `CTPDockPositionRestrictNoChange`
- `CTPDockPositionRestrictNoHorizontal`
- `CTPDockPositionRestrictNoVertical`

Delete()

Deletes the custom task pane.

class CTPBridgeBase

Base class of bridges between the Python UI toolkits and PyXLL's Custom Task Panes.

This can be used to add support for UI toolkits other than the standard ones supported by PyXLL.

__init__(*self*, *control*)

Construct the custom task pane bridge. The control is the object passed to `create_ctp`.

close(*self*)

Close the host CTP window.

Do not override

get_hwnd(*self*)

Return the window handle as an integer.

Required: Override in subclass

get_title(*self*)

Return the window title as a string.

Optional: Can be overridden in subclass

pre_attach(*self*, *hwnd*)

Called before the window is attached to the Custom Task Pane host window.

Optional: Can be overridden in subclass

post_attach(*self*, *hwnd*)

Called after the window is attached to the Custom Task Pane host window.

Optional: Can be overridden in subclass

on_close(*self*)

Called when the Custom Task Pane host window is closed.

Optional: Can be overridden in subclass

on_window_closed(*self*)

Called when control window received a WM_CLOSE Windows message.

Optional: Can be overridden in subclass

on_window_destroyed(*self*)

Called when control window received a WM_DESTROY Windows message.

Optional: Can be overridden in subclass

process_message(*self*, *hwnd*, *msg*, *wparam*, *lparam*)

Called when the Custom Task Panel host window received a Windows message.

Should return a tuple of (result, handled) where the result is an integer and handled is a bool indicating whether the message has been handled or not. Messages that have been handled will not be passed to the default message handler.

Returning None is equivalent to returning (0, False).

Parameters

- **hwnd** – Window handle
- **msg** – Windows message id
- **wparam** – wparam passed with the message
- **lparam** – lparam passed with the message

Optional: Can be overridden in subclass

translate_accelerator(*self*, *hwnd*, *msg*, *wparam*, *lparam*, *modifier*)

Called when the Custom Task Panel host control's TranslateAccelerator Windows method is called.

This can be used to convert key presses into commands or events to pass to the UI toolkit control.

Should return a tuple of (result, handled) where the result is an integer and handled is a bool indicating whether the message has been handled or not. Messages that have been handled will not be passed to the default message handler.

Returning None is equivalent to returning (0, False).

Parameters

- **hwnd** – Window handle
- **msg** – Windows message id
- **wparam** – wparam passed with the message
- **lparam** – lparam passed with the message
- **modifier** – If the message is a WM_KEYDOWN message, the modifier will be set to indicate any key modifiers currently pressed. 0x0 = no key modifiers, 0x1 = Shift key pressed, 0x2 = Control key pressed, 0x4 = Alt key pressed.

Optional: Can be overridden in subclass

on_timer(*self*)

If this method is overridden then it will be called periodically and can be used to poll the UI toolkit's message loop.

The interval between calls can be set by passing `timer_interval` to *create_ctp* or by setting `ctp_timer_interval` in the `PYXLL` section of the *pyxll.cfg* config file.

Optional: Can be overridden in subclass

`CTPDockPositionLeft = 0`

`CTPDockPositionTop = 1`

`CTPDockPositionRight = 2`

`CTPDockPositionBottom = 3`

`CTPDockPositionFloating = 4`

`CTPDockPositionRestrictNone = 0`

`CTPDockPositionRestrictNoChange = 1`

`CTPDockPositionRestrictNoHorizontal = 2`

`CTPDockPositionRestrictNoVertical = 3`

4.9 ActiveX Controls

New in PyXLL 5.9

See *ActiveX Controls* for more information about ActiveX controls in PyXLL.

- *create_activex_control*
- *ActiveXHost*
- *AtxBridgeBase*
- *Creating an ActiveX Control from a Worksheet Function*

create_activex_control(*control*, *name=None*, *sheet=None*, *width=None*, *height=None*, *top=None*, *left=None*, *timer_interval=0.1*, *bridge_cls=None*)

Creates an ActiveX control a Python UI control object.

The control object can be any of the following:

- tkinter.Toplevel
- PySide2.QtWidgets.QWidget
- PySide6.QtWidgets.QWidget
- PyQt5.QtWidgets.QWidget
- PyQt6.QtWidgets.QWidget
- wx.Frame

Parameters

- **control** – UI control of one of the supported types.
- **name** – Name of the ActiveX control to be created.
- **width** – Initial width of the control in points.
- **height** – Initial height of the control in points.
- **top** – Initial top position of the control in points.
- **left** – Initial left position of the control in points.

- **timer_interval** – Time in seconds between calls to *AtxBridgeBase.on_timer*.

  The ActiveX bridge classes are what integrate the Python UI toolkit with the Excel Windows message loop. They use on_timer to poll their own message queues. If you are finding the panel is not responsive enough you can reduce the timer interval with this setting.

  This can also be defaulted by setting `activex_timer_interval` in the `PYXLL` section of the *pyxll.cfg* config file.

- **bridge_cls** – Class to use for integrating the control into Excel. If None this will be selected automatically based on the type of control.

Returns
ActiveXHost

class ActiveXHost

Returned by *create_activex_control*.

Invalidate()

Reqeusts that Excel redraws the control.

class AtxBridgeBase

Base class of bridges between the Python UI toolkits and PyXLL's ActiveX control.

This can be used to add support for UI toolkits other than the standard ones supported by PyXLL.

__init__(*self*, *control*)

Construct the ActiveX bridge. The control is the object passed to *create_activex_control*.

close(*self*)

Close the host ActiveX window.

Do not override

get_hwnd(*self*)

Return the window handle as an integer.

Required: Override in subclass

width(*self*, *dpi*)

Return the width of the control, in points.

Required: Override in subclass

height(*self*, *dpi*)

Return the height of the control, in points.

Required: Override in subclass

pre_attach(*self*, *hwnd*)

Called before the window is attached to the ActiveX control host window.

Optional: Can be overridden in subclass

post_attach(*self*, *hwnd*)

Called after the window is attached to the ActiveX control host window.

Optional: Can be overridden in subclass

on_close(*self*)

Called when the ActiveX control host window is closed.

Optional: Can be overridden in subclass

on_window_closed(*self*)

Called when control window received a WM_CLOSE Windows message.

Optional: Can be overridden in subclass

on_window_destroyed(*self*)

Called when control window received a WM_DESTROY Windows message.

Optional: Can be overridden in subclass

filter_message(*self*, *hwnd*, *msg*, *wparam*, *lparam*)

Called when the ActiveX control host window received a Windows message.

Should return `True` if the message should be filtered and not passed to the window's message handler, or `False` otherwise.

Parameters

- **hwnd** – Window handle
- **msg** – Windows message id
- **wparam** – wparam passed with the message
- **lparam** – lparam passed with the message

Optional: Can be overridden in subclass

process_message(*self*, *hwnd*, *msg*, *wparam*, *lparam*)

Called when the ActiveX control host window received a Windows message.

Should return `True` if the message was handled and shouldn't be processed further, or `False` otherwise.

Parameters

- **hwnd** – Window handle
- **msg** – Windows message id
- **wparam** – wparam passed with the message
- **lparam** – lparam passed with the message

Optional: Can be overridden in subclass

translate_accelerator(*self*, *hwnd*, *msg*, *wparam*, *lparam*, *modifier*)

Called when the ActiveX control host control's TranslateAccelerator Windows method is called.

This can be used to convert key presses into commands or events to pass to the UI toolkit control.

Should return `True` if the message was handled and shouldn't be processed further, or `False` otherwise.

Returning None is equivalent to returning (0, False).

Parameters

- **hwnd** – Window handle
- **msg** – Windows message id
- **wparam** – wparam passed with the message
- **lparam** – lparam passed with the message
- **modifier** – If the message is a WM_KEYDOWN message, the modifier will be set to indicate any key modifiers currently pressed. 0x0 = no key modifiers, 0x1 = Shift key pressed, 0x2 = Control key pressed, 0x4 = Alt key pressed.

Optional: Can be overridden in subclass

on_timer(*self*)

If this method is overridden then it will be called periodically and can be used to poll the UI toolkit's message loop.

The interval between calls can be set by passing `timer_interval` to *create_activex_control* or by setting `activex_timer_interval` in the `PYXLL` section of the `pyxll.cfg` config file.

Optional: Can be overridden in subclass

4.9.1 Creating an ActiveX Control from a Worksheet Function

Sometimes you will want to create an ActiveX control when a worksheet function is called, instead of when a ribbon button or macro function is called.

For example, if you want an ActiveX control to be created each time a workbook is opened one easy way to do that is to use a worksheet function using the `recalc_on_open` argument to the *@xl_func* decorator. That way, whenever the workbook is opened the function will be called and can create the control.

However, in order to create the ActiveX control when the function is called you need to also use *schedule_call*. You can't make changes to the worksheet while Excel is calculating and so instead you must use *schedule_call* to schedule another function to be called after Excel has finished calculating.

Additionally, you should pass `name` to *create_activex_control* so that when the control is created, it re-uses any existing control with the same name rather than creating a new control every time it is called.

For example:

```
from pyxll import xl_func, schedule_call

@xl_func(recalc_on_open=True)
def create_my_control():
    # This inner function will be called after Excel has finished calculating
    def inner_func():
        # Create your widget using whichever supported UI toolkit you prefer
        widget = your_code_to_create_the_widget()

        # Using the same name each time means that only one control will be used,
        # even if this function is called multiple times.
        create_activex_control(widget, name="My Custom Control")

    # Calling our inner_func function once Excel has finished calculating
    schedule_call(inner_func)
```

See *Recalculating On Open* for more details about how the `recalc_on_open` option works.

4.10 Cell Formatting

See *Cell Formatting* for more information about cell formatting in PyXLL.

- *Formatter*
- *DataFrameFormatter*
- *DateFormatter*
- *ConditionalFormatter*
- *ConditionalFormatterBase*

class Formatter

Formatter for formatting values returned via `@xl_func`, or using `XLCell.options` and `XLCell.value`.

Use `Formatter.rgb` for constructing color values.

Formatters may be combined by adding them together.

Custom formatters should use this class as their base class.

See *Cell Formatting* for more details.

Parameters

- **interior_color** – Value to set the interior color to.
- **text_color** – Value to set the text color to.
- **bold** – If true, set the text style to bold.
- **italic** – If true, set the text style to italic.
- **font_size** – Value to set the font size to.
- **number_format** – Excel number format to use.
- **auto_fit** – Auto-fit to the content of the cells. May be True (fit column width), False (don't fit), 'columns' (fit column width), 'rows' (fit row width), 'both' (fit column and row width).
- **clear** – *New in PyXLL 5.11* If False, don't clear existing formatting (default is True)

apply(*self*, *cell*, *value=None*, *datatype=None*, *datatype_ndim=0*, *datatype_kwargs={}*, *transpose=False*)

The apply method is called to apply a formatter to a cell or range of cells.

It is called after a worksheet function decorated with `@xl_func` has returned if using the `formatter` kwarg. It can also be used directly with an `XLCell` instance from a macro function.

This method may be implemented by a sub-class for custom formatting. For array functions, if the formatter should be applied cell by cell for each cell in the range, use `apply_cell` instead.

Parameters

- **cell** – Instance of an `XLCell` the formatting is to be applied to.
- **value** – The value returned from the `@xl_func` or `XLCell.value`.
- **datatype** – The datatype of the value being formatted.
- **datatye_ndim** – The number of dimensions (0, 1 or 2) of the value being formatted.
- **datatype_kwargs** – The parameters of the datatype of the value being formatted.
- **transpose** – The transpose option from the `@xl_func` decorator.

When a value is returned from an `@xl_func` the formatter is applied after Excel has finished calculating.

The apply method is called with the value returned, and any details about the datatype of the returned value. This allows the formatter to apply formatting relevant to the returned datatype, and can be conditional on the returned value.

apply_cell(*self*, *cell*, *value=None*, *datatype=None*, *datatype_kwargs={}*)

For use by custom formatters.

If you need the formatter to be called for each individual cell when formatting an array formula, override this method instead of `Formatter.apply`.

Unlike `Formatter.apply` this method is called for each item in the returned value. If you need to apply formatting at the array level and the item level you may override both, but ensure you call the super-class method `Formatter.apply` from your override `apply` method.

Parameters

- **cell** – Instance of an *XLCell* the formatting is to be applied to.
- **value** – The value returned from the *@xl_func* or *XLCell.value*.
- **datatype** – The datatype of the value being formatted.
- **datatype_kwargs** – The parameters of the datatype of the value being formatted.

clear(*self*, *cell*)

Clear any formatting from a cell, or range of cells.

This is called before applying the formatter.

For a resizing array function, the cell passed to this `clear` method is the previous range that was formatted, allowing arrays to contract without leaving formatting of empty cells behind.

The default implementation clears all formatting, but this may be overridden in a sub-class if more selective clearing is required.

If `clear=False` was passed to the constructor, this method does nothing (**new in PyXLL 5.11**).

Parameters
cell – Instance of *XLCell* that should have its formatting cleared.

Returns
True if formatting was cleared, False otherwise (**new in PyXLL 5.11**)

apply_style(*cell*, *style*)

Apply a style dictionary to an instance of an *XLCell*.

This can be used to apply basic styling to a cell without having to use *XLCell.to_range* and `win32com`.

The style dictionary may have the following entries:

- interior_color: Interior color of the cell (see *Formatter.rgb*).
- text_color: Text color (see *Formatter.rgb*).
- bold: Set to True for bold text, False otherwise.
- italic: Set to True for italic text, False otherwise.
- font_size: Font size in points (int).
- number_format: Excel number format to apply to the cell.
- auto_fit: Auto-fit to the content of the cells. May be True (fit column width), False (don't fit), 'columns' (fit column width), 'rows' (fit row width), 'both' (fit column and row width).

Parameters

- **cell** – Instance of *XLCell* to apply the style to.
- **style** – Dict specifying the style to be applied.

rgb(*red*, *green*, *blue*)

Return a color value understood by Excel.

Excel colors are in the form 'BGR' instead of the usual 'RGB' and this utility method constructs color values from their RGB components.

Parameters

- **red** (*int*) – Red component between 0 and 255.
- **green** (*int*) – Green component between 0 and 255.
- **blue** (*int*) – Blue component between 0 and 255.

class DataFrameFormatter(*Formatter*)

Formatter for DataFrames.

For each argument expecting a `Formatter`, a dict may also be provided.

When a list of formatters is used (e.g. for the row or index formatters) the formatters will cycle through the list and repeat. For example, to format a table with striped rows only two row formatters are needed.

__init__(*rows=default_row_formatters*, *header=default_header_formatter*, *index=default_index_formatter*, *columns=none*, *conditional_formatters=None*, ***kwargs*)

Parameters

- **rows** – Formatter or list of formatters to be applied to the rows.
- **header** – Formatter to use for the header column names.
- **index** – Formatter or list of formatters to be applied to the index.
- **columns** – Dict of column name to formatter or list of formatters to be applied for specific columns (in addition to the any row formatters).
- **conditional_formatters** – A list of :py:class`ConditionalFormatters` to be applied in order after any other formatting has been applied.
- **kwargs** – Additional Formatter kwargs that will affect the entire formatted range.

default_row_formatters

List of two default formatters for alternating row formats.

default_header_formatter

Default formatter used for any column headers.

default_index_formatter

Default formatter used for any index columns.

class DateFormatter(*Formatter*)

Formatter for dates, times and datetimes.

All formats are in the standard Python datetime format.

This formatter tests the values and applies the relevant number format according to the type.

__init__(*date_format='%Y-%m-%d'*, *time_format='%H:%M:%S'*, *datetime_format=None*, ***kwargs*)

Parameters

- **date_format** – Format used for date values.
- **time_format** – Format used for time values.
- **datetime_format** – Format used for datetime values.
- **kwargs** – Base formatter kwargs.

If datetime_format is not specified then it is constructed by combining date_format and time_format.

class ConditionalFormatter(*ConditionalFormatterBase*)

Conditional formatter for use with `DataFrameFormatter`.

This can be used to apply formatting to a DataFrame that is conditional on the values in the DataFrame.

The ConditionalFormatter works by evaluating an expression on the DataFrame using `DataFrame.eval`. Rows where the expression returns True have a formatter applied to them. The formatting can be further restricted to one or more columns.

To apply different formats to different values use multiple ConditionalFormatters.

__init__(*expr*, *formatter*, *columns=None*, ***kwargs*)

Parameters

- **expr** – Boolean expression for selecting rows to which the formatter will apply.
- **formatter** – Formatter that will be applied to the selected cells.
- **columns** – Column name or list of columns that the formatter will be applied to. May also be a callable, in which case it should accept a DataFrame and return a column or list of columns.
- **kwargs** – Additional arguments passed to `DataFrame.eval` when selecting the rows to apply the formatter to.

The following example shows how to color rows green where column A is greater than 0 and red where column A is less than 0.

```
from pyxll import DataFrameFormatter, ConditionalFormatter, Formatter, xl_func

green_formatter = Formatter(interior_color=Formatter.rgb(0, 0xff, 0))
red_formatter = Formatter(interior_color=Formatter.rgb(0xff, 0, 0))

a_gt_zero = ConditionalFormatter("A > 0", formatter=green_formatter)
a_lt_zero = ConditionalFormatter("A < 0", formatter=red_formatter)

df_formatter = DataFrameFormatter(conditional_formatters=[
                                    a_gt_zero,
                                    a_lt_zero])

@xl_func("var x: dataframe", formatter=df_formatter, auto_resize=True)
def load_dataframe(x):
    # load a dataframe with column 'A'
    return df
```

class ConditionalFormatterBase

Base class for conditional formatters.

Subclass this class to create your own custom conditional formatters for use with *DataFrameFormatter*.

get_formatters(*self*, *df*)

Parameters

df – DataFrame to be formatted.

Returns

A new DataFrame with the same index and columns as 'df' with values being instances of the *Formatter* class.

Where no formatting is to be applied the returned DataFrame value should be None.

4.11 Tables

PyXLL can read and write Excel tables, as well as plain ranges of data.

Please see *Working with Tables* for more details.

class Table(*TableBase*)

The *Table* class is used to write an Excel table from a macro function using *XLCell.value*.

For example, the following will write a pandas `DataFrame` as a table in Excel.

```
from pyxll import xl_macro, XLCell, Table

@xl_macro
def write_excel_table():
    # Get an XLCell object for the cell 'A1' in the active sheet.
    # We could fully specify the range, for example "[Book1]Sheet1!A1" if
    # needed, or use a COM Range object instead of the address string.
    # The table will be written with this cell as the top left of the table.
    cell = XLCell.from_range("A1")

    # Create the DataFrame we want to write to Excel as a table
    df = your_code_to_construct_the_dataframe()

    # Construct a Table instance, wrapping our DataFrame
    table = Table(df, type="dataframe")

    # Write the table to Excel
    cell.value = table
```

The `Table` class can be used as a base class for custom table classes with methods overriden to customize the writing and updating of Excel tables.

The methods available to override are documented in the base class, `TableBase`.

__init__(*data*, *name=None*, *type=None*, *preserve_columns=False*, ***kwargs*)

Parameters

- **data** – `DataFrame` or other type that can be converted to `var[][]`.
- **name** – Table name (optional)
- **type** – Type signature for *data*. If None, data must be a pandas or polars DataFrame or a list of lists.
- **preserve_columns** – *New in PyXLL 5.11*

  If *True* and there is an existing Excel table, the columns in the Excel table are not modified.
- **kwargs** – Additional type parameters.

class TableBase

Base class of `Table`.

The `TableBase` class can be used as a base class for user defined table classes to customize how tables get written to Excel.

When writing a table to Excel, for example:

```
cell = XLCell.from_range(rng)
cell.value = Table(...)
```

the following happens:

1. `TableBase.find_table` is called to see if there is an existing `ListObject` object.
2. If no existing `ListObject` is found, `TableBase.create_table` is called.
3. If the `ListObject` size is different from that returned by `TableBase.rows` and `TableBase.columns`, `TableBase.resize_table` is called.
4. `TableBase.update_table` is called to update the data in the `ListObject` table object.
5. Finally, `TableBase.apply_filters` and `TableBase.apply_sorting` are called to apply any filtering and sorting required to the table.

> **Note**
>
> Knowledge of the *Excel Object Model* is required to write an implementation of this class.

com_package(*self*)

Return the `com_package` to use when passing COM objects to methods of this class.

Can be one of *pythoncom*, *win32com*, *comtypes* or *None* to use the default set in the pyxll.cfg file.

find_table(*self*, *xl_range*)

Finds the ListObject for the table.

This is called when writing a table to a range to find an existing table to update, if there is one.

Parameters
xl_range – Range where the table is being written to as a Range COM object.

Returns
ListObject COM object or None if not found.

create_table(*self*, *xl_range*)

Creates a new ListObject for the table.

This is called if `find_table` returns None.

Parameters
xl_range – Range where the table should be placed as a Range COM object.

Returns
ListObject COM object.

rows(*self*)

Return the number of rows in the table, excluding any header row.

columns(*self*)

Return the number of columns in the table.

resize_table(*self*, *xl_list_object*, *rows*, *columns*)

Resizes the ListObject to match the new data.

Parameters

- **xl_list_object** – Existing table as a ListObject COM object.
- **rows** – Number of rows to resize the table to (excluding header row).
- **columns** – Number of columns to resize the table to.

Returns
ListObject COM object.

update_table(*self*, *xl_list_object*)

Update the ListObject by setting the data on it.

Parameters
xl_list_object – Existing table as a ListObject COM object.

apply_filters(*self*, *xl_list_object*)

Apply any filters to the table object.

Parameters
xl_list_object – Existing table as a ListObject COM object.

apply_sorting(*self*, *xl_list_object*)

Apply any sorting to the table object.

Parameters

xl_list_object – Existing table as a ListObject COM object.

on_error(*self*, *exc_value*, *xl_range*)

Called if an error occurs when creating or updating a table.

Parameters

- **exc_value** – The exception object raised.
- **xl_range** – The range that was to be used for the table.

4.12 Errors and Exceptions

- *get_last_error*
- *ErrorContext*
- *ObjectCacheKeyError*
- *SpillError*

get_last_error(*xl_cell*)

When a Python function is called from an Excel worksheet, if an uncaught exception is raised PyXLL caches the exception and traceback as well as logging it to the log file.

The last exception raised while evaluating a cell can be retrieved using this function.

The cache used by PyXLL to store thrown exceptions is limited to a maximum size, and so if there are more cells with errors than the cache size the least recently thrown exceptions are discarded. The cache size may be set via the *error_cache_size* setting in the *config*.

When a cell returns a value and no exception is thrown any previous error is **not** discarded. This is because doing so would add additional performance overhead to every function call.

Parameters

xl_cell – An *XLCell* instance or a COM *Range* object (the exact type depends on the *com_package* setting in the *config*.

Returns

The last exception raised by a Python function evaluated in the cell, as a tuple *(type, value, traceback)*.

Example usage:

```
from pyxll import xl_func, xl_menu, xl_version, get_last_error
import traceback


@xl_func("xl_cell: string")
def python_error(cell):
    """Call with a cell reference to get the last Python error"""
    exc_type, exc_value, exc_traceback = pyxll.get_last_error(cell)
    if exc_type is None:
        return "No error"

    return "".join(traceback.format_exception_only(exc_type, exc_value))
```

(continues on next page)

(continued from previous page)

```
@xl_menu("Show last error")
def show_last_error():
    """Select a cell and then use this menu item to see the last error"""
    selection = xl_app().Selection
    exc_type, exc_value, exc_traceback = get_last_error(selection)

    if exc_type is None:
        xlcAlert("No error found for the selected cell")
        return

    msg = "".join(traceback.format_exception(exc_type, exc_value, exc_traceback))
    if xl_version() < 12:
        msg = msg[:254]

    xlcAlert(msg)
```

class ErrorContext

An ErrorContext is passed to any error handler specified in the pyxll.cfg file.

When an unhandled exception is raised, the error handler is called with a context object and the exception details.

See *Error Handling* for details about customizing PyXLL's error handling.

type

Type of function where the exception occurred.

Can be any of the attributes of the `ErrorContext.Type` class.

function_name

Name of the function being called when the error occurred.

This may be none if the error was not the result of calling a function (eg when `type == ErrorContext.Type.IMPORT`).

import_errors

Only applicable when `type == ErrorContext.Type.IMPORT`.

A list of `(modulename, (exc_type, exc_value, exc_traceback))` for all modules that failed to import.

ErrorContext.Type:

Enum-style type to indicate the origination of the error.

UDF

Used to indicate the error was raised while calling a UDF.

MACRO

Used to indicate the error was raised while calling a macro.

MENU

Used to indicate the error was raised while calling a menu function.

RIBBON

Used to indicate the error was raised while calling a ribbon function.

IMPORT

Used to indicate the error was raised while importing a Python module.

class ObjectCacheKeyError(*KeyError*)

`ObjectCacheKeyError` is raised when attempting to retrieve an object from PyXLL's object cache with an object handle missing from the cache.

See *Cached Objects* for details of how to pass Python objects between Python and Excel using PyXLL's object cache feature.

class SpillError(*RuntimeError*)

`SpillError` is raised when attempting to write Python data to Excel using `XLCell.value` that doesn't fit in the target range, and would cause existing data to be overwritten if the target range was to be resized, and resizing is enabled.

Resizing occurs when the value being written is larger than the cells referenced by the `XLCell` instance and, either `auto_resize=True` is passed to `XLCell.options`, or when writing an Excel table (tables are always resized to fit the data being written).

4.13 Utility Functions

- `schedule_call`
- `reload`
- `rebind`
- `xl_version`
- `get_config`
- `get_dialog_type`
- `cached_object_count`
- `get_event_loop`
- `get_registered_functions`

schedule_call(*func*, **args*, *delay=0*, *nowait=False*, *retries=0*, *retry_delay=0*, *retry_backoff=1.0*, *retry_filter=None*, *disable_calculation=False*, *disable_screen_updating=False*)

Schedule a function to be called after the current Excel calculation cycle has completed.

The function is called in an Excel macro context so it is safe to use `xl_app` and other COM and macro functions.

This can be used by worksheet functions that need to modify the worksheet where calling back into Excel would fail or cause a deadlock.

From Python 3.7 onwards when called from the PyXLL asyncio event loop and 'nowait' is not set this function returns an `asyncio.Future`. This future can be awaited on to get the result of the call (see warning about awaiting in an async UDF below).

NOTE: In the stubs version (not embedded in PyXLL) the function is always called immediately and will not retry.

Parameters

- **func** – Function or callable object to call in an Excel macro context at some time in the near future.
- **args** – Arguments to be passed to the the function.
- **delay** – Delay in seconds to wait before calling the function.
- **nowait** – Do not return a Future even if called from the asyncio event loop.
- **retries** – Integer number of times to retry.

- **retry_delay** – Time in seconds to wait between retries.
- **retry_backoff** – Multiplier to apply to 'retry_delay' after each retry. This can be used to increase the time between each retry by setting 'retry_backoff' to > 1.0.
- **retry_filter** – Callable that receives the exception value in the case of an error. It should return True if a retry should be attempted or False otherwise.
- **disable_calculation** – Disable automatic calculations while the callback is being called. This switches the Excel calculation mode to manual and restores it to its previous mode after the call is complete.

  New in PyXLL 5.2
- **disable_screen_updating** – Disable Excel's screen updating while the callback is being called. Screen updating is restored to its previous mode after the call is complete.

  New in PyXLL 5.2

Example usage:

```
from pyxll import xl_func, xl_app, xlfCaller, schedule_call

@xl_func(macro=True)
def set_values(rows, cols, value):
    """Copies 'value' to a range of rows x cols below the calling cell."""

    # Get the address of the calling cell
    caller = xlfCaller()
    address = caller.address

    # The update is done asynchronously so as not to block Excel
    # by updating the worksheet from a worksheet function.
    def update_func():
        xl = xl_app()
        xl_range = xl.Range(address)

        # get the cell below and expand it to rows x cols
        xl_range = xl.Range(range.Resize(2, 1), range.Resize(rows+1, cols))

        # and set the range's value
        xl_range.Value = value

    # Schedule calling the update function
    pyxll.schedule_call(update_func)

    return address
```

Note

This function doesn't allow passing keyword arguments to the schedule function. To do that, use functools.partial().

```
# Will schedule "print("Hello", flush=True)"
schedule_call(functools.partial(print, "Hello", flush=True))
```

Warning

> If called from an async UDF it is important *not* to await on the result of this call! Doing so is likely to cause a deadlock resulting in the Excel process hanging.
>
> This is because the scheduled call won't run until the current calculation has completed, so your function will not complete if awaiting for the result.

reload()

Causes the PyXLL addin and any modules listed in the config file to be reloaded once the calling function has returned control back to Excel.

If the 'deep_reload' configuration option is turned on then any dependencies of the modules listed in the config file will also be reloaded.

The Python interpreter is not restarted.

rebind()

Causes the PyXLL addin to rebuild the bindings between the exposed Python functions and Excel once the calling function has returned control back to Excel.

This can be useful when importing modules or declaring new Python functions dynamically and you want newly imported or created Python functions to be exposed to Excel without reloading.

Example usage:

```
from pyxll import xl_macro, rebind

@xl_macro
def load_python_modules():
    import another_module_with_pyxll_functions
    rebind()
```

xl_version()

Returns
the version of Excel the addin is running in, as a float.

- 8.0 => Excel 97
- 9.0 => Excel 2000
- 10.0 => Excel 2002
- 11.0 => Excel 2003
- 12.0 => Excel 2007
- 14.0 => Excel 2010
- 15.0 => Excel 2013
- 16.0 => Excel 2016

get_config() → ConfigParser.SafeConfigParser

Returns
the PyXLL config as a `ConfigParser.SafeConfigParser` instance

See also *Configuring PyXLL*.

get_dialog_type() → int

Returns

the type of the current dialog that initiated the call into the current Python function

`xlDialogTypeNone`

or `xlDialogTypeFunctionWizard`

or `xlDialogTypeSearchAndReplace`

`xlDialogTypeNone = 0`

`xlDialogTypeFunctionWizard = 1`

`xlDialogTypeSearchAndReplace = 2`

`cached_object_count()` → int

Returns the current number of cached objects.

When objects are returns from worksheet functions using the `object` or `var` type they are stored in an internal object cache and a handle is returned to Excel. Once the object is no longer referenced in Excel the object is removed from the cache automatically.

See *Cached Objects*.

`get_event_loop()` → asyncio.AbstractEventLoop

New in PyXLL 4.2

Get the async event loop used by PyXLL for scheduling async tasks.

If called in Excel and the event loop is not already running it is started.

If called outside of Excel then the event loop is returned without starting it.

Returns
asyncio.AbstractEventLoop

See *Asynchronous Functions*.

`get_registered_functions`(*func_type*) → list

New in PyXLL 5.13

Return a list of all functions registered using one of the PyXLL function decorators.

Parameters
`func_type` – String identifier for the type of function to return. Must be one of:

- `"xl_func"`
- `"xl_macro"`
- `"xl_menu"`
- `"xl_event"`
- `"xl_arg_type"`
- `"xl_return_type"`
- `"xl_on_open"`
- `"xl_on_close"`
- `"xl_on_reload"`
- `"xl_license_notifier"`

Returns

A list of dicts, one per registered function, each containing the function (keyed by `"func"`) and any non-default keyword arguments that were passed to the decorator when registering it.

For `"xl_func"` and `"xl_macro"` the dicts also include:

- `"signature"`: a human-readable signature string, e.g. `"int a, int b: int"`. The signature is normalized and incorporates all type information for the function, so it may not be identical to the signature string passed to `@xl_func` or `@xl_macro`.

- "arg_types": a list of {"name": ..., "type": ...} dicts, one per argument (including any *args/**kwargs argument). If the type was specified only using a type hint, this type will be that type hint, otherwise it's a string describing the full type specification. See *Argument and Return Types*.
- "return_type": the function's return type, in the same form as the "type" values in "arg_types".

Example usage:

```
from pyxll import xl_func, get_registered_functions

@xl_func
def add(x: int, y: int) -> int:
    """Add two numbers together."""
    return x + y

for func in get_registered_functions("xl_func"):
    print(func["name"], func["signature"], func["arg_types"], func["return_type
↪"])
```

4.14 LRU Cache

New in PyXLL 5.11

The following functions relate to PyXLL's *Least Recently Used* function cache.

See *Cached Functions* for additional details.

- *lru_cache_info*
- *lru_cache_clear*
- *CacheKey*
- *CachedValue*

lru_cache_info(*function=None*)

Returns a dictionary of stats for the LRU cache.

When called with a function, a dictionary with the following keys is returned if a cache for that function is found:

- maxsize - maximum number of cached reuslts, or 0 if unbounded
- currsize - current number of cached results
- hits - number of times a cached result has been returned
- misses - number of times the function was called without finding a cached result

If called with no function, a dictionary of function names to dictionaries as described above is returned for cached functions.

Parameters

function – Function to get cache info for, or None for all cached functions.

lru_cache_clear(*function=None*)

Clears cached result from the LRU cache.

When called with a function, cached results for that function only are cleared.

If called with no function, all cached results are cleared.

Parameters
function – Function to clear cached results for, or `None` for all cached functions.

class CacheKey

Hashable and comparable key object used for storing results in a custom function cache.

This is an opaque class with no public properties or methods.

See *Alternative Caching Methods* for details.

class CachedValue

Value stored and returned by custom function caches.

This class wraps the actual Python value returned from cached function, but also contains other information to reduce the work required each time the value is returned to Excel.

The underlying returned value is available as the `value` property, unless the function failed in which case the error is available using the `exc_info` property.

Both the `value` and `exc_info` objects should be considered immutable and modifying them in any way may result in unexpected behaviour.

value

Value returned previously by the cached function.

exc_info

`(exc_type, exc_value, exc_traceback)` tuple from a previous failed call of the cached function.

4.15 Excel Application Events

New in PyXLL 5.12

The `@xl_event` decorators are used to register Python functions that will be called when specific Excel Application events occur.

The events that can be handled are the same as the Excel Application events available through VBA, and the Python functions have the same arguments as their VBA counterparts.

For more detailed information please see *Excel Application Events* in the user guide.

- `@xl_event`
- `@xl_event.new_workbook`
- `@xl_event.sheet_selection_change`
- `@xl_event.sheet_before_double_click`
- `@xl_event.sheet_before_right_click`
- `@xl_event.sheet_activate`
- `@xl_event.sheet_deactivate`
- `@xl_event.sheet_calculate`
- `@xl_event.sheet_change`
- `@xl_event.workbook_open`
- `@xl_event.workbook_activate`
- `@xl_event.workbook_deactivate`
- `@xl_event.workbook_before_close`
- `@xl_event.workbook_before_save`

- *@xl_event.workbook_before_print*
- *@xl_event.workbook_new_sheet*
- *@xl_event.workbook_addin_install*
- *@xl_event.workbook_addin_uninstall*
- *@xl_event.window_resize*
- *@xl_event.window_activate*
- *@xl_event.window_deactivate*
- *@xl_event.sheet_follow_hyperlink*
- *@xl_event.sheet_pivot_table_update*
- *@xl_event.workbook_pivot_table_close_connection*
- *@xl_event.workbook_pivot_table_open_connection*
- *@xl_event.workbook_sync*
- *@xl_event.workbook_before_xml_import*
- *@xl_event.workbook_after_xml_import*
- *@xl_event.workbook_before_xml_export*
- *@xl_event.workbook_after_xml_export*
- *@xl_event.workbook_rowset_complete*
- *@xl_event.after_calculate*
- *@xl_event.sheet_pivot_table_after_value_change*
- *@xl_event.sheet_pivot_table_before_allocate_changes*
- *@xl_event.sheet_pivot_table_before_commit_changes*
- *@xl_event.sheet_pivot_table_before_discard_changes*
- *@xl_event.protected_view_window_open*
- *@xl_event.protected_view_window_before_edit*
- *@xl_event.protected_view_window_before_close*
- *@xl_event.protected_view_window_resize*
- *@xl_event.protected_view_window_activate*
- *@xl_event.protected_view_window_deactivate*
- *@xl_event.workbook_after_save*
- *@xl_event.workbook_new_chart*

@xl_event(*event: str*, *com_package: str = None*, *filter: Callable = None*)

Registers an Excel Application event handler.

This is the generic decorator that can be used for all event types.

Specific helper decorators are provided that can be used instead of this, and those are listed below.

Example:

```
from pyxll import xl_event

@xl_event("AfterCalculate")
```

(continues on next page)

(continued from previous page)

```
def on_after_calculate():
    print("AfterCalculate fired")
```

Parameters

- **event** – Excel Application event name, as it is defined in VBA.
- **com_package** – The COM package to use when passing COM arguments to the Python event hander.

  Can be either `pywin32` or `comtypes`.

  If not specified, the default can be specified using the `com_package` setting in the `[PYXLL]]` section of the pyxll.cfg config file. `pywin32` is used if no other default is specified.
- **filter** – Function to call prior to the event handler to test if the handler should be called or not.

  The filter function should have the same signature as the event handler.

`@xl_event.`**`new_workbook`**(*com_package: str = None, filter: Callable = None*)

Registers a `NewWorkbook` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.new_workbook
def on_new_workbook(workbook: object):
    print("NewWorkbook fired")
```

Equivalent to `@xl_event("NewWorkbook")`.

`@xl_event.`**`sheet_selection_change`**(*com_package: str = None, filter: Callable = None*)

Registers a `SheetSelectionChange` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.sheet_selection_change
def on_sheet_selection_change(sheet: object, target: object):
    print("SheetSelectionChange fired")
```

Equivalent to `@xl_event("SheetSelectionChange")`.

`@xl_event.`**`sheet_before_double_click`**(*com_package: str = None, filter: Callable = None*)

Registers a `SheetBeforeDoubleClick` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.sheet_before_double_click
def on_sheet_before_double_click(sheet: object, target: object, cancel: bool):
    print("SheetBeforeDoubleClick fired")
```

Equivalent to `@xl_event("SheetBeforeDoubleClick")`.

`SheetBeforeDoubleClick` takes a `cancel` argument, which is passed whatever value was left by any handler called before this one. Return `True` from the handler to cancel the action, the same as setting `Cancel = True` in VBA, or `False` to let it go ahead. Returning anything else, including `None`, leaves `cancel` as it was.

`@xl_event.`**`sheet_before_right_click`**(*com_package: str = None, filter: Callable = None*)

Registers a `SheetBeforeRightClick` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.sheet_before_right_click
def on_sheet_before_right_click(sheet: object, target: object, cancel: bool):
    print("SheetBeforeRightClick fired")
```

Equivalent to `@xl_event("SheetBeforeRightClick")`.

`SheetBeforeRightClick` takes a `cancel` argument, which is passed whatever value was left by any handler called before this one. Return `True` from the handler to cancel the action, the same as setting `Cancel = True` in VBA, or `False` to let it go ahead. Returning anything else, including `None`, leaves `cancel` as it was.

`@xl_event.`**`sheet_activate`**(*com_package: str = None, filter: Callable = None*)

Registers a `SheetActivate` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.sheet_activate
def on_sheet_activate(sheet: object):
    print("SheetActivate fired")
```

Equivalent to `@xl_event("SheetActivate")`.

`@xl_event.`**`sheet_deactivate`**(*com_package: str = None, filter: Callable = None*)

Registers a `SheetDeactivate` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.sheet_deactivate
def on_sheet_deactivate(sheet: object):
    print("SheetDeactivate fired")
```

Equivalent to `@xl_event("SheetDeactivate")`.

`@xl_event.`**`sheet_calculate`**(*com_package: str = None, filter: Callable = None*)

Registers a `SheetCalculate` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.sheet_calculate
def on_sheet_calculate(sheet: object):
    print("SheetCalculate fired")
```

Equivalent to `@xl_event("SheetCalculate")`.

`@xl_event.`**`sheet_change`**(*com_package: str = None, filter: Callable = None*)

Registers a `SheetChange` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.sheet_change
def on_sheet_change(sheet: object, target: object):
    print("SheetChange fired")
```

Equivalent to `@xl_event("SheetChange")`.

`@xl_event.`**`workbook_open`**(*com_package: str = None, filter: Callable = None*)

Registers a `WorkbookOpen` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_open
def on_workbook_open(workbook: object):
    print("WorkbookOpen fired")
```

Equivalent to `@xl_event("WorkbookOpen")`.

`@xl_event.workbook_activate`(*com_package: str = None, filter: Callable = None*)

Registers a `WorkbookActivate` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_activate
def on_workbook_activate(workbook: object):
    print("WorkbookActivate fired")
```

Equivalent to `@xl_event("WorkbookActivate")`.

`@xl_event.workbook_deactivate`(*com_package: str = None, filter: Callable = None*)

Registers a `WorkbookDeactivate` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_deactivate
def on_workbook_deactivate(workbook: object):
    print("WorkbookDeactivate fired")
```

Equivalent to `@xl_event("WorkbookDeactivate")`.

`@xl_event.workbook_before_close`(*com_package: str = None, filter: Callable = None*)

Registers a `WorkbookBeforeClose` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_before_close
def on_workbook_before_close(workbook: object, cancel: bool):
    print("WorkbookBeforeClose fired")
```

Equivalent to `@xl_event("WorkbookBeforeClose")`.

`WorkbookBeforeClose` takes a `cancel` argument, which is passed whatever value was left by any handler called before this one. Return `True` from the handler to cancel the action, the same as setting `Cancel = True` in VBA, or `False` to let it go ahead. Returning anything else, including `None`, leaves `cancel` as it was.

`@xl_event.workbook_before_save`(*com_package: str = None, filter: Callable = None*)

Registers a `WorkbookBeforeSave` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_before_save
def on_workbook_before_save(workbook: object, save_as_ui: bool, cancel: bool):
    print("WorkbookBeforeSave fired")
```

Equivalent to `@xl_event("WorkbookBeforeSave")`.

`WorkbookBeforeSave` takes a `cancel` argument, which is passed whatever value was left by any handler called before this one. Return `True` from the handler to cancel the action, the same as setting `Cancel = True` in VBA, or `False` to let it go ahead. Returning anything else, including `None`, leaves `cancel` as it was.

`@xl_event.workbook_before_print(com_package: str = None, filter: Callable = None)`

Registers a `WorkbookBeforePrint` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_before_print
def on_workbook_before_print(workbook: object, cancel: bool):
    print("WorkbookBeforePrint fired")
```

Equivalent to `@xl_event("WorkbookBeforePrint")`.

`WorkbookBeforePrint` takes a `cancel` argument, which is passed whatever value was left by any handler called before this one. Return `True` from the handler to cancel the action, the same as setting `Cancel = True` in VBA, or `False` to let it go ahead. Returning anything else, including `None`, leaves `cancel` as it was.

`@xl_event.workbook_new_sheet(com_package: str = None, filter: Callable = None)`

Registers a `WorkbookNewSheet` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_new_sheet
def on_workbook_new_sheet(workbook: object, sheet: object):
    print("WorkbookNewSheet fired")
```

Equivalent to `@xl_event("WorkbookNewSheet")`.

`@xl_event.workbook_addin_install(com_package: str = None, filter: Callable = None)`

Registers a `WorkbookAddinInstall` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_addin_install
def on_workbook_addin_install(workbook: object):
    print("WorkbookAddinInstall fired")
```

Equivalent to `@xl_event("WorkbookAddinInstall")`.

`@xl_event.workbook_addin_uninstall(com_package: str = None, filter: Callable = None)`

Registers a `WorkbookAddinUninstall` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_addin_uninstall
def on_workbook_addin_uninstall(workbook: object):
    print("WorkbookAddinUninstall fired")
```

Equivalent to `@xl_event("WorkbookAddinUninstall")`.

`@xl_event.window_resize(com_package: str = None, filter: Callable = None)`

Registers a `WindowResize` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.window_resize
def on_window_resize(workbook: object, wn: object):
    print("WindowResize fired")
```

Equivalent to `@xl_event("WindowResize")`.

`@xl_event.window_activate(com_package: str = None, filter: Callable = None)`

Registers a `WindowActivate` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.window_activate
def on_window_activate(workbook: object, wn: object):
    print("WindowActivate fired")
```

Equivalent to `@xl_event("WindowActivate")`.

`@xl_event.window_deactivate(com_package: str = None, filter: Callable = None)`

Registers a `WindowDeactivate` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.window_deactivate
def on_window_deactivate(workbook: object, wn: object):
    print("WindowDeactivate fired")
```

Equivalent to `@xl_event("WindowDeactivate")`.

`@xl_event.sheet_follow_hyperlink(com_package: str = None, filter: Callable = None)`

Registers a `SheetFollowHyperlink` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.sheet_follow_hyperlink
def on_sheet_follow_hyperlink(sheet: object, target: str):
    print("SheetFollowHyperlink fired")
```

Equivalent to `@xl_event("SheetFollowHyperlink")`.

`@xl_event.sheet_pivot_table_update(com_package: str = None, filter: Callable = None)`

Registers a `SheetPivotTableUpdate` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.sheet_pivot_table_update
def on_sheet_pivot_table_update(sheet: object, target: object):
    print("SheetPivotTableUpdate fired")
```

Equivalent to `@xl_event("SheetPivotTableUpdate")`.

`@xl_event.workbook_pivot_table_close_connection(com_package: str = None, filter: Callable = None)`

Registers a `WorkbookPivotTableCloseConnection` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_pivot_table_close_connection
def on_workbook_pivot_table_close_connection(workbook: object, target: object):
    print("WorkbookPivotTableCloseConnection fired")
```

Equivalent to `@xl_event("WorkbookPivotTableCloseConnection")`.

`@xl_event.workbook_pivot_table_open_connection(com_package: str = None, filter: Callable = None)`

Registers a `WorkbookPivotTableOpenConnection` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_pivot_table_open_connection
def on_workbook_pivot_table_open_connection(workbook: object, target: object):
    print("WorkbookPivotTableOpenConnection fired")
```

Equivalent to `@xl_event("WorkbookPivotTableOpenConnection")`.

`@xl_event.workbook_sync`(*com_package: str = None, filter: Callable = None*)

Registers a `WorkbookSync` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_sync
def on_workbook_sync(workbook: object, sync_event_type: int):
    print("WorkbookSync fired")
```

Equivalent to `@xl_event("WorkbookSync")`.

`@xl_event.workbook_before_xml_import`(*com_package: str = None, filter: Callable = None*)

Registers a `WorkbookBeforeXmlImport` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_before_xml_import
def on_workbook_before_xml_import(workbook: object, xml_map: object, url: str,␣
↪is_refresh: bool, cancel: bool):
    print("WorkbookBeforeXmlImport fired")
```

Equivalent to `@xl_event("WorkbookBeforeXmlImport")`.

`WorkbookBeforeXmlImport` takes a `cancel` argument, which is passed whatever value was left by any handler called before this one. Return `True` from the handler to cancel the action, the same as setting `Cancel = True` in VBA, or `False` to let it go ahead. Returning anything else, including `None`, leaves `cancel` as it was.

`@xl_event.workbook_after_xml_import`(*com_package: str = None, filter: Callable = None*)

Registers a `WorkbookAfterXmlImport` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_after_xml_import
def on_workbook_after_xml_import(workbook: object, xml_map: object, is_refresh:␣
↪bool, result: object):
    print("WorkbookAfterXmlImport fired")
```

Equivalent to `@xl_event("WorkbookAfterXmlImport")`.

`@xl_event.workbook_before_xml_export`(*com_package: str = None, filter: Callable = None*)

Registers a `WorkbookBeforeXmlExport` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_before_xml_export
def on_workbook_before_xml_export(workbook: object, xml_map: object, url: str,␣
↪cancel: bool):
    print("WorkbookBeforeXmlExport fired")
```

Equivalent to `@xl_event("WorkbookBeforeXmlExport")`.

WorkbookBeforeXmlExport takes a cancel argument, which is passed whatever value was left by any handler called before this one. Return True from the handler to cancel the action, the same as setting Cancel = True in VBA, or False to let it go ahead. Returning anything else, including None, leaves cancel as it was.

@xl_event.**workbook_after_xml_export**(*com_package: str = None, filter: Callable = None*)

Registers a WorkbookAfterXmlExport event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_after_xml_export
def on_workbook_after_xml_export(workbook: object, xml_map: object, url: str,␣
↪result: object):
    print("WorkbookAfterXmlExport fired")
```

Equivalent to @xl_event("WorkbookAfterXmlExport").

@xl_event.**workbook_rowset_complete**(*com_package: str = None, filter: Callable = None*)

Registers a WorkbookRowsetComplete event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_rowset_complete
def on_workbook_rowset_complete(workbook: object, description: str, sheet: str,␣
↪success: bool):
    print("WorkbookRowsetComplete fired")
```

Equivalent to @xl_event("WorkbookRowsetComplete").

@xl_event.**after_calculate**(*com_package: str = None, filter: Callable = None*)

Registers a AfterCalculate event handler. Example usage:

```
from pyxll import xl_event

@xl_event.after_calculate
def on_after_calculate():
    print("AfterCalculate fired")
```

Equivalent to @xl_event("AfterCalculate").

@xl_event.**sheet_pivot_table_after_value_change**(*com_package: str = None, filter: Callable = None*)

Registers a SheetPivotTableAfterValueChange event handler. Example usage:

```
from pyxll import xl_event

@xl_event.sheet_pivot_table_after_value_change
def on_sheet_pivot_table_after_value_change(sheet: object, target_pivot_table:␣
↪object, target_range: object):
    print("SheetPivotTableAfterValueChange fired")
```

Equivalent to @xl_event("SheetPivotTableAfterValueChange").

@xl_event.**sheet_pivot_table_before_allocate_changes**(*com_package: str = None, filter: Callable = None*)

Registers a SheetPivotTableBeforeAllocateChanges event handler. Example usage:

```
from pyxll import xl_event

@xl_event.sheet_pivot_table_before_allocate_changes
def on_sheet_pivot_table_before_allocate_changes(sheet: object, target_pivot_
```

(continues on next page)

(continued from previous page)

```
↪table: object, value_change_start: int, value_change_end: int, cancel: bool):
    print("SheetPivotTableBeforeAllocateChanges fired")
```

Equivalent to `@xl_event("SheetPivotTableBeforeAllocateChanges")`.

`SheetPivotTableBeforeAllocateChanges` takes a `cancel` argument, which is passed whatever value was left by any handler called before this one. Return `True` from the handler to cancel the action, the same as setting `Cancel = True` in VBA, or `False` to let it go ahead. Returning anything else, including `None`, leaves `cancel` as it was.

`@xl_event.`**`sheet_pivot_table_before_commit_changes`**(*com_package: str = None, filter: Callable = None*)

Registers a `SheetPivotTableBeforeCommitChanges` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.sheet_pivot_table_before_commit_changes
def on_sheet_pivot_table_before_commit_changes(sheet: object, target_pivot_
↪table: object, value_change_start: int, value_change_end: int, cancel: bool):
    print("SheetPivotTableBeforeCommitChanges fired")
```

Equivalent to `@xl_event("SheetPivotTableBeforeCommitChanges")`.

`SheetPivotTableBeforeCommitChanges` takes a `cancel` argument, which is passed whatever value was left by any handler called before this one. Return `True` from the handler to cancel the action, the same as setting `Cancel = True` in VBA, or `False` to let it go ahead. Returning anything else, including `None`, leaves `cancel` as it was.

`@xl_event.`**`sheet_pivot_table_before_discard_changes`**(*com_package: str = None, filter: Callable = None*)

Registers a `SheetPivotTableBeforeDiscardChanges` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.sheet_pivot_table_before_discard_changes
def on_sheet_pivot_table_before_discard_changes(sheet: object, target_pivot_
↪table: object, value_change_start: int, value_change_end: int):
    print("SheetPivotTableBeforeDiscardChanges fired")
```

Equivalent to `@xl_event("SheetPivotTableBeforeDiscardChanges")`.

`@xl_event.`**`protected_view_window_open`**(*com_package: str = None, filter: Callable = None*)

Registers a `ProtectedViewWindowOpen` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.protected_view_window_open
def on_protected_view_window_open(window: object):
    print("ProtectedViewWindowOpen fired")
```

Equivalent to `@xl_event("ProtectedViewWindowOpen")`.

`@xl_event.`**`protected_view_window_before_edit`**(*com_package: str = None, filter: Callable = None*)

Registers a `ProtectedViewWindowBeforeEdit` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.protected_view_window_before_edit
```

(continues on next page)

(continued from previous page)

```
def on_protected_view_window_before_edit(window: object, cancel: bool):
    print("ProtectedViewWindowBeforeEdit fired")
```

Equivalent to `@xl_event("ProtectedViewWindowBeforeEdit")`.

`ProtectedViewWindowBeforeEdit` takes a `cancel` argument, which is passed whatever value was left by any handler called before this one. Return `True` from the handler to cancel the action, the same as setting `Cancel = True` in VBA, or `False` to let it go ahead. Returning anything else, including `None`, leaves `cancel` as it was.

`@xl_event.`**`protected_view_window_before_close`**(*com_package: str = None, filter: Callable = None*)

Registers a `ProtectedViewWindowBeforeClose` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.protected_view_window_before_close
def on_protected_view_window_before_close(window: object, reason: int, cancel:␣
↪bool):
    print("ProtectedViewWindowBeforeClose fired")
```

Equivalent to `@xl_event("ProtectedViewWindowBeforeClose")`.

`ProtectedViewWindowBeforeClose` takes a `cancel` argument, which is passed whatever value was left by any handler called before this one. Return `True` from the handler to cancel the action, the same as setting `Cancel = True` in VBA, or `False` to let it go ahead. Returning anything else, including `None`, leaves `cancel` as it was.

`@xl_event.`**`protected_view_window_resize`**(*com_package: str = None, filter: Callable = None*)

Registers a `ProtectedViewWindowResize` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.protected_view_window_resize
def on_protected_view_window_resize(window: object):
    print("ProtectedViewWindowResize fired")
```

Equivalent to `@xl_event("ProtectedViewWindowResize")`.

`@xl_event.`**`protected_view_window_activate`**(*com_package: str = None, filter: Callable = None*)

Registers a `ProtectedViewWindowActivate` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.protected_view_window_activate
def on_protected_view_window_activate(window: object):
    print("ProtectedViewWindowActivate fired")
```

Equivalent to `@xl_event("ProtectedViewWindowActivate")`.

`@xl_event.`**`protected_view_window_deactivate`**(*com_package: str = None, filter: Callable = None*)

Registers a `ProtectedViewWindowDeactivate` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.protected_view_window_deactivate
def on_protected_view_window_deactivate(window: object):
    print("ProtectedViewWindowDeactivate fired")
```

Equivalent to `@xl_event("ProtectedViewWindowDeactivate")`.

`@xl_event.workbook_after_save(com_package: str = None, filter: Callable = None)`

Registers a `WorkbookAfterSave` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_after_save
def on_workbook_after_save(workbook: object, success: bool):
    print("WorkbookAfterSave fired")
```

Equivalent to `@xl_event("WorkbookAfterSave")`.

`@xl_event.workbook_new_chart(com_package: str = None, filter: Callable = None)`

Registers a `WorkbookNewChart` event handler. Example usage:

```
from pyxll import xl_event

@xl_event.workbook_new_chart
def on_workbook_new_chart(workbook: object, chart: object):
    print("WorkbookNewChart fired")
```

Equivalent to `@xl_event("WorkbookNewChart")`.

4.16 PyXLL Add-in Events

These decorators enable the user to register functions that will be called when certain events occur in the PyXLL addin.

For Excel *Application* event handlers, see *Excel Application Events*

- *@xl_on_open*
- *@xl_on_reload*
- *@xl_on_close*
- *@xl_license_notifier*

@xl_on_open

Decorator for callbacks that should be called after PyXLL has been opened and the user modules have been imported.

The callback takes a list of tuples of three three items: `(modulename, module, exc_info)`

When a module has been loaded successfully, `exc_info` is `None`.

When a module has failed to load, `module` is None and `exc_info` is the exception information `(exc_type, exc_value, exc_traceback)`.

Example usage:

```
from pyxll import xl_on_open

@xl_on_open
def on_open(import_info):
    for modulename, module, exc_info in import_info:
        if module is None:
            exc_type, exc_value, exc_traceback = exc_info
            ... do something with this error ...
```

@xl_on_reload

Decorator for callbacks that should be called after a reload is attempted.

The callback takes a list of tuples of three three items: `(modulename, module, exc_info)`

When a module has been loaded successfully, `exc_info` is `None`.

When a module has failed to load, `module` is None and `exc_info` is the exception information `(exc_type, exc_value, exc_traceback)`.

Example usage:

```
from pyxll import xl_on_reload, xlcCalculateNow

@xl_on_reload
def on_reload(reload_info):
    for modulename, module, exc_info in reload_info:
        if module is None:
            exc_type, exc_value, exc_traceback = exc_info
            ... do something with this error ...

    # print a message to the log file
    print("PyXLL reloaded")
```

@xl_on_close

Decorator for registering a function that will be called when Excel is about to close.

This can be useful if, for example, you've created some background threads and need to stop them cleanly for Excel to shutdown successfully. You may have other resources that you need to release before Excel closes as well, such as COM objects, that would prevent Excel from shutting down. This callback is the place to do that.

This callback is called when the user goes to close Excel. However, they may choose to then cancel the close operation but the callback will already have been called. Therefore you should ensure that anything you clean up here will be re-created later on-demand if the user decides to cancel and continue using Excel.

To get a callback when Python is shutting down, which occurs when Excel is finally quitting, you should use the standard atexit Python module. Python will not shut down in some circumstances (e.g. when a non-daemonic thread is still running or if there are any handles to Excel COM objects that haven't been released) so a combination of the two callbacks is sometimes required.

Example usage:

```
from pyxll import xl_on_close

@xl_on_close
def on_close():
    print("closing...")
```

@xl_license_notifier

Decorator for registering a function that will be called when PyXLL is starting up and checking the license key.

It can be used to alert the user or to email a support or IT person when the license is coming up for renewal, so a new license can be arranged in advance to minimize any disruption.

The registered function takes 3 arguments: name (string), expdate (datetime.date), days_left (int).

Example usage:

```
from pyxll import xl_license_notifier

@xl_license_notifier
```

(continues on next page)

(continued from previous page)

```
def my_license_notifier(name, expdate, days_left, is_perpetual):
    if days_left < 30:
        ... do something here...
```

4.17 Excel C API Functions

PyXLL exposes certain functions from the Excel C API. These mostly should only be called from a worksheet, menu, event handler, or macro functions. Some should only be called from macro-sheet equivalent functions if being called from a worksheet function[1].

Functions that can only be called from a macro or menu can be called from elsewhere using `schedule_call`. This allows these C API functions to be called as `schedule_call` schedules a function call on Excel's main thread and in a macro context.

- `xlfCaller`
- `xlSheetId`
- `xlfGetWorkspace`
- `xlfGetWorkbook`
- `xlfGetWindow`
- `xlfWindows`
- `xlfVolatile`
- `xlcAlert`
- `xlcCalculation`
- `xlcCalculateNow`
- `xlcCalculateDocument`
- `xlAsyncReturn`
- `xlAbort`
- `xlSheetNm`
- `xlfGetDocument`

xlfCaller()

Returns
calling cell as an `XLCell` instance.

Callable from any function, but most properties of XLCell are only accessible from macro sheet equivalent functions[1]

xlSheetId(*sheet_name*)

Returns
integer sheet id from a sheet name (e.g. '[Book1.xls]Sheet1')

xlfGetWorkspace(*arg_num*)

Parameters
arg_num (*int*) – number of 1 to 72 specifying the type of workspace information to return

[1] A macro sheet equivalent function is a function exposed using `@xl_func` with *macro=True*.

Returns
depends on arg_num

xlfGetWorkbook(*arg_num workbook=None*)

Parameters

- **arg_num** (*int*) – number from 1 to 38 specifying the type of workbook information to return
- **workbook** (*string*) – workbook name

Returns
depends on arg_num

xlfGetWindow(*arg_num, window=None*)

Parameters

- **arg_num** (*int*) – number from 1 to 39 specifying the type of window information to return
- **window** (*string*) – window name

Returns
depends on arg_num

xlfWindows(*match_type=0, mask=None*)

Parameters

- **match_type** (*int*) – a number from 1 to 3 specifying the type of windows to match

  1 (or omitted) = non-add-in windows

  2 = add-in windows

  3 = all windows
- **mask** (*string*) – window name mask

Returns
list of matching window names

xlfVolatile(*volatile*)

Parameters
volatile (*bool*) – boolean indicating whether the calling function is volatile or not.

Usually it is better to declare a function as volatile via the `@xl_func` decorator. This function can be used to make a function behave as a volatile or non-volatile function regardless of how it was declared, which can be useful in some cases.

Callable from a macro equivalent function only[Page 282, 1]

xlcAlert(*alert*)

Pops up an alert window.

Callable from a macro or menu function only[2]

Parameters
alert (*string*) – text to display

xlcCalculation(*calc_type*)

set the calculation type to automatic or manual.

Callable from a macro or menu function only[2]

[2] Some Excel functions can only be called from a macro context or menu context. To call them from another context use `schedule_call`.

Parameters

calc_type (*int*) – `xlCalculationAutomatic`

or `xlCalculationSemiAutomatic`

or `xlCalculationManual`

xlCalculationAutomatic = 1

xlCalculationSemiAutomatic = 2

xlCalculationManual = 3

xlcCalculateNow()

recalculate all cells that have been marked as dirty (i.e. have dependencies that have changed) or that are volatile functions.

Equivalent to pressing F9.

*Callable from a macro or menu function only*Page 283, 2

xlcCalculateDocument()

recalculate all cells that have been marked as dirty (i.e. have dependencies that have changed) or that are volatile functions for the current worksheet *only*

*Callable from a macro or menu function only*Page 283, 2

xlAsyncReturn(*handle*, *value*)

Used by asynchronous functions to return the result to Excel see *Asynchronous Functions*

This function can be called from any thread and doesn't have to be from a macro sheet equivalent function

Parameters

- **handle** (*object*) – async handle passed to the worksheet function
- **value** (*object*) – value to return to Excel

xlAbort(*retain=True*)

Yields the processor to other tasks in the system and checks whether the user has pressed ESC to cancel a macro or workbook recalculation.

Parameters

retain (*bool*) – If False and a break condition has been set it is reset, otherwise don't change the break condition.

Returns

True if the user has pressed ESC, False otherwise.

xlSheetNm(*sheet_id*)

Returns

sheet name from a sheet id (as returned by `xlSheetId` or `XLCell.sheet_id`).

xlfGetDocument(*arg_num*[, *name*])

Parameters

- **arg_num** (*int*) – number from 1 to 88 specifying the type of document information to return
- **name** (*string*) – sheet or workbook name

Returns

depends on arg_num

INDEX

Symbols

__init__() (*AsyncIterRTD method*), 232
__init__() (*AtxBridgeBase method*), 253
__init__() (*CTPBridgeBase method*), 250
__init__() (*ConditionalFormatter method*), 258
__init__() (*DataFrameFormatter method*), 258
__init__() (*DateFormatter method*), 258
__init__() (*IterRTD method*), 232
__init__() (*PlotBridgeBase method*), 247
__init__() (*Table method*), 260

A

ActivateTab() (*IRibbonUI method*), 244
ActivateTabMso() (*IRibbonUI method*), 245
ActivateTabQ() (*IRibbonUI method*), 245
ActiveXHost (*class in pyxll*), 253
address (*XLCell attribute*), 237
Application (*CustomTaskPane attribute*), 249
apply() (*Formatter method*), 256
apply_cell() (*Formatter method*), 256
apply_filters() (*TableBase method*), 261
apply_sorting() (*TableBase method*), 261
apply_style() (*Formatter method*), 257
AsyncIterRTD (*class in pyxll*), 232
AtxBridgeBase (*class in pyxll*), 253

C

cached_object_count() (*in module pyxll*), 267
CachedValue (*class in pyxll*), 269
CacheKey (*class in pyxll*), 269
can_export() (*PlotBridgeBase method*), 247
clear() (*Formatter method*), 257
close() (*AtxBridgeBase method*), 253
close() (*CTPBridgeBase method*), 250
columns() (*TableBase method*), 261
com_package() (*TableBase method*), 261
ConditionalFormatter (*class in pyxll*), 258
ConditionalFormatterBase (*class in pyxll*), 259
connect() (*RTD method*), 231
ContentControl (*CustomTaskPane attribute*), 249
Context (*IRibbonControl attribute*), 244
create_activex_control() (*in module pyxll*), 252
create_ctp() (*in module pyxll*), 248
create_table() (*TableBase method*), 261
CTPBridgeBase (*class in pyxll*), 250
CustomTaskPane (*class in pyxll*), 249

D

DataFrameFormatter (*class in pyxll*), 257
DateFormatter (*class in pyxll*), 258
default_header_formatter (*DataFrameFormatter attribute*), 258
default_index_formatter (*DataFrameFormatter attribute*), 258
default_row_formatters (*DataFrameFormatter attribute*), 258
Delete() (*CustomTaskPane method*), 250
detach() (*RTD method*), 231
disconnect() (*RTD method*), 231
DockPosition (*CustomTaskPane attribute*), 250
DockPositionRestrict (*CustomTaskPane attribute*), 250

E

ErrorContext (*class in pyxll*), 263
exc_info (*CachedValue attribute*), 269
export() (*PlotBridgeBase method*), 248

F

filter_message() (*AtxBridgeBase method*), 254
find_table() (*TableBase method*), 261
first_col (*XLRect attribute*), 239
first_row (*XLRect attribute*), 239
Formatter (*class in pyxll*), 255
formula (*XLCell attribute*), 237
from_range() (*XLCell method*), 236
function_name (*ErrorContext attribute*), 263

G

get_config() (*in module pyxll*), 266
get_dialog_type() (*in module pyxll*), 266
get_event_loop() (*in module pyxll*), 267
get_formatters() (*ConditionalFormatterBase method*), 259
get_hwnd() (*AtxBridgeBase method*), 253
get_hwnd() (*CTPBridgeBase method*), 250
get_last_error() (*in module pyxll*), 262
get_registered_functions() (*in module pyxll*), 267
get_ribbon_xml() (*in module pyxll*), 243
get_size_hint() (*PlotBridgeBase method*), 247
get_title() (*CTPBridgeBase method*), 250
get_type_converter() (*in module pyxll*), 240

H

Height (*CustomTaskPane attribute*), 249
height() (*AtxBridgeBase method*), 253

I

Id (*IRibbonControl attribute*), 244
IMPORT (*ErrorContext attribute*), 263
import_errors (*ErrorContext attribute*), 263
Invalidate() (*ActiveXHost method*), 253
Invalidate() (*IRibbonUI method*), 244
InvalidateControl() (*IRibbonUI method*), 244
InvalidateControlMso() (*IRibbonUI method*), 244
IRibbonControl (*class in pyxll*), 244
IRibbonUI (*class in pyxll*), 244
is_calculated (*XLCell attribute*), 237
IterRTD (*class in pyxll*), 231

L

last_col (*XLRect attribute*), 239
last_row (*XLRect attribute*), 239
load_image() (*in module pyxll*), 243
lru_cache_clear() (*in module pyxll*), 268
lru_cache_info() (*in module pyxll*), 268

M

MACRO (*ErrorContext attribute*), 263
MENU (*ErrorContext attribute*), 263

N

note (*XLCell attribute*), 237

O

Object (*in module pyxll*), 242
ObjectCacheKeyError (*class in pyxll*), 263
offset() (*XLCell method*), 239
on_close() (*AtxBridgeBase method*), 253
on_close() (*CTPBridgeBase method*), 250
on_error() (*TableBase method*), 262
on_timer() (*AtxBridgeBase method*), 254
on_timer() (*CTPBridgeBase method*), 251
on_window_closed() (*AtxBridgeBase method*), 253
on_window_closed() (*CTPBridgeBase method*), 251
on_window_destroyed() (*AtxBridgeBase method*), 254
on_window_destroyed() (*CTPBridgeBase method*), 251
options() (*XLCell method*), 237

P

plot() (*in module pyxll*), 246
PlotBridgeBase (*class in pyxll*), 247
post_attach() (*AtxBridgeBase method*), 253
post_attach() (*CTPBridgeBase method*), 250
pre_attach() (*AtxBridgeBase method*), 253
pre_attach() (*CTPBridgeBase method*), 250
process_message() (*AtxBridgeBase method*), 254
process_message() (*CTPBridgeBase method*), 251

R

rebind() (*in module pyxll*), 266
rect (*XLCell attribute*), 237
reload() (*in module pyxll*), 266
remove_ribbon_tab() (*in module pyxll*), 244
resize() (*XLCell method*), 239
resize_table() (*TableBase method*), 261
rgb() (*Formatter method*), 257
RIBBON (*ErrorContext attribute*), 263
rows() (*TableBase method*), 261
RTD (*class in pyxll*), 231
RTDAsyncGenerator (*in module pyxll*), 232
RTDGenerator (*in module pyxll*), 232

S

schedule_call() (*in module pyxll*), 264
set_error() (*RTD method*), 231
set_error() (*XLAsyncHandle method*), 230
set_ribbon_tab() (*in module pyxll*), 243
set_ribbon_xml() (*in module pyxll*), 243
set_value() (*XLAsyncHandle method*), 230
sheet_id (*XLCell attribute*), 237
sheet_name (*XLCell attribute*), 237
SpillError (*class in pyxll*), 264

T

Table (*class in pyxll*), 259
TableBase (*class in pyxll*), 260
Tag (*IRibbonControl attribute*), 244
Title (*CustomTaskPane attribute*), 249
to_range() (*XLCell method*), 238
translate_accelerator() (*AtxBridgeBase method*), 254
translate_accelerator() (*CTPBridgeBase method*), 251
type (*ErrorContext attribute*), 263
TypeParameters (*class in pyxll*), 241

U

UDF (*ErrorContext attribute*), 263
update_table() (*TableBase method*), 261

V

value (*CachedValue attribute*), 269
value (*RTD attribute*), 231
value (*XLCell attribute*), 237
Visible (*CustomTaskPane attribute*), 249

W

Width (*CustomTaskPane attribute*), 249
width() (*AtxBridgeBase method*), 253
Window (*CustomTaskPane attribute*), 249

X

xl_app() (*in module pyxll*), 235
xl_arg() (*in module pyxll*), 240
xl_arg_type() (*in module pyxll*), 241

xl_disable() (*in module pyxll*), 235
xl_event() (*in module pyxll*), 270
xl_event.after_calculate() (*in module pyxll*), 277
xl_event.new_workbook() (*in module pyxll*), 271
xl_event.protected_view_window_activate() (*in module pyxll*), 279
xl_event.protected_view_window_before_close() (*in module pyxll*), 279
xl_event.protected_view_window_before_edit() (*in module pyxll*), 278
xl_event.protected_view_window_deactivate() (*in module pyxll*), 279
xl_event.protected_view_window_open() (*in module pyxll*), 278
xl_event.protected_view_window_resize() (*in module pyxll*), 279
xl_event.sheet_activate() (*in module pyxll*), 272
xl_event.sheet_before_double_click() (*in module pyxll*), 271
xl_event.sheet_before_right_click() (*in module pyxll*), 271
xl_event.sheet_calculate() (*in module pyxll*), 272
xl_event.sheet_change() (*in module pyxll*), 272
xl_event.sheet_deactivate() (*in module pyxll*), 272
xl_event.sheet_follow_hyperlink() (*in module pyxll*), 275
xl_event.sheet_pivot_table_after_value_change() (*in module pyxll*), 277
xl_event.sheet_pivot_table_before_allocate_changes() (*in module pyxll*), 277
xl_event.sheet_pivot_table_before_commit_changes() (*in module pyxll*), 278
xl_event.sheet_pivot_table_before_discard_changes() (*in module pyxll*), 278
xl_event.sheet_pivot_table_update() (*in module pyxll*), 275
xl_event.sheet_selection_change() (*in module pyxll*), 271
xl_event.window_activate() (*in module pyxll*), 274
xl_event.window_deactivate() (*in module pyxll*), 275
xl_event.window_resize() (*in module pyxll*), 274
xl_event.workbook_activate() (*in module pyxll*), 273
xl_event.workbook_addin_install() (*in module pyxll*), 274
xl_event.workbook_addin_uninstall() (*in module pyxll*), 274
xl_event.workbook_after_save() (*in module pyxll*), 279
xl_event.workbook_after_xml_export() (*in module pyxll*), 277
xl_event.workbook_after_xml_import() (*in module pyxll*), 276
xl_event.workbook_before_close() (*in module pyxll*), 273
xl_event.workbook_before_print() (*in module pyxll*), 273
xl_event.workbook_before_save() (*in module pyxll*), 273
xl_event.workbook_before_xml_export() (*in module pyxll*), 276
xl_event.workbook_before_xml_import() (*in module pyxll*), 276
xl_event.workbook_deactivate() (*in module pyxll*), 273
xl_event.workbook_new_chart() (*in module pyxll*), 280
xl_event.workbook_new_sheet() (*in module pyxll*), 274
xl_event.workbook_open() (*in module pyxll*), 272
xl_event.workbook_pivot_table_close_connection() (*in module pyxll*), 275
xl_event.workbook_pivot_table_open_connection() (*in module pyxll*), 275
xl_event.workbook_rowset_complete() (*in module pyxll*), 277
xl_event.workbook_sync() (*in module pyxll*), 276
xl_func() (*in module pyxll*), 227
xl_license_notifier() (*in module pyxll*), 281
xl_macro() (*in module pyxll*), 233
xl_menu() (*in module pyxll*), 245
xl_on_close() (*in module pyxll*), 281
xl_on_open() (*in module pyxll*), 280
xl_on_reload() (*in module pyxll*), 280
xl_return() (*in module pyxll*), 241
xl_return_type() (*in module pyxll*), 241
xl_version() (*in module pyxll*), 266
xlAbort() (*in module pyxll*), 284
XlAsyncHandle (*class in pyxll*), 230
xlAsyncReturn() (*in module pyxll*), 284
xlcAlert() (*in module pyxll*), 283
xlcCalculateDocument() (*in module pyxll*), 284
xlcCalculateNow() (*in module pyxll*), 284
xlcCalculation() (*in module pyxll*), 283
XLCell (*class in pyxll*), 236
xlfCaller() (*in module pyxll*), 282
xlfGetDocument() (*in module pyxll*), 284
xlfGetWindow() (*in module pyxll*), 283
xlfGetWorkbook() (*in module pyxll*), 283
xlfGetWorkspace() (*in module pyxll*), 282
xlfVolatile() (*in module pyxll*), 283
xlfWindows() (*in module pyxll*), 283
XLRect (*class in pyxll*), 239
xlSheetId() (*in module pyxll*), 282
xlSheetNm() (*in module pyxll*), 284