PyXLL User Guide

Release 4.4.2

PyXLL Ltd.

Apr 06, 2020

Contents

CHAPTER 1

Introduction to PyXLL

1.1 What is PyXLL?

PyXLL is an Excel Add-In that enables developers to extend Excel's capabilities with Python code.

PyXLL makes Python a productive, flexible back-end for Excel worksheets, and lets you use the familiar Excel user interface to interact with other parts of your information infrastructure.

With PyXLL, your Python code runs in Excel using any common Python distribution(e.g. Anaconda, Enthought's Canopy or any other CPython distribution from 2.3 to 3.8).

Because PyXLL runs your own full Python distribution you have access to all third party Python packages such as NumPy, Pandas and SciPy and can call them from Excel.

Example use cases include:

- Calling existing Python code to perform calculations in Excel
- Data processing and analysis that's too slow or cumbersome to do in VBA
- Pulling in data from external systems such as databases
- Querying large datasets to present summary level data in Excel

- Exposing internal or third party libraries to Excel users

1.2 How does it work?

PyXLL runs Python code in Excel according to the specifications in its config file, in which you configure how Python is run and which modules PyXLL should load. When PyXLL starts up it loads those modules and exposes certain functions that have been tagged with PyXLL decorators.

For example, an Excel user defined function (UDF) to compute the n th Fibonacci number can be written in Python as follows:

```python
from pyxll import xl_func

@xl_func
def fib(n):
    "Naiive Fibonacci implementation."
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

The *xl_func*-decorated function `fib` is detected by PyXLL and exposed to Excel as a user-defined function.

Excel types are automatically converted to Python types based on an optional function signature. Where there is no simple conversion (e.g. when returning an arbitrary class instance from a method) PyXLL stores the Python object reference as a cell value in Excel. When another function is called with a reference to that cell PyXLL retrieves the object and passes it to the method. PyXLL keeps track of cells referencing objects so that once an object is no longer referenced by Excel it can be dereferenced in Python.

1.3 Before You Start

Existing users might want to study *What's new in PyXLL 4*. Those upgrading from earlier versions will should read "*Important notes for upgrading from previous versions*". If you prefer to learn by watching, perhaps you would prefer our video guides and tutorials.

Note that you cannot mix 32-bit and 64-bit versions of Excel, Python and PyXLL – they all must be the same.

Install the add-in according to the *installation instructions*, making sure to update the configuration file if necessary. For specific instructions about installing with Anaconda or Miniconda see userguide/anaconda.

Once PyXLL is installed you will be able to try out the examples workbook that is included in the download. All the code used in the examples workbook is also included in the download.

Note that any errors will be written to the log file, so if you are having difficulties always look in the log file to see what's going wrong, and if in doubt please contact us.

1.4 Next Steps

After you've installed PyXLL here is a simple exercise that will show you how to write your first Python user-defined function.

1.4.1 Calling a Python Function in Excel

One of the main features of PyXLL is being able to call a Python function from a formula in an Excel workbook.

First start by creating a new Python module and writing a simple Python function. To expose that function to Excel all you have to do is to apply the `xl_func` decorator to it.:

```
from pyxll import xl_func

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

Save your module and edit the *pyxll.cfg* file again to add your new module to the list of modules to load and add the directory containing your module to the pythonpath.

```
[PYXLL]
modules = <add the name of your new module here>

[PYTHON]
pythonpath = <add the folder containing your Python module here>
```

Go to the *Addins* menu in Excel and select *PyXLL -> Reload*. This causes PyXLL to reload the config and Python modules, allowing new and updated modules to be discovered.

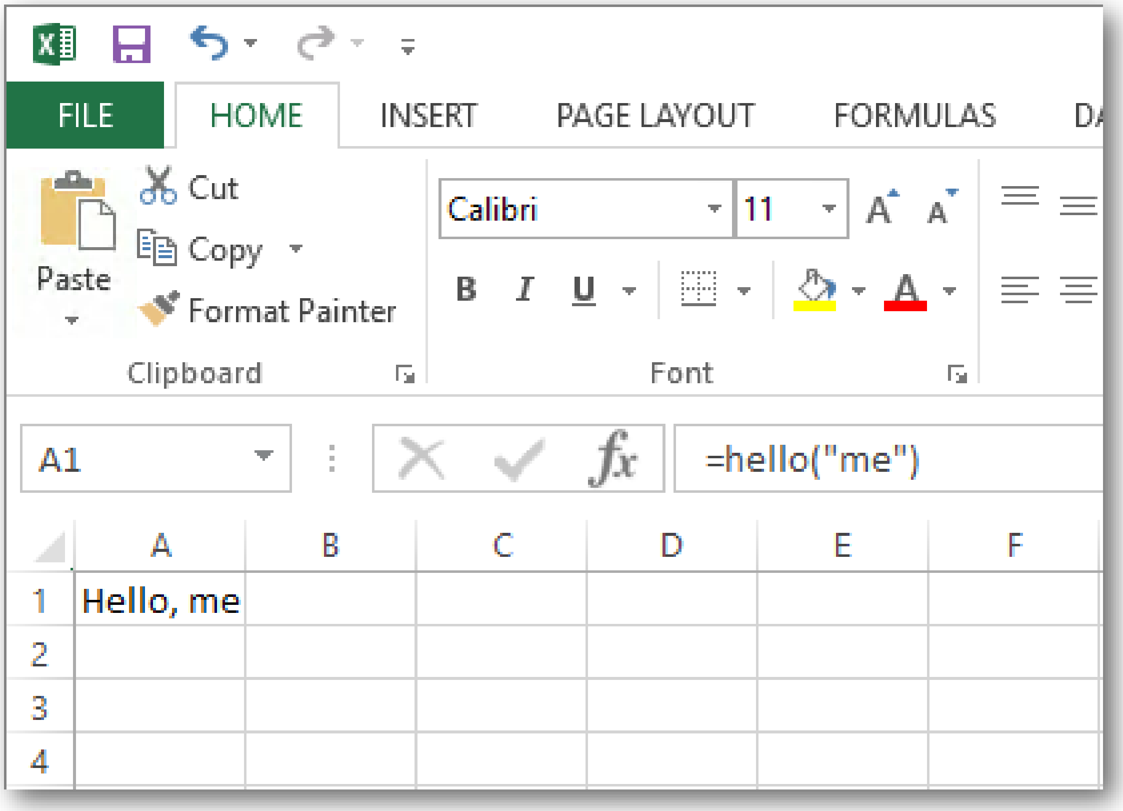
Now in a worksheet you will find you can type a formula using your new Python function.:

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

Using PyCharm, Eclipse or Visual Studio?

You can interactively debug Python code running in PyXLL with Eclipse, PyCharm, Visual Studio and other IDEs by attaching them as a debugger to a running PyXLL. See our blog post Debugging Your Python Excel Add-In for details.

If you make any mistakes in your code or your function returns an error you can check the log file to find out what the error was, make the necessary changes to your code and reload PyXLL again.



=hello("me")

	A	B	C	D	E	F
1	Hello, me					
2						
3						
4						

1.4.2 Additional Resources

The *documentation* explains how to use all the features of PyXLL, and contains a complete API reference. PyXLL's features are also well demonstrated in the examples included in download. These are a good place to start to learn more about what PyXLL can do.

More example code can be found on PyXLL's GitHub page.

If there is anything specifically you're trying to achieve and can't find an example or help in the documentation please contact us and we will do our best to help.

CHAPTER 2

What's new in PyXLL 4

Looking for an earlier version?

See 3.x/whatsnew for a detailed overview of the features added in PyXLL 3.

2.1 New Features and Improvements

2.1.1 Python 3.7 and 3.8 support added

PyXLL 4 is available for 32 and 64 bit versions of Python 3.7, and starting with PyXLL 4.3.0 Python 3.8 is also supported.

2.1.2 Auto-reloading (new in PyXLL 4.3)

PyXLL can now reload automatically whenever any modifications to your Python code or pyxll.cfg file are made.

To enable this set the following in your config file:

Note that *developer_mode* also has to be enabled.

This works with and without deep_reloading. If you are using deep reloading, any changes to any of the modules imported will cause a reload. If you are not using deep reloading, only changes to the modules listed in your pyxll.cfg file will cause a reload.

See *Reloading and Rebinding* for details.

2.1.3 Pandas DataFrame and Series support

The Pandas types `DataFrame` and `Series` can be used with PyXLL without the need for 3rd party packages like `pyxll_utils`.

When specifying a UDF function signature (see *Worksheet Functions*) the types `dataframe` and `series` may now be used to indicate that a range of data will be accepted by the function, and PyXLL will automatically construct a pandas DataFrame or Series to pass to the function (or if used as the return type, the pandas type will automatically be converted to a range of data before being returned to Excel).

Both the `dataframe` and `series` types can be parameterized to control exactly how the Excel data will be converted to and from the pandas types.

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

  `dataframe<index=0, columns=1, dtype=None, dtypes=None, index_dtype=None>`

  `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.

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*.

index_dtype Datatype for the values in the dataframe's index.

- **dataframe**, when used as a return type

  `dataframe<index=None, columns=True>`

  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.

- **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.

The Pandas types work with cached objects also, meaning you can return a pandas type as with the return type 'object' and an object handle will be returned to Excel, and pass that to a function with an argument type 'dataframe' or 'series' and the cached object will be passed to your function without having to reconstruct it.

2.1.4 Object Cache for returning complex types

Often it's necessary to pass a Python object between functions. You might have one function that creates a complex Python object that is used as an input to various other functions, and it's not convenient or efficient to keep recreating that object from lots of primitive inputs every time you need it.

PyXLL now allows you to pass objects around as easily as primitive values via its managed object cache. When a function returns a Python type that can't be converted to a simple Excel type, Python assigns it a unique handle and returns that to Excel. Any function that takes an object can be passed that handle and PyXLL will fetch the object from the cache and pass it to the Python method.

PyXLL takes care of making sure that once objects are no longer referenced they are removed from the cache so they can be destroyed.

Cached objects are not persisted when Excel closes, and they are not saved with the workbook. This means that every time you open a workbook that depends on cached objects it has to be recalculated so that the object cache is repopulated.

To use the object cache use the `object` type in your function signature. If you are using the `var` type then returning an object with no other type conversion will result in a cached object handle being returned.

RTD functions can return cached objects either implicitly with the return type `rtd`, or explicitly by using `rtd<object>`.

This replaces the `objectcache` example and `pyxll_utils.objectcache`.

2.1.5 One dimensional arrays

In previous versions of PyXLL, array types were always expected to be 2d lists of lists, eg:

```
x = [[1, 2, 3], [4, 5, 6]]
```

This was a source of confusion when passing a single row or column of data from Excel to Python as the (incorrectly) expected behaviour was that a simple 1d list of data would be passed to the function.

In PyXLL 4, it is now possible to specify in the function signature whether a function expects a 1d list of data or a 2d list of lists.

For a 1d list, use the `[]` type suffix, e.g:

```
@xl_func("float[] numbers: float")
def py_sum(numbers):
    total = 0
    for x in numbers:  # loop used to illustrate, could use 'sum' here
        total += x
    return total
```

For a 2d list of lists, use the `[][]` type suffix, e.g.:

```
@xl_func("float[][] numbers: float")
def py_sum(numbers):
    total = 0
    for row in numbers:
        for x in row:
            total += x
    return total
```

Note This change is a breaking change and if you are upgrading from a previous version of PyXLL you may want to disable this feature until you have updated your code. You can disable it by setting the following in your pyxll.cfg file:

```
[PYXLL]
always_use_2d_arrays = 1
```

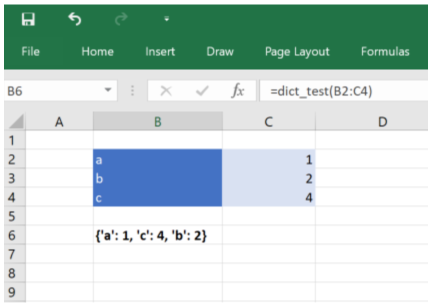
Tip When returning 1d arrays, use the *transpose* option to `xl_func` if you want to return the array as a row instead of a column (new in PyXLL 4.2).

2.1.6 Pass dictionaries between Python and Excel

A new `dict` type has been added for passing dictionaries between Python and Excel.

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.

```
@xl_func("dict<str, int>: str")
def dict_test(x):
    return str(x)
```



B6 =dict_test(B2:C4)

	A	B	C	D
1				
2		a	1	
3		b	2	
4		c	4	
5				
6		{'a': 1, 'c': 4, 'b': 2}		
7				
8				
9				

See *Dictionary Types* for more details.

2.1.7 RTD array formulas

RTD functions may now return arrays of data. This will simplify many cases where a table of ticking data is returned.

However, RTD array functions cannot be resized automatically.

Note: RTD functions may also now return cached objects. A useful pattern is to have an RTD function return a cached object containing the latest data, and another function to *explode* that cached object into an array of data.

2.1.8 Functions with variable number of arguments

In PyXLL 4 it is now possible to expose functions taking a variable number of arguments to Excel. Functions taking `*args` are automatically exposed to Excel as variable argument functions. It is only possible to use a single type for all of the `*args` however, so if you need them to be multiple different types you should use the `var` type.

For example, the following function takes a separator and a variable number of string arguments and joins them:

```
@xl_func("str sep, str *args: str")
def py_join(sep, *args):
    return sep.join(args)
```

2.1.9 Typed RTD and async values

Both types `rtd` and `async_handle` can now be parameterized with the return type. This is used when converting the returned value to Excel, so the conversion is no longer required to be done prior to returning the value.

e.g.

```
@xl_func("str x: rtd<object>")
def rtd_function(x):
    # Calling 'MyRtd.value = x' updates Excel with a new cached object handle
    return MyRtd(x)

@xl_func("str x: rtd<dataframe<index=True>>")
def rtd_function(x):
    # Calling 'MyDataFrameRtd.value = df' updates Excel with an array
    return MyDataFrameRtd(x)
```

2.1.10 Customizable error handling

When an unhandled Exception is raised during a worksheet function (UDF), previously PyXLL would log the exception and return an error to Excel. For other Excel function types (menus, macros and ribbon functions), the error was written to the log file.

This behaviour can now be customized, allowing more human readable errors to be returned to Excel.

A default error handler is provided that converts the unhandled Exception to a string and returns that for UDFs, and shows a dialog box with the error that occurred for other function types.

See *Error Handling* for details.

2.1.11 Parameterised custom types

As seen in some of the new features above, types in PyXLL can now be parameterized (e.g. `dataframe<index=1>` or `rtd<object>`).

Your own custom types registered with `xl_arg_type` and `xl_return_type` can also be parameterized in exactly the same way.

Any function decorated with `xl_arg_type` and `xl_return_type` with named keyword arguments can be used as a parameterized type. For example,

```
_lookups = {"MyCustomType:A": A, "MyCustomType:B", B}  # etc...
_ci_lookups = {k.lower(): v for k, v in _lookups.items()}  # lowercase lookup

# convert a named object to a Python value
@xl_arg_type("my_custom_type", "str")
def my_custom_type_from_var(x, case_sensitive=False):
    if case_insensitive:
        return _ci_lookups[x.lower()]
    return _lookups[x]

# this Excel function takes a string, which is converted via 'my_custom_type_from_var'
@xl_func("my_custom_type<case_insensitive=True> x: var")
def do_something(x):
    pass
```

The base type (`str` in the example above) can also be a function of the type parameters. Rather than using a string as the base type, a function taking the same keyword arguments can be used instead. That function returns the base type to be used. This can be used for even more generic types, e.g.:

```
def _get_base_type(dtype=float):
    if dtype == float:
        return "float[][]"
    elif dtype == int:
        return "int[][]"
    raise NotImplementedError()

@xl_arg_type("my_array_type", _get_base_type)
def my_custom_type_from_var(x, dtype=float):
    pass  # convert array into instance of 'my_array_type'

# this Excel function takes an array, which is converted via 'my_custom_type_from_var'
@xl_func("my_array_type<dtype=int> x: var")
def do_something(x):
    pass
```

As well as specifying the type parameters in the function signature, they can also be specified using *xl_arg* and *xl_return*, or as kwargs to *get_type_converter*.

2.1.12 Improved NumPy performance

NumPy floating point array arguments and return types are now *significantly* faster.

Tests show as much as a 15x speed up for passing floating point numpy arrays (e.g. `numpy_array<float>`) between Python and Excel.

2.1.13 asyncio support for UDFs and RTD functions

New in PyXLL 4.2 - requires Python 3.5.1 or higher

The Python keyword *async* introduced in Python 3.5 can now be used to declare an Excel function decorated with *xl_func* as being asynchronous.

Asynchronous functions declared in this way are run in an asyncio EventLoop, managed by PyXLL.

The *RTD* methods *RTD.connect* and *RTD.disconnect* may also now be asynchronous using the *async* keyword. They too will be run on the asyncio EventLoop managed by PyXLL.

This simplifies writing asynchronous code in Excel and reduces the need for user managed background threads and thread pools.

The asyncio EventLoop used by PyXLL can be obtained using the new *get_event_loop* function. The default behaviour is that this event loop runs on a single background thread, but this may be configured with an alternative implementation specified in the pyxll.cfg config file (see *Configuring PyXLL*).

2.1.14 Run a custom script when Excel starts (new in PyXLL 4.4)

Run a script when Excel starts to do any set up required as part of your PyXLL add-in.

This can be used for downloading the latest version of your Python code, or downloading a Python environment for the PyXLL add-in to use.

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

2.1.15 New win32com settings (new in PyXLL 4.4)

Control where the win32com auto-genereated wrapper class files are written to, and optionally clear them out each time Excel starts so they are always recreated.

If you've ever come across the following error then these new settings will help.:

```
AttributeError: module 'win32com.gen_py.00020813-0000-0000-C000-000000000046x0x1x9'␣
↪has no attribute 'CLSIDToPackageMap'
```

By putting the wrapper files somewhere other than the default location they won't conflict with other Python processes or other versions of win32com.

If they get corrupted (for example, if the process is interrupted while win32com is part way through writing a file) then you can delete them without affecting any other Python applications that might depend on them, or you can have PyXLL delete them for you each time Excel starts.

See *win32com Settings* for more details.

2.2 Important notes for upgrading from previous versions

PyXLL 4.0 contains some changes that may require you to make changes to your code and/or config before upgrading from previous versions.

2.2.1 The signature for array types has changed: replace [] with [][]

This can be disabled for backwards compatibility

In previous versions, `[]` was used to mean a 2d array or range of values that was passed to Python as a list of lists. As of PyXLL 4.0, arrays may now be either 1d or 2d, and `[]` is used to mean a 1d array. To pass 2d arrays you need to change your signature to use `[][]` when your function accepts or returns a list of lists (e.g. replace `float[]` with `float[][]`).

To disable this new feature and preserve the old behaviour where `[]` is used to indicate a 2d array or list of lists, set the following in your pyxll.cfg

```
[PYXLL]
always_use_2d_arrays = 1
```

2.2.2 Exceptions thrown in UDFs are returned as human readable strings

This can be disabled for backwards compatibility

A new config option allows an error handler to convert Exceptions raised when calling a worksheet function to a more human readable error message. If you are relying on Python functions to return Excel errors so that functions like `ISERROR` works, then remove or comment out this option in your config file.

```
[PYXLL]
error_handler = pyxll.error_to_string
```

2.2.3 Objects returned from UDFs are now returned as cached object handles

If you have functions with the return type unspecified or set to `var` that return non-trivial Python objects, then those objects will now be stored in the new object cache and a handle to that object will be displayed in Excel.

Previously objects of types not known to PyXLL were string converted using `str` and that string was returned to Excel. If you require this behaviour, change the return type of the function to `str`, or string convert the object before returning it from the function.

2.2.4 No need to use objectcache example: replace 'cached_object' with 'object'

This is not required, but recommended

If you were previously using the *Object Cache* example (or the object cache module from the *pyxll_utils* package) then you should now switch to using the built-in object cache.

The type to use in your function signatures that return or take an object as an argument is `object`. Functions accepting or returning `var` may also accept or return cached objects.

2.2.5 No need to use pyxll_utils.pandastypes

This is not required, but recommended

If you were previously using *pyxll_utils.pandastypes* or the *pandastypes.py* example code then you can now use the more powerful built-in types instead.

The datatype names are the same, so all that is required is to remove *pandastypes* from your pyxll.cfg file.

If you are using a customized version of the *pandastypes* code, it is recommended to rename your custom types so they do not conflict with the built in ones.

CHAPTER 3

User Guide

3.1 Installing the PyXLL Excel Add-In

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 anaconda.

3.1.1 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.*

3.1.2 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.

3.1.3 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.

3.1.4 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.

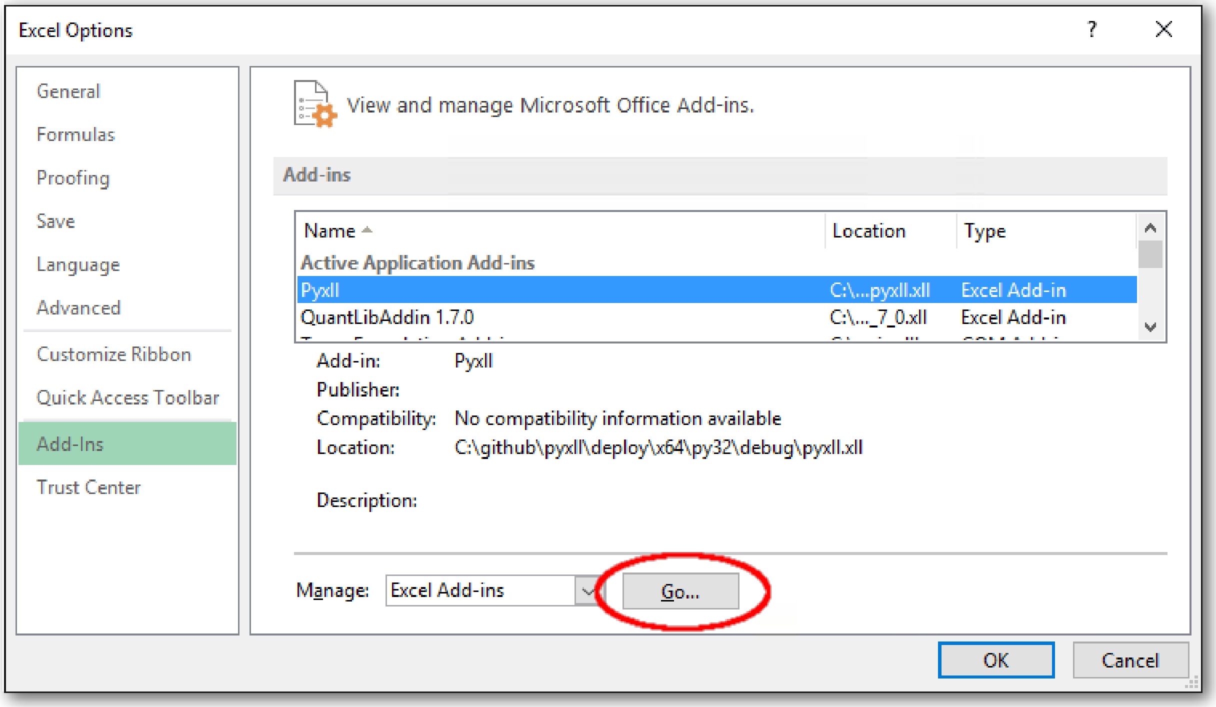
If setting the executable doesn't resolve the problem then it's possible your Python dll is in a non-standard location. You can set the dll location in the config to tell PyXLL where to find it.

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.



Name	Location	Type
Active Application Add-ins		
Pyxll	C:\...pyxll.xll	Excel Add-in
QuantLibAddin 1.7.0	C:\..._7_0.xll	Excel Add-in

Add-in: Pyxll
Publisher:
Compatibility: No compatibility information available
Location: C:\github\pyxll\deploy\x64\py32\debug\pyxll.xll

Description:

3.1.5 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).

3.1.6 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.

3.2 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 the PyXLL Excel Add-In*.

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.

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
```

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

3.2.2 PyXLL Settings

```
[PYXLL]
;
modules = comma or new line delimited list of python modules
ribbon = filename of a ribbon xml document
developer_mode = 1 or 0 indicating whether or not to use the developer mode
name = name of the addin visible in Excel
external_config = paths or URLs of additional config files to load
;
; reload settings
;
auto_reload = 1 or 0 to enable or disable automatic reloading
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␣
↪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
;
; 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
;
; 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, delete win32com's __gen_path__ folder when␣
↪starting
win32com_no_dynamic_dispatch = 1 or 0. If set, don't use win32com's dynamic wrappers
;
; error handling
;
error_handler = function for handling uncaught exceptions
error_cache_size = maximum number of exceptions to cache for failed function calls
;
; other settings
;
disable_com_addin = 1 or 0 to disable the COM addin component of PyXLL
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 of custom ribbon user interface XML file. The file name may be an absolute path or relative to the config file.

  The XML file 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.

  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'.

- **`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.

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.

- **`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.

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 set 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, delete the `win32com.__gen_path__` folder used for generated wrapper classes when PyXLL starts.

  If you have also set `win32com_gen_path`, that is the folder 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.

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.

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.

- **`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 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.

```
[LOG]
path = directory of where to write the log file
file = filename of the log file
format = format string
verbosity = logging level (debug, info, warning, error or critical)
encoding = encoding to use when writing the logfile (defaults to 'utf-8')
```

PyXLL creates some configuration substitution values that are useful when setting up logging.

Substitution Variable	Description
pid	process id
date	current date
xlversion	Excel version

- **`path`** Path where the log file will be written to.

  This may include substitution variables as listed above, e.g.

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

- **`file`** Filename of the log file.

  This may include substitution variables as listed above, e.g.

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

- **`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"
  ```

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

- **`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.

- **`encoding`** Encoding to use when writing the log file.

  Defaults to 'utf-8'.

  New in PyXLL 4.2.0.

3.2.4 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*.

- **`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*.

3.2.5 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 assert 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.

3.2.6 Startup Script

New in PyXLL 4.4.0

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

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-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"
```

3.2.7 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.

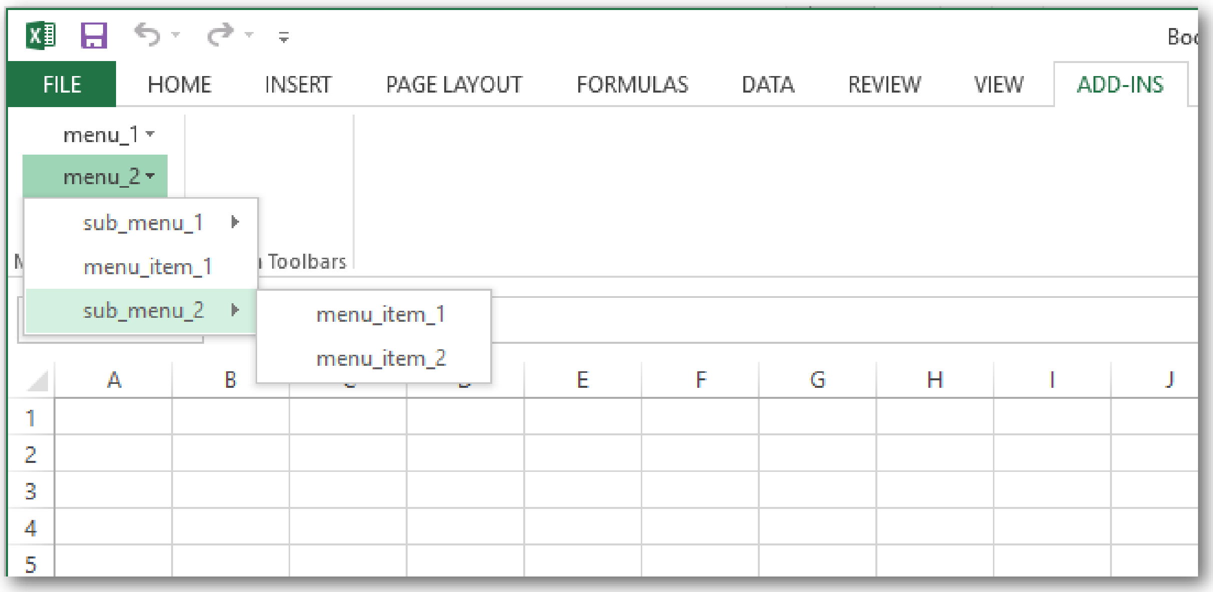
To specify the order of sub-menus and items within the sub-menus use a "." between the menu name, sub-menu name and item name.

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:



3.2.8 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.

3.3 Worksheet Functions

3.3.1 Introduction

"argument" vs. "parameter"

To avoid confusion we use the term *parameter(s)* to describe the formal argument specifications in the function definition, and *argument(s)* to indicate the actual values provided in a function call.

This section explains how to write PyXLL functions that handle simple values, and make those functions available in Excel.

If you've not installed the PyXLL addin yet, see *Installing the PyXLL Excel Add-In*.

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*).

Here's a simple example. Suppose you had the following file stored at *C:\Users\pyxll\modules\my_module.py*:

```python
from pyxll import xl_func

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

The decorator `xl_func` tells PyXLL to register the immediately following Python function as a worksheet function in Excel.

Once you have saved that code you need to ensure the interpreter can find it by adding the containing directory to the `pythonpath` setting and the module name to the `modules` setting in the *pyxll.cfg* config file:

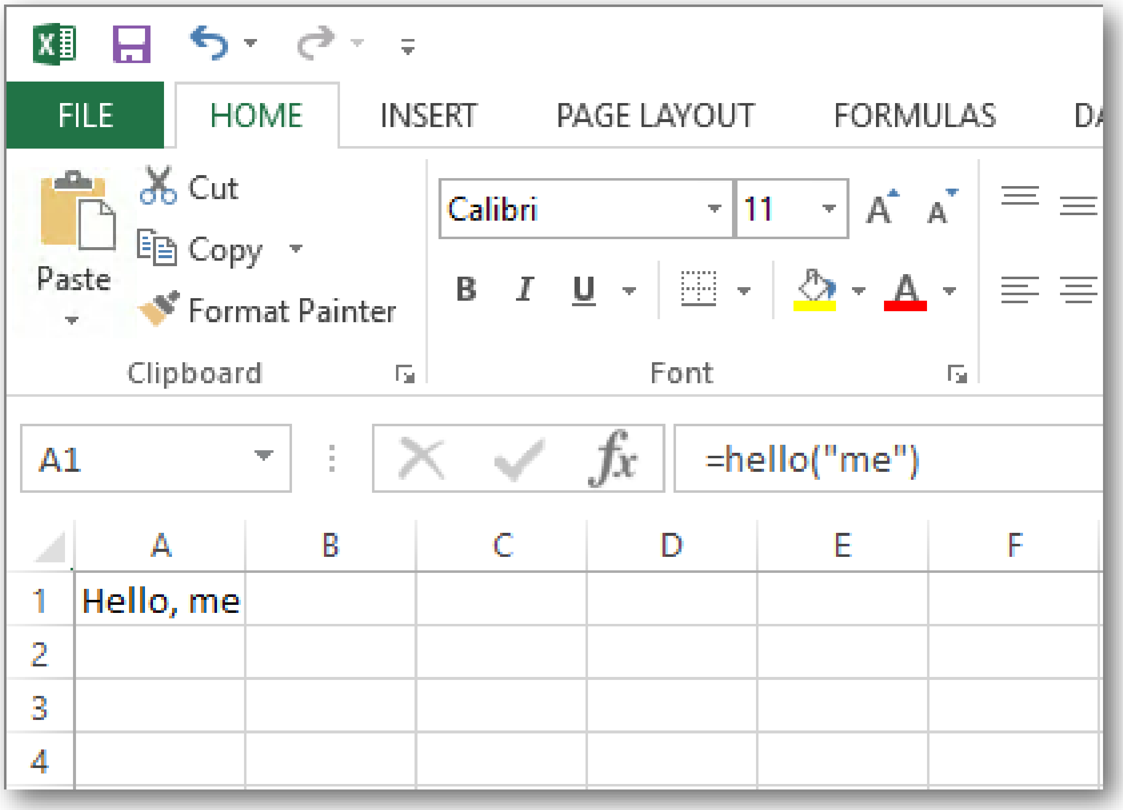
Tip: No Need to Restart Excel!

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 thr need to restart Excel itself.

```ini
[PYXLL]
;
; Make sure that PyXLL imports the module when loaded.
;
modules = my_module

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

If you make these changes and 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")
```



Worksheet functions can take simple values, as in the example above, or more complex arguments such as *arrays of data*. In particular, function arguments can be cell references, in which case the value passed to the function will be the value of the cell.

Examples of more complex types supported by PyXLL include *NumPy arrays*, *Pandas DataFrames and Series* and *Python objects*. You can add support for other types using PyXLL's *custom type system*.

In order for PyXLL to apply to correct type conversion the Python function must have a *function signature*.

3.3.2 Argument and Return Types

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.

@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 Function Annotations

Type information can also be provided using type annotations in Python 3. 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:

```python
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:

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

@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)
```

Standard Types

Simple Types

Several standard types may be used in the signature specifed when exposing a Python worksheet function. These types have a straighforward 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.

PyXLL type	Python type
bool	`bool`
datetime	`datetime.datetime`[1]
date	`datetime.date`
float	`float`
int	`int`
object	`object`[2]
rtd	*RTD*[3]
str	`str`
time	`datetime.time`
unicode	`unicode`[4]
var	`object`[5]
xl_cell	*XLCell*[6]
range	Excel Range COM Wrapper[7]

[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

Notes

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:

```python
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.

a date or time format.

[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*.

[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.

[5] The `var` type can be used when the argument or return type isn't fixed. Using the strong types 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.

[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 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.

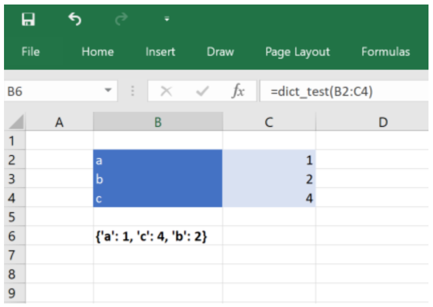
The default Python COM package used is `win32com`, but this can be changed via an argument to the range type. For example, to use `xlwings` instead of `win32com` you would use `range<xlwings>`.

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.

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



=dict_test(B2:C4)

	A	B	C	D
1				
2		a	1	
3		b	2	
4		c	4	
5				
6		{'a': 1, 'c': 4, 'b': 2}		
7				
8				
9				

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>`

  – `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.

- **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.

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:

```python
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.

Pandas Types

Pandas DataFrames and Series can be used as function arguments and return types for Excel worksheet functions.

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:

```python
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)
```

The following options are available for the `dataframe` and `series` argument and return types:

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

  `dataframe<index=0, columns=1, dtype=None, dtypes=None, index_dtype=None>`

  `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.

  `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*.

  `index_dtype` Datatype for the values in the dataframe's index.

- **dataframe**, when used as a return type

  `dataframe<index=None, columns=True>`

  `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.

- **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.

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.

See *Using Pandas in Excel* for more information.

Using Python Objects Directly

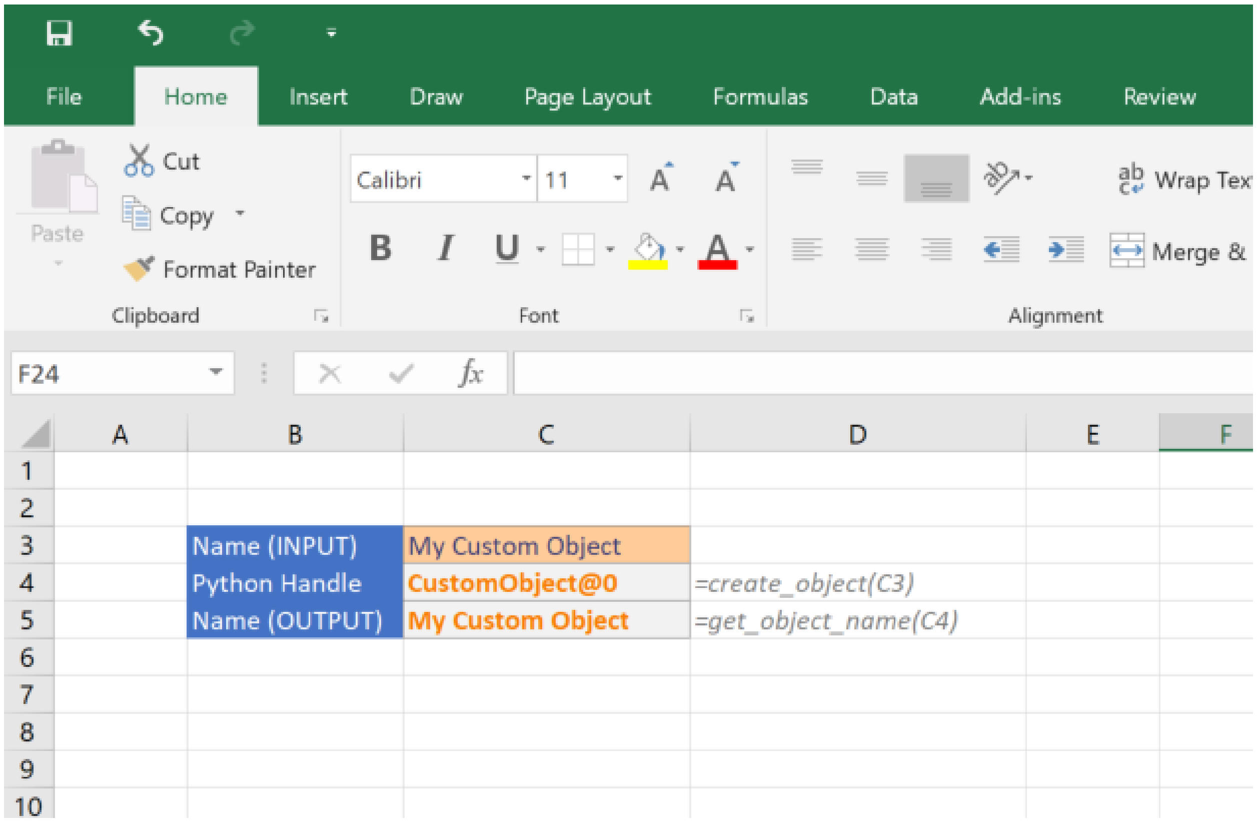
Not all Python types can be conveniently converted to a type that can be represented in Excel.

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. 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.

The following example shows one function that returns a Python object, and another that takes that Python object as an argument:

```python
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
```



	A	B	C	D	E	F
1						
2						
3		Name (INPUT)	My Custom Object			
4		Python Handle	CustomObject@0	=create_object(C3)		
5		Name (OUTPUT)	My Custom Object	=get_object_name(C4)		
6						
7						
8						
9						
10						

Note that the object is not copied. This means if you modify the object passed to your function then you will be modifying the object in the cache.

When using the `var` type, if an object of a type that has no converter is returned then the `object` type is used.

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 Excel first starts, or when PyXLL is reloaded, the cache is empty and so functions returning objects must be run to populate the cache. The easiest way to ensure all required cached objects have been created is to fully recalculate by pressing *Ctrl+Alt+F9*.

To fetch an object from the cache by its handle, `get_type_converter` can be used, e.g.:

```python
from pyxll import xl_func, get_type_converter

@xl_func("str handle: object")
def check_object_handle(handle):
    get_cached_object = get_type_converter("str", "object")
    obj = get_cached_object(handle)
    return obj
```

The method of generating object handles can be customized by setting `get_cached_object_id` in the `PYXLL` section of the config file.

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:

```python
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
    # 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:

```python
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 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.

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.

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*.

3.3.4 Asynchronous Functions

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.

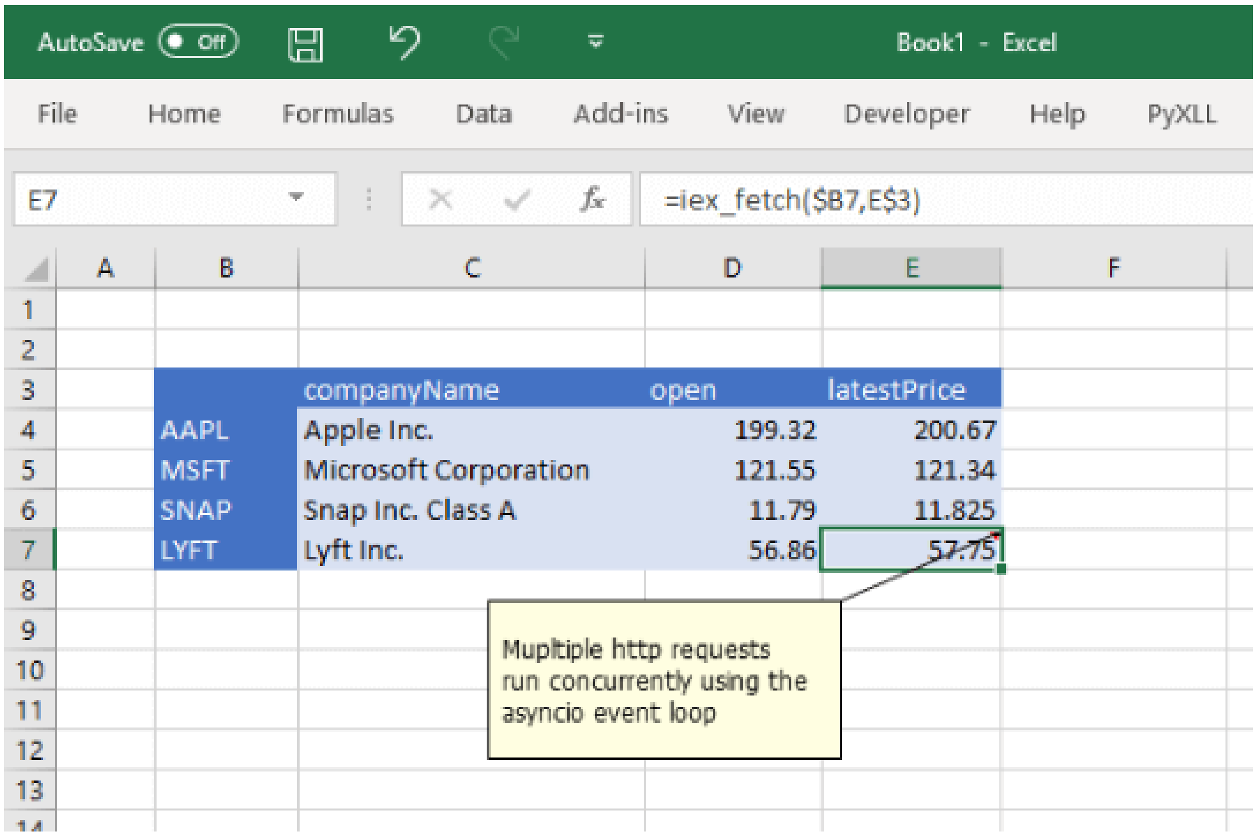
Excel asynchronous worksheet functions are called as part of Excel's calculation in the same way as other functions, but rather than return a result, they can schedule some work and return immediately, allowing Excel's calculation to progress while the scheduled work for the asynchronous function continues 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.

[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.



E7 =iex_fetch($B7,E$3)

	companyName	open	latestPrice
AAPL	Apple Inc.	199.32	200.67
MSFT	Microsoft Corporation	121.55	121.34
SNAP	Snap Inc. Class A	11.79	11.825
LYFT	Lyft Inc.	56.86	57.75

Mupltiple http requests run concurrently using the asyncio event loop

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]

```python
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:
```

[2] Asynchronous functions are only available in Excel 2010. Attempting to use them in an earlier version will result in an error.

```
        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:

```
@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[2]

```
from pyxll import xl_func, xlAsyncReturn
from threading import Thread
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.

3.3.5 Handling Errors

Exceptions raised by a UDF

Whenever an unhandled exception is raised in a Python function, PyXLL will write it to the log file and return an error to Excel.

If no error handler is set, an Excel error code will be returned. The exact error code returned depends on the exception type as follows:

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

You can customize PyXLL's error handling with the *error-handler* setting in the `PYXLL` section of the pyxll.cfg config file. You specify the function as a list of dotted references, much as you would for a Python import statement.

```
[PYXLL]
;
; Set custom error handler function
;
error_handler = your_module.error_handler_function
```

The error handler should be a function that takes three arguments: the exception type, the exception value and traceback of the uncaught exception, e.g.:

```
def error_handler_function(exc_type, exc_value, exc_traceback):
    """
    Convert a Python exception to a string value
    of form "<exception_type_name>: <value".
    """
    error = "##" + getattr(exc_type, "__name__", "Error")
    msg = str(exc_value)
    if msg:
        error += ": " + msg
    return error
```

When the error handler is executed, the return value of the error handler becomes the value of the cell from which it was referenced. See *Error Handling* for details.

Passing Errors as Values

Sometimes it is useful to be able to pass a cell value from Excel to python (or vice-versa) when the cell value is actually an error.

1. Any function with return type `var` (or a type that derives from it) will return an error code to Excel if an Exception is returned. The exact error code depends on the type of the exception, following the table in the section above.

   This is useful when you want to return an array of data (or other array-like data, e.g. a pandas DataFrame) and where only some values should be returned as errors. By setting the values that should be errors to instances of exceptions they will appear in Excel as errors.

2. Alternatively, the special type: *float_nan* can be used.

   float_nan behaves in almost exactly the same way as the normal *float* type. It can be used as an array type, or as an element type in a `numpy` array, e.g. *numpy_array<float_nan>*. The only difference is that if the Excel value is an error or a non-numeric type (e.g. an empty cell), the value passed to python will be *float('nan')* or *#QNAN!*, which is equivalent to `numpy.nan`.

   The two different float types exist because sometimes you don't want your function to be called if there's an error with the inputs, but sometimes you do. There is a slight performance penalty for using the *float_nan* type when compared to a plain *float*.

Retrieving Error Information

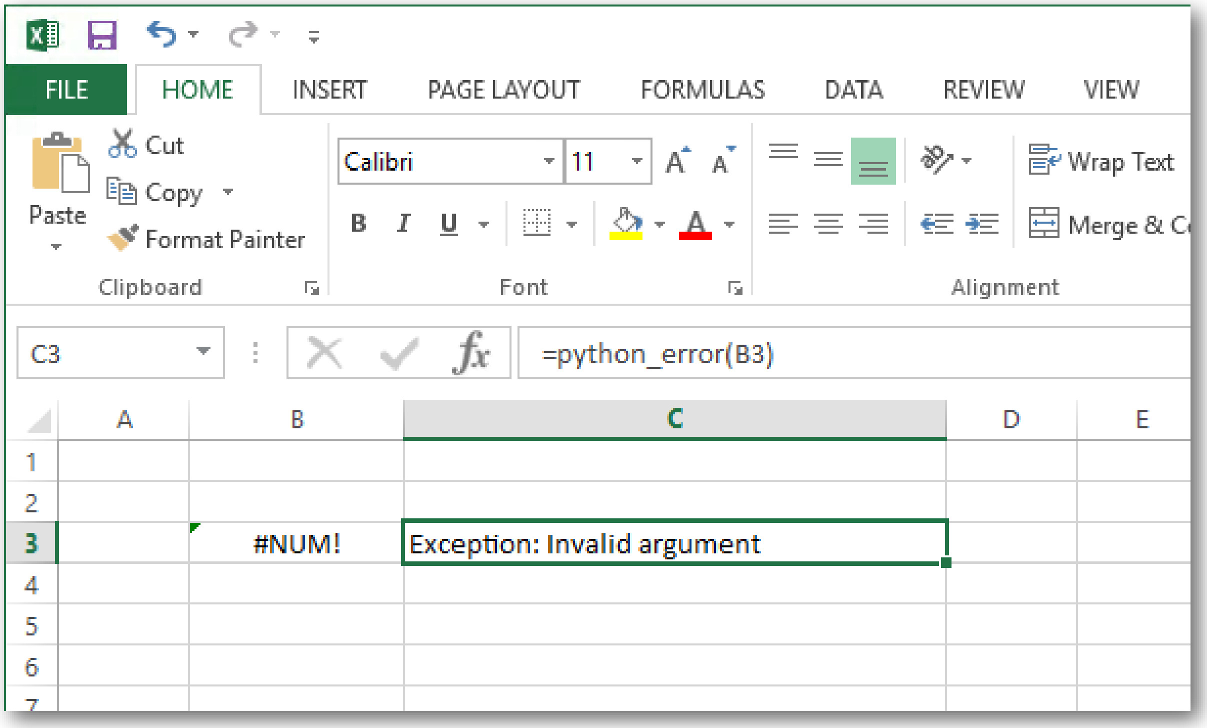
When a Python function is called from an Excel worksheet, if an uncaught exception is raised PyXLL caches the exception and traceback as well as logging it to the log file.

The last exception raised while evaluating a cell can be retrieved 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*.

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.



=python_error(B3)

	A	B	C	D	E
1					
2					
3		#NUM!	Exception: Invalid argument		
4					
5					
6					

```python
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)
```

3.3.6 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.

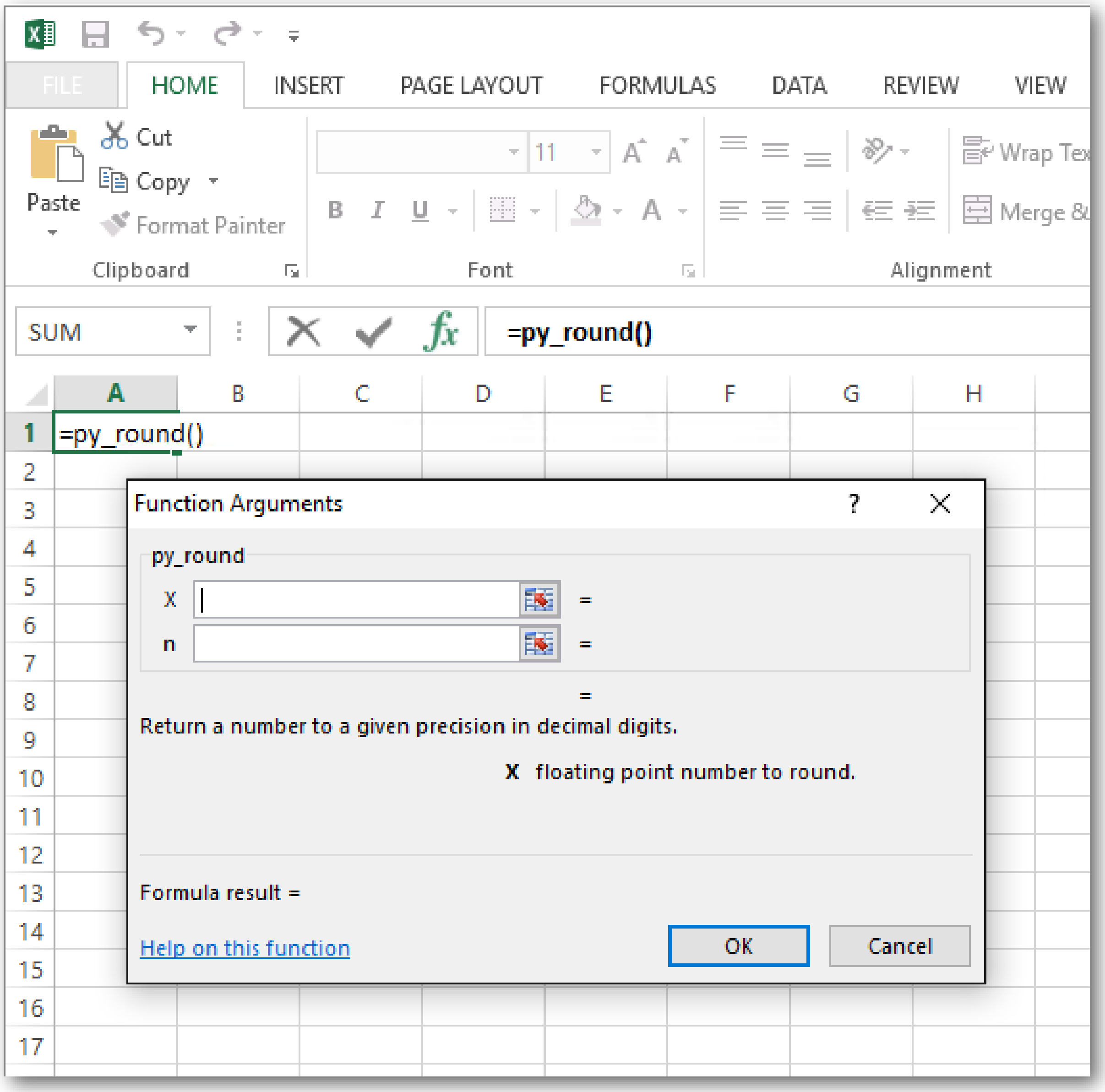
Parameter documentation may also be provided help the user know how to call the function. The most convenient way to add parameter documentation is to add it to the docstring as shown in the following example:

```python
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. They will appear in the function wizard as help strings for the parameters when selected. The first line will be used as the function description.

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.

As you can see, the arguments and documentation you provide are fully integrated with Excel's function wizard:



=py_round()

py_round

X =

n =

Return a number to a given precision in decimal digits.

X floating point number to round.

Formula result =

3.3.7 Variable Arguments

In Python it is possible to declare a function that takes a variable number of 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.

```python
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].

3.3.8 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.*

```ini
[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 Using Pandas in Excel

[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.

Pandas DataFrames and Series can be used as function arguments and return types for Excel worksheet functions using the decorator `xl_func`.

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 can automatically resize the range of the array formula to match the returned data by setting `auto_resize=True` in `xl_func`.

The following code shows a function that returns 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)
```

A function can also take a DataFrame or Series as one its arguments. When passing a DataFrame or Series to a function the whole data area must be selected in Excel and used as the argument to the function.

The following function takes a DataFrame including the column headers row, but not including the index column and returns the sum of a single column.:

```
from pyxll import xl_func

@xl_func("dataframe<index=False, columns=True>, str: float")
def sum_column(df, col_name):
    col = df[col_name]
    return col.sum()
```

3.4.1 Pandas Types Options

The following options are available for the `dataframe` and `series` argument and return types:

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

  `dataframe<index=0, columns=1, dtype=None, dtypes=None, index_dtype=None>`

  `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.

  `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*.

  `index_dtype` Datatype for the values in the dataframe's index.

- **dataframe**, when used as a return type

  `dataframe<index=None, columns=True>`

  `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.

- **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.

3.4.2 Passing as Python objects instead of Excel 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):
    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.

3.4.3 Using the Pandas type converters outside of a UDF

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*. Or, you might have a function that takes the `var` type, which could be a DataFrame depending on other arguments.

In these cases the function *get_type_converter* can be used. For example:

```python
from pyxll import get_type_converter

to_dataframe = get_type_converter("var", "dataframe<index=True>")
df = to_dataframe(data)
```

Or the other way:

```python
to_array = get_type_converter("dataframe", "var")
data = to_array(df)
```

All the parameters for the `dataframe` and `series` types can be used to control how the conversion is performed.

3.5 Menu Functions

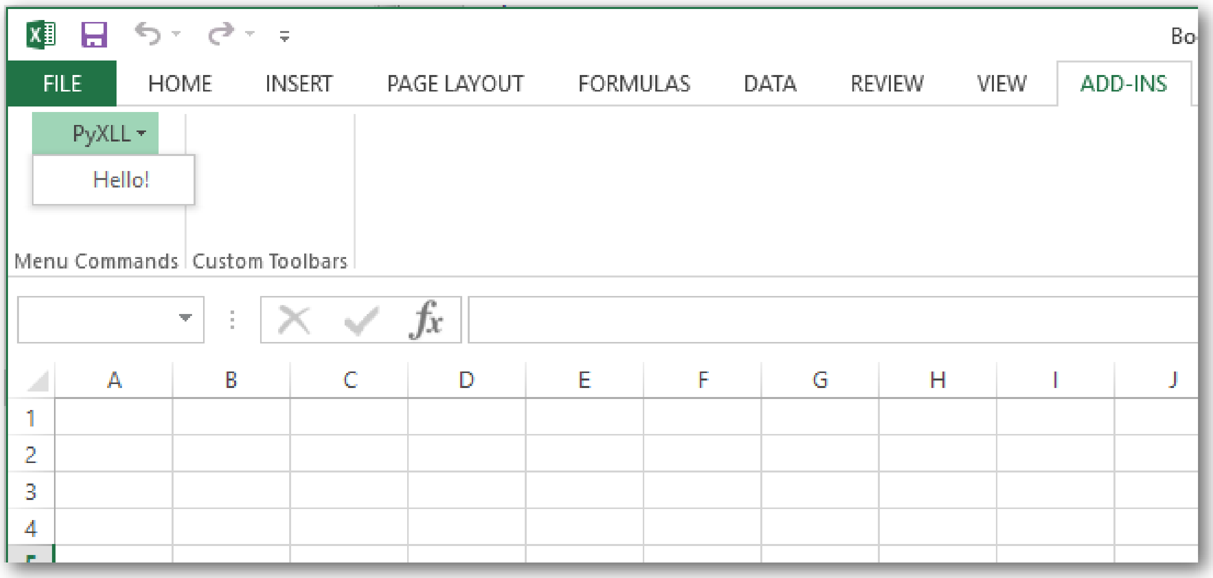
3.5.1 Custom Menu Items

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:

```python
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!"
```

3.5.2 New Menus

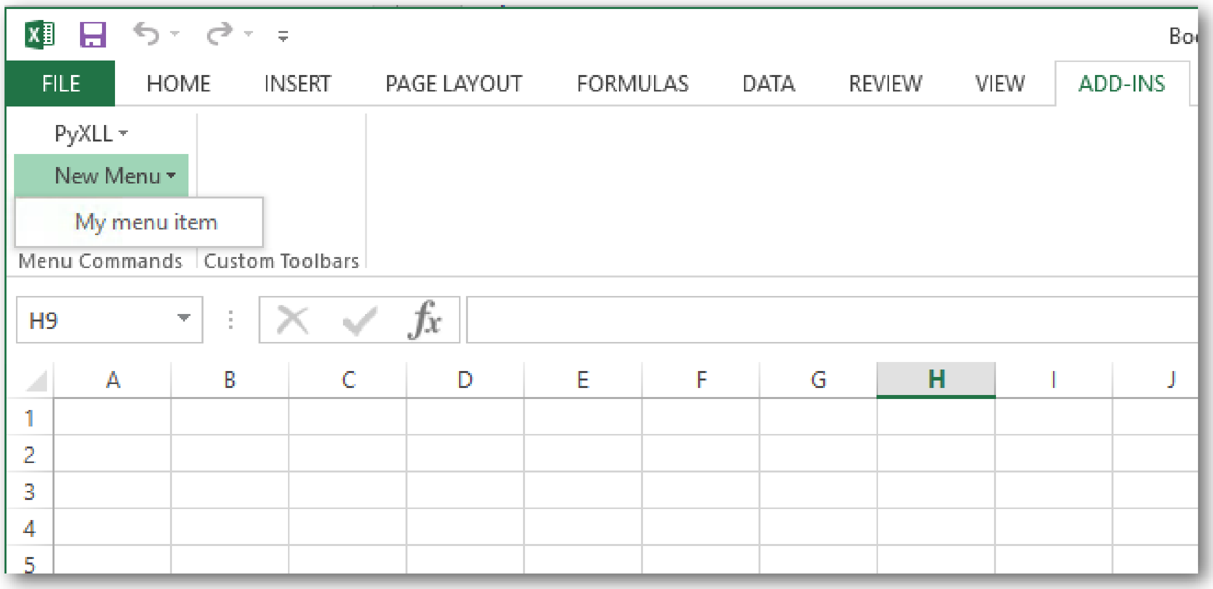
As well as adding menu items to the main PyXLL addin menu it's possible to create entirely new menus.

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")
```



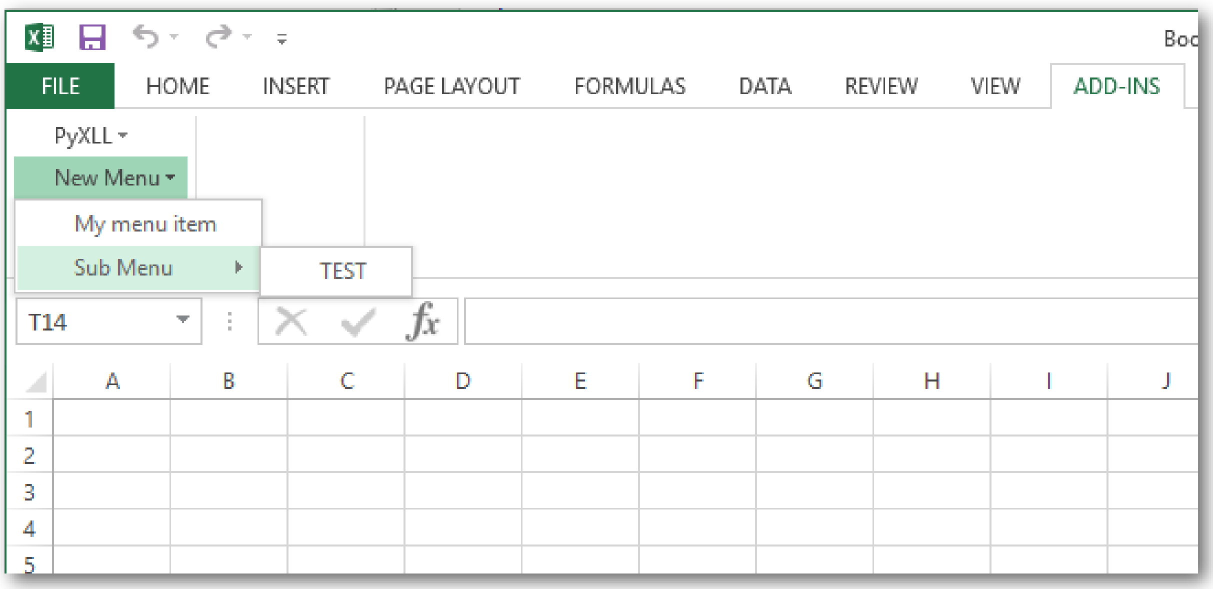
3.5.3 Sub-Menus

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")
```



3.6 Customizing the Ribbon

3.6.1 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.

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].

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

[1] Microsoft Ribbon Resources

- Ribbon XML
- Walkthrough: Creating a Custom Tab by Using Ribbon XML
- XML Schema Reference

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.

3.6.2 Creating a Custom Tab

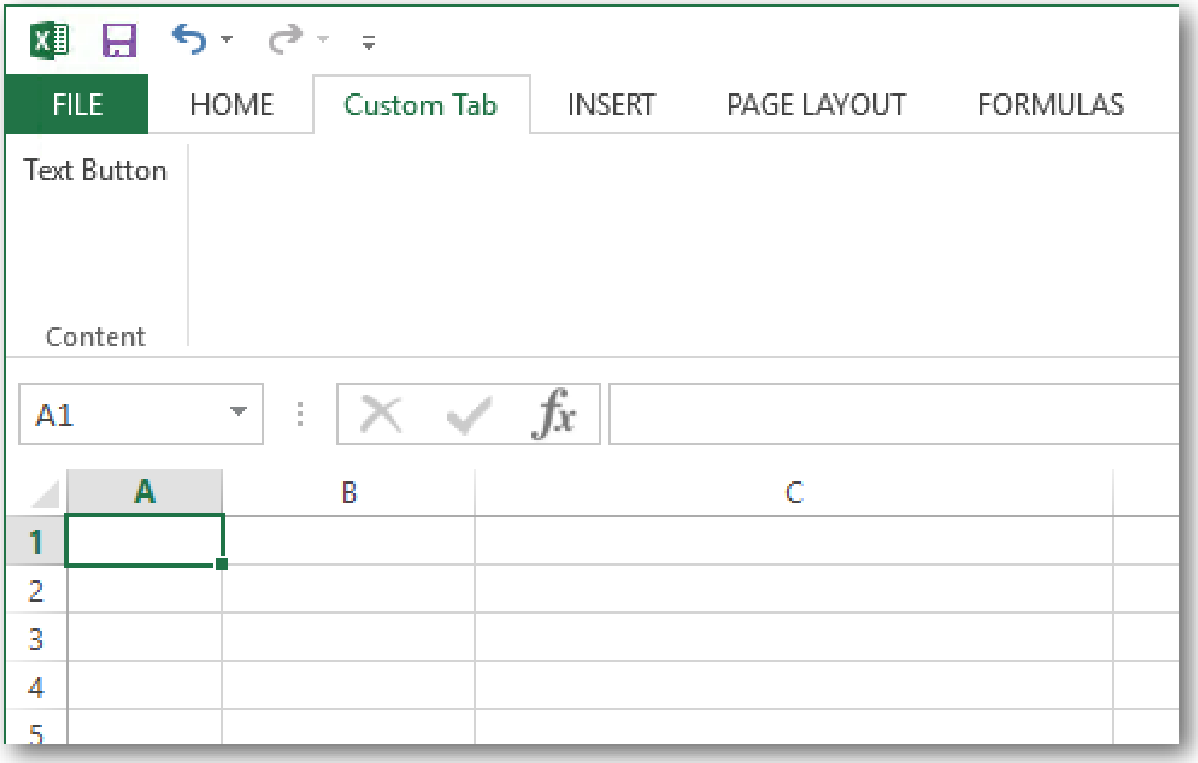
- Create a new ribbon xml file. The one below contains a single tab *Custom Tab* and a single button.

```
<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.

3.6.3 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.

3.6.4 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.

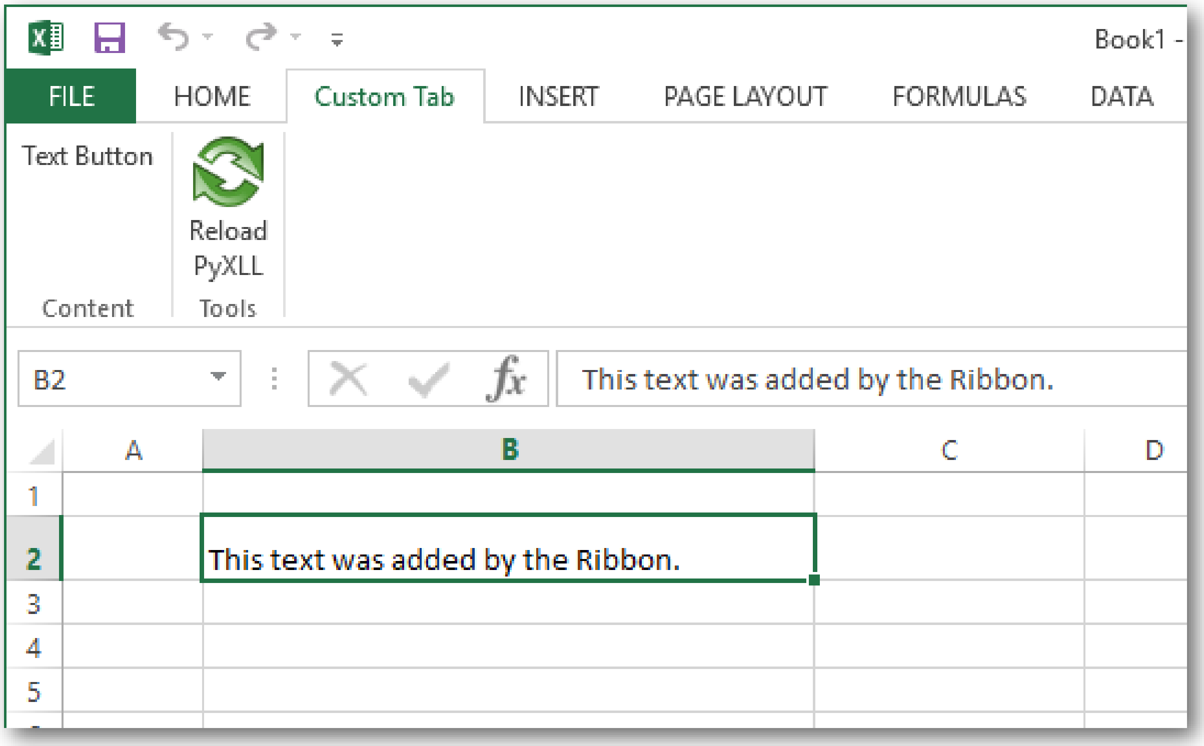
When images are referenced by filename using the *image* attribute Excel will load them using a basic image handler. This basic image handler is rather limited and doesn't handle PNG files with transparency, so it's recommended to use `load_image` instead. The image handler can be set as the *loadImage* attribute on the *customUI* element.

The following shows the example above with a new button added and the *loadImage* handler set.

[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.

```
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui"
          loadImage="pyxll.load_image">
    <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>
                <group id="Tools" label="Tools">
                    <button id="Reload"
                            size="large"
                            label="Reload PyXLL"
                            onAction="pyxll.reload"
                            image="reload.png"/>
                </group>
            </tab>
        </tabs>
    </ribbon>
</customUI>
```



This text was added by the Ribbon.

3.6.5 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.

3.7 Context Menu Functions

3.7.1 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.

3.7.2 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.

[1] XML Schema Reference

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()

        # 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.

3.7.3 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

        # skip any cells that don't contain text
        if not isinstance(value, str):
```

[2] This isn't strictly necessary but is helpful as it means the module will be reloaded when PyXLL is reloaded.

```
            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()
```

3.7.4 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">
            <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
```

3.7.5 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

3.8 Macro Functions

3.8.1 Introduction

You can write an Excel macro in python to do whatever you would previously have used VBA for. Macros work in a very similar way to worksheet functions. To register a function as a macro you use the `xl_macro` decorator.

Macros are useful as they can be called when GUI elements (buttons, checkboxes etc.) fire events. They can also be called from VBA.

Macro functions can call back into Excel using the Excel COM API (which is identical to the VBA Excel object model). 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 also *Python as a VBA Replacement*.

3.8.2 Exposing Functions as Macros

Python functions to be exposed as macros are decorated with the *xl_macro* decorator imported from the pyxll module.

```
from pyxll import xl_macro, xl_app, xlcAlert

@xl_macro
def popup_messagebox():
    xlcAlert("Hello")

@xl_macro
def set_current_cell(value):
    xl = xl_app()
    xl.Selection.Value = value

@xl_macro("string n: int")
def py_strlen(n):
    return len(x)
```

3.8.3 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, xl_app

@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):

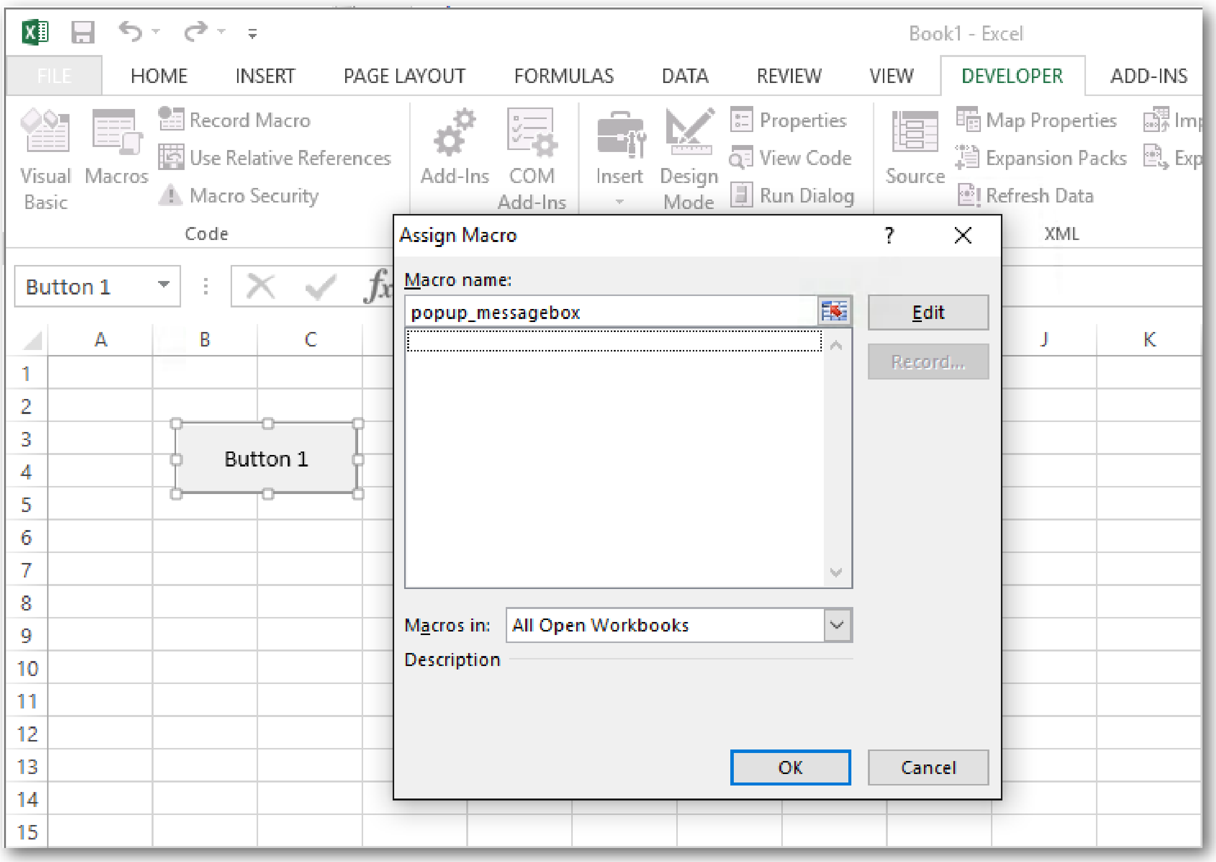
- Backspace
- Break
- CapsLock
- Clear
- Delete
- Down
- End

- Enter
- Escape
- Home
- Insert
- Left
- NumLock
- PgDn
- PgUp
- Right
- ScrollLock
- Tab

3.8.4 Calling Macros From Excel

Macros defined with PyXLL can be called from Excel the same way as any other Excel macros.

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.

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
```

3.9 Real Time Data

3.9.1 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.

3.9.2 Streaming Data From Python

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. Instead of returning a single fixed value, however, they return an instance of an class derived from *RTD*.

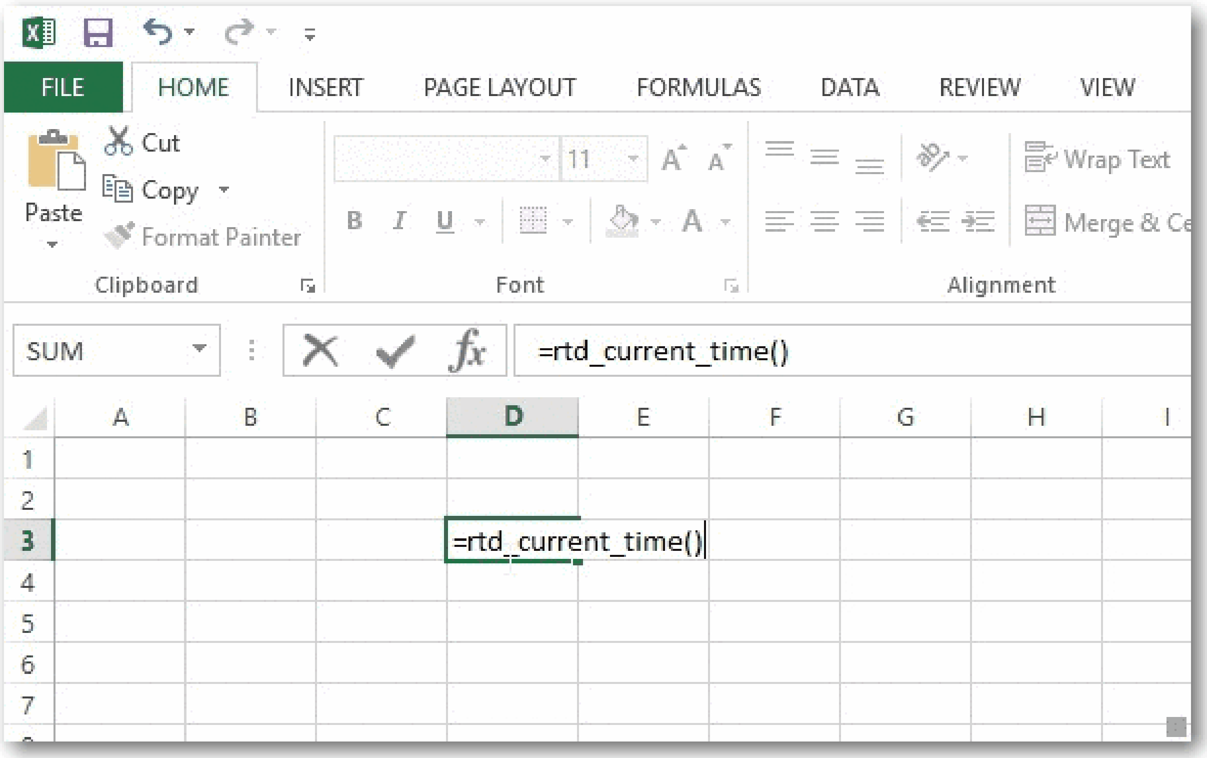
RTD functions have the return type *rtd*.

When a function returns a *RTD* instance PyXLL sets up the real time data subscription in Excel and each time the *value* property of the *RTD* instance is set Excel is notified that new data is ready.

If multiple function calls from different cells return the same instance of an *RTD* class then those cells are subscribed to the same real time data source, so they will all update whenever the *value* property is set.

3.9.3 Example Usage

The following example shows a class derived from *RTD* that periodically updates its value to the current time.



```
=rtd_current_time()
```

It uses a separate thread to set the *value* property, which notifies Excel that new data is ready.

```python
from pyxll import xl_func, RTD
from datetime import datetime
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()
```

3.9.4 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)
```

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*).

3.9.5 Using the asyncio Event Loop

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()
```

3.9.6 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 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
```

3.10 Reloading and Rebinding

3.10.1 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.

3.10.2 How to Reload PyXLL

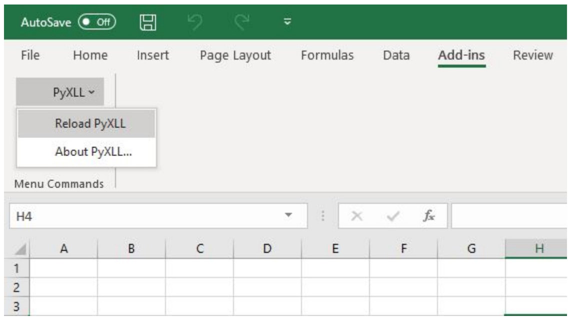
Before you can reload your Python modules with PyXLL, you need to make sure you have `developer_mode` enabled in your pyxll.cfg file.

```
[PYXLL]
developer_mode = 1
```

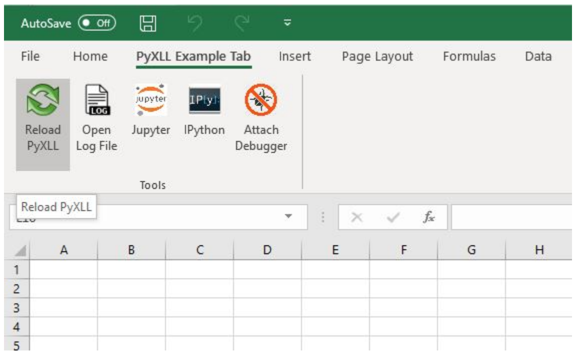
This setting enables reloading and adds the "Reload PyXLL" menu item to Excel. It is enabled by default.

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.

3.10.3 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*.

Listing 3.1: my_excel_addin/__init__.py

```
from . import functions
from . import macros
```

In your pyxll.cfg file, you would only need to list *my_excel_addin*.

```
[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.

3.10.4 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.

To tell PyXLL to rebind you must call 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.

3.11 Error Handling

3.11.1 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.

3.11.2 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:

[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.

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

3.11.3 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.

Listing 3.2: my_error_handler.py

```python
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.

[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`

3.12 Python as a VBA Replacement

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 *Macro Functions*.

3.12.1 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.

3.12.2 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.

```python
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()
```

[1] If you prefer to use `comtypes` instead of `win32com` you can still use *xl_app* by passing `com_package='comtypes'`.

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 *async_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 *async_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')
```

3.12.3 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.

```python
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.

3.12.4 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

```python
from pyxll import xl_app
from win32com.client import constants

xl = xl_app()

cell = xl.Range('A1')
cell2 = cell.End(Direction=constants.xlDown)
```

3.12.5 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.

3.13 Distributing Python Code

For other users to be able to share your Python code and PyXLL based Excel add-in they will need to have:

1. The PyXLL add-in installed and configured
2. Access to your Python code
3. A Python environment with any dependencies installed

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.

So, the question is, how can the Python code be distributed to each user that needs it, and how can updates be deployed?

3.13.1 Sharing Python code on a network drive

This is a simple solution that allows everyone to read the same Python code at all times. The Python code is copied to a location on the network drive and PyXLL is configured to load its Python modules from there.

PyXLL can be configured to load Python code from a network drive by changing the `pythonpath` setting in pyxll.cfg to the network location. For example, if you were to deploy your Python code to a folder `X:\Python\PyXLL\v1` then you would update your pyxll.cfg file as follows:

```
[PYTHON]
pythonpath = X:\Python\PyXLL\v1
```

To deploy changes, rather than updating the code in-place it is better to create another copy of your code in a new folder. You can then update the `pythonpath` in a shared pyxll.cfg file (see *Using a common pyxll.cfg file*) to point to the new folder. This way if you need to rollback to the previous version you can do, and it avoids any problems with some files (e.g. .dll and .pyd files) that may be locked while in use by your users.

It is advisable to make the shared folder read-only to prevent any accidental modifications to the code by your end users. You can pre-compile your .py Python files to .pyc files using python -m compileall before copying your code to the shared folder.

3.13.2 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.

3.13.3 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.

3.13.4 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
```

3.13.5 Building an installer

If you need to deploy to PCs that might not have fast or reliable access to your network, and so accessing a shared drive or using a deployment script is not feasible, building an install can be a solution.

The Python runtime can be bundled with PyXLL into a single standalone installer.

For detailed instructions and an example project for building an MSI installer, see the pyxll-installer project on GitHub.

CHAPTER 4

API Reference

4.1 Function Decorators

These decorators are used to expose Python functions to Excel as worksheet functions, menu functions and macros.

4.1.1 xl_func

xl_func(*signature=None*, *category=PyXLL*, *help_topic=""*, *thread_safe=False*, *macro=False*, *allow_abort=None*, *volatile=False*, *disable_function_wizard_calc=False*, *disable_replace_calc=False*, *name=None*, *auto_resize=False*, *hidden=False*)

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 *Simple 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) 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
- **`disable_replace_calc`** (*boolean*) – Set to True to stop the function being called from Excel's find and replace dialog.
- **`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

Example usage:

```python
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)
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.

4.1.2 xl_menu

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.

4.1.3 xl_macro

xl_macro(*signature=None*, *allow_abort=None*, *name=None*, *shortcut=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.

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 *Macro Functions* for more details about using the xl_macro decorator.

4.1.4 xl_arg_type

xl_arg_type(*name, base_type [, allow_arrays=True] [, macro=None] [, thread_safe=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

4.1.5 xl_return_type

xl_return_type(*name, base_type [, allow_arrays=True] [, macro=None] [, thread_safe=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

4.1.6 xl_arg

xl_arg(*_name [, _type=None] [, **kwargs]*)

Decorator for providing type information for a function argument. This can be used instead of providing a function signature to `xl_func`.

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.
- **kwargs** – Type parameters for parameterized types (eg *NumPy arrays* and *Pandas types*).

4.1.7 xl_return

`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`*.

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*).

4.2 Utility Functions

4.2.1 reload

`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.

4.2.2 rebind

`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()
```

4.2.3 xl_app

xl_app(*com_package=None*)

Gets the Excel Application COM object and returns it as a win32com.Dispach, comtypes.POINTER(IUknown), pythoncom.PyIUnknown or xlwings.App depending on which COM package is being used.

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.

4.2.4 xl_version

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

4.2.5 async_call

async_call(*callable*, **args*, ***kwargs*)

Schedules a callable object (e.g. a function) in Excel's main thread at some point in the (near) future. The callable will be called from a macro context, meaning that it is generally safe to call back into Excel using COM.

This can be useful when calling back into Excel (e.g. updating a cell value) from a worksheet function.

When using this function from a worksheet function care must be taken to ensure that an infinite loop doesn't occur (e.g. if it writes to a cell that's an input to the function, which would cause the function to be called again and again locking up Excel).

Note that and Excel COM objects created in the one thread should not be used in another thread and doing so may cause Excel to crash. Often the same thread will be used to call your worksheet function and run the async callback, but in some cases they may be different. To be safe it is best to always obtain the Excel Application object inside the callback function.

Parameters

- **`callable`** – Callable object to call in the near future.
- **`args`** – Arguments to pass to the callable object.
- **`kwargs`** – Keyword arguments to pass to the callable object.

Example usage:

```
from pyxll import xl_func, xl_app, xlfCaller, async_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

    # kick off the asynchronous call the update function
    pyxll.async_call(update_func)

    return address
```

4.2.6 get_config

`get_config``()`

Returns the PyXLL config as a `ConfigParser.SafeConfigParser` instance

See also *Configuring PyXLL*.

4.2.7 get_dialog_type

`get_dialog_type``()`

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`

4.2.8 get_last_error

`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))


@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)
```

4.2.9 get_type_converter

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** (*string*) – Signature of type to convert from.
- **dest_type** (*string*) – 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))
```

4.2.10 load_image

load_image(*filename*)

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 **filename** (*string*) – Filename of the image file to load. This may be an absolute path or relative to the ribbon XML file.

Returns A COM *IPicture* object (the exact type depends on the *com_package* setting in the *config*.

4.2.11 cached_object_count

cached_object_count()

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 *Using Python Objects Directly*.

4.2.12 get_event_loop

get_event_loop()

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*.

4.3 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.

4.3.1 get_ribbon_xml

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.

4.3.2 set_ribbon_xml

set_ribbon_xml(*xml*, *reload=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.

4.3.3 set_ribbon_tab

set_ribbon_tab(*xml*, *tab_id=None*, *reload=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 on *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).

4.3.4 remove_ribbon_tab

remove_ribbon_tab(*tab_id*, *reload=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.

4.4 Event Handlers

These decorators enable the user to register functions that will be called when certain events occur in the PyXLL addin.

4.4.1 xl_on_open

xl_on_open(*func*)

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 ...
```

4.4.2 xl_on_reload

xl_on_reload(*func*)

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 ...

    # recalcuate all open workbooks
    xlcCalculateNow()
```

4.4.3 xl_on_close

xl_on_close(*func*)

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...")
```

4.4.4 xl_license_notifier

xl_license_notifier(*func*)

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 4 arguments: string name, datetime.date expdate, int days_left, bool is_perpetual.

If the license is perpetual (doesn't expire) expdate will be the end date of the maintenance agreement (when maintenance builds are available until) and days_left will be the days between the PyXLL build date and expdate.

Example usage:

```
from pyxll import xl_license_notifier

@xl_license_notifier
def my_license_notifier(name, expdate, days_left, is_perpetual):
    if days_left < 30:
        ... do something here...
```

4.5 Excel C API Functions

PyXLL exposes certain functions from the Excel C API. These mostly should only be called from a worksheet, menu or macro functions, and some should only be called from macro-sheet equivalent functions[1].

4.5.1 xlfCaller

`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]

4.5.2 xlfSheetId

`xlSheetId``(`*sheet_name*`)`

> **Returns** integer sheet id from a sheet name (e.g. '[Book1.xls]Sheet1')

4.5.3 xlfGetWorkspace

`xlfGetWorkspace``(`*arg_num*`)`

> **Parameters** **`arg_num`** (`int`) – number of 1 to 72 specifying the type of workspace information to return
>
> **Returns** depends on arg_num

[1] A macro sheet equivalent function is a function exposed using `xl_func` with *macro=True*.

4.5.4 xlfGetWorkbook

`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

4.5.5 xlfGetWindow

`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

4.5.6 xlfWindows

`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

4.5.7 xlfVolatile

`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[1]

4.5.8 xlcAlert

`xlcAlert`(*alert*)

Pops up an alert window.

Callable from a macro or menu function only[1]

Parameters **`alert`** (*`string`*) – text to display

4.5.9 xlcCalculation

`xlcCalculation`(*calc_type*)

set the calculation type to automatic or manual.

Callable from a macro or menu function only[1]

Parameters **`calc_type`** (*`int`*) – `xlCalculationAutomatic`

or `xlCalculationSemiAutomatic`

or `xlCalculationManual`

`xlCalculationAutomatic = 1`

`xlCalculationSemiAutomatic = 2`

`xlCalculationManual = 3`

4.5.10 xlcCalculateNow

`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[1]

4.5.11 xlcCalculateDocument

`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[1]

4.5.12 xlAsyncReturn

`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

4.5.13 xlAbort

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.

4.5.14 xlSheetNm

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

4.6 Classes

4.6.1 RTD

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.

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.

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())
```

4.6.2 XLCell

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[1].

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)
```

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 *`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[1]

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[1]

[1] A macro sheet equivalent function is a function exposed using *`xl_func`* with *macro=True*.

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[1]

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[1]

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=None*, *auto_resize=None*, *type_kwargs=None*)
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.
- **type_kwargs** – If setting `type`, type_kwargs can also be set as the options for that type.

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
```

from_range(*range*)
Static method to construct an XLCell from an Excel Range instance. The 'Range' class is part of the Excel Object Model, and can be obtained via *`xl_app`*.

See *Python as a VBA Replacement*.

By getting an XLCell from a Range, values can be set in Excel using PyXLL type converters and object cache.

Parameters **range** – Excel Range object obtained via *`xl_app`*.

Example usage:

```
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[1]

to_range(*self*, *com_wrapper=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

4.6.3 XLRect

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.

4.6.4 XLAsyncHandle

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.

4.6.5 ErrorContext

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.

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.

class 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.

CHAPTER 5

Examples

5.1 UDF Examples

All examples are included in the PyXLL download.

`Plain text version`

```
"""
PyXLL Examples: Worksheet functions

The PyXLL Excel Addin is configured to load one or more
python modules when it's loaded. Functions are exposed
to Excel as worksheet functions by decorators declared in
the pyxll module.

Functions decorated with the xl_func decorator are exposed
to Excel as UDFs (User Defined Functions) and may be called
from cells in Excel.
"""

#
# 1) Basics - exposing functions to Excel
#

#
# xl_func is the main decorator and is used for exposing
# python functions to excel.
#
from pyxll import xl_func

#
# Decorating a function with xl_func is all that's required
# to make it callable in Excel as a worksheet function.
#
@xl_func
```

```
def basic_pyxll_function_1(x, y, z):
    """returns (x * y) ** z """
    return (x * y) ** z


#
# xl_func takes an optional signature of the function to be exposed to excel.
# There are a number of basic types that can be used in
# the function signature. These include:
#   int, float, bool and string
# There are more types that we'll come to later.
#

@xl_func("int x, float y, bool z: float")
def basic_pyxll_function_2(x, y, z):
    """if z return x, else return y"""
    if z:
        # we're returning an integer, but the signature
        # says we're returning a float.
        # PyXLL will convert the integer to a float for us.
        return x
    return y


#
# You can change the category the function appears under in
# Excel by using the optional argument 'category'.
#

@xl_func(category="My new PyXLL Category")
def basic_pyxll_function_3(x):
    """docstrings appear as help text in Excel"""
    return x


#
# 2) The var type
#

#
# A basic type is the var type. This can represent any
# of the basic types, depending on what type is passed to the
# function, or what type is returned.
#
# When no type information is given the var type is used.
#

@xl_func("var x: string")
def var_pyxll_function_1(x):
    """takes an float, bool, string, None or array"""
    # we'll return the type of the object passed to us, pyxll
    # will then convert that to a string when it's returned to
    # excel.
    return type(x)


#
# If var is the return type. PyXll will convert it to the
# most suitable basic type. If it's not a basic type and
# no suitable conversion can be found, it will be converted
# to a string and the string will be returned.
#

@xl_func("bool x: var")
def var_pyxll_function_2(x):
    """if x return string, else a number"""
    if x:
        return "var can be used to return different types"
    return 123.456

#
# 3) Date and time types
#

#
# There are three date and time types: date, time, datetime
#
# Excel represents dates and times as floating point numbers.
# The pyxll datetime types convert the excel number to a
# python datetime.date, datetime.time and datetime.datetime
# object depending on what type you specify in the signature.
#
# dates and times may be returned using their type as the return
# type in the signature, or as the var type.
#

import datetime

@xl_func("date x: string")
def datetime_pyxll_function_1(x):
    """returns a string description of the date"""
    return "type=%s, date=%s" % (type(x), x)


@xl_func("time x: string")
def datetime_pyxll_function_2(x):
    """returns a string description of the time"""
    return "type=%s, time=%s" % (type(x), x)


@xl_func("datetime x: string")
def datetime_pyxll_function_3(x):
    """returns a string description of the datetime"""
    return "type=%s, datetime=%s" % (type(x), x)


@xl_func("datetime[][] x: datetime")
def datetime_pyxll_function_4(x):
    """returns the max datetime"""
    m = datetime.datetime(1900, 1, 1)
    for row in x:
        m = max(m, max(row))
    return m


#
```

```
# 4) xl_cell
#
# The xl_cell type can be used to receive a cell
# object rather than a plain value. The cell object
# has the value, address, formula and note of the
# reference cell passed to the function.
#
# The function must be a macro sheet equivalent function
# in order to access the value, address, formula and note
# properties of the cell.
#

@xl_func("xl_cell cell : string", macro=True)
def xl_cell_example(cell):
    """a cell has a value, address, formula and note"""
    return "[value=%s, address=%s, formula=%s, note=%s]" % (cell.value,
                                                            cell.address,
                                                            cell.formula,
                                                            cell.note)
```

5.2 Pandas Examples

All examples are included in the PyXLL download.

`Plain text version`

```
"""
PyXLL Examples: Pandas

This module contains example functions that show how pandas DataFrames and Series
can be passed to and from Excel to Python functions using PyXLL.

Pandas needs to be installed for this example to work correctly.

See also the included examples.xlsx file.
"""
from pyxll import xl_func


@xl_func(volatile=True)
def pandas_is_installed():
    """returns True if pandas is installed"""
    try:
        import pandas
        return True
    except ImportError:
        return False


@xl_func("int, int: dataframe<index=True>", auto_resize=True)
def random_dataframe(rows, columns):
    """
    Creates a DataFrame of random numbers.

    :param rows: Number of rows to create the DataFrame with.
    :param columns: Number of columns to create the DataFrame with.
    """
    import pandas as pa
    import numpy as np

    data = np.random.rand(rows, columns)
    column_names = [chr(ord('A') + x) for x in range(columns)]
    df = pa.DataFrame(data, columns=column_names)

    return df


@xl_func("dataframe<index=True>, float[], str[], str[]: dataframe<index=True>", auto_
↪resize=True)
def describe_dataframe(df, percentiles=[], include=[], exclude=[]):
    """
    Generates descriptive statistics that summarize the central tendency, dispersion␣
↪and shape of a dataset's
    distribution, excluding NaN values.

    :param df: DataFrame to describe.
    :param percentiles: The percentiles to include in the output. All should fall␣
↪between 0 and 1.
    :param include: dtypes to include.
    :param exclude: dtypes to exclude.
    :return:
    """
    # filter out any blanks
    percentiles = list(filter(None, percentiles))
    include = list(filter(None, include))
    exclude = list(filter(None, exclude))

    return df.describe(percentiles=percentiles or None,
                       include=include or None,
                       exclude=exclude or None)
```

5.3 Cached Objects Examples

All examples are included in the PyXLL download.

`Plain text version`

```
"""
PyXLL Examples: Object Cache Example

This module contains example functions that make use of the PyXLL
object cache.

When Python objects that can't be transformed into a basic type that
Excel can display are returned, PyXLL inserts them into a global
object cache and returns a reference id for the object. When this reference
id is passed to another PyXLL function the object is retrieved from the
cache and passed to the PyXLL function.

PyXLL keeps track of uses of the cached objects and removes items from the
```

```
cache when they are no longer needed.

See also the included examples.xlsx file.
"""
from pyxll import xl_func


class MyTestClass(object):
    """A basic class for testing the cached_object type"""

    def __init__(self, x):
        self.__x = x

    def __str__(self):
        return "%s(%s)" % (self.__class__.__name__, self.__x)


@xl_func("var: object")
def cached_object_return_test(x):
    """returns an instance of MyTestClass"""
    return MyTestClass(x)


@xl_func("object: string")
def cached_object_arg_test(x):
    """takes a MyTestClass instance and returns a string"""
    return str(x)


class MyDataGrid(object):
    """
    A second class for demonstrating cached_object types.
    This class is constructed with a grid of data and has
    some basic methods which are also exposed as worksheet
    functions.
    """

    def __init__(self, grid):
        self.__grid = grid

    def sum(self):
        """returns the sum of the numbers in the grid"""
        total = 0
        for row in self.__grid:
            total += sum(row)
        return total

    def __len__(self):
        total = 0
        for row in self.__grid:
            total += len(row)
        return total

    def __str__(self):
        return "%s(%d values)" % (self.__class__.__name__, len(self))


@xl_func("float[][]: object")
```

```
def make_datagrid(x):
    """returns a MyDataGrid object"""
    return MyDataGrid(x)


@xl_func("object: int")
def datagrid_len(x):
    """returns the length of a MyDataGrid object"""
    return len(x)


@xl_func("object: float")
def datagrid_sum(x):
    """returns the sum of a MyDataGrid object"""
    return x.sum()


@xl_func("object: string")
def datagrid_str(x):
    """returns the string representation of a MyDataGrid object"""
    return str(x)
```

5.4 Custom Type Examples

All examples are included in the PyXLL download.

`Plain text version`

```
"""
PyXLL Examples: Custom types

Worksheet functions can use a number of standard types
as shown in the worksheetfuncs example.

It's also possible to define custom types that
can be used in the PyXLL function signatures
as shown by these examples.

For a more complicated custom type example see the
object cache example.
"""

#
# xl_arg_type and xl_return type are decorators that can
# be used to declare types that our excel functions
# can use in addition to the standard types
#
from pyxll import xl_func, xl_arg_type, xl_return_type

#
# 1) Custom types
#

#
# All variables are passed to and from excel as the basic types,
# but it's possible to register conversion functions that will
# convert those basic types to whatever types you like before
# they reach your function, (or after you function returns them
# in the case of returned values).
#

#
# CustomType1 is a very simple class used to demonstrate
# custom types.
#
class CustomType1:

    def __init__(self, name):
        self.name = name

    def greeting(self):
        return "Hello, my name is %s" % self.name

#
# To use CustomType1 as an argument to a pyxll function you have to
# register a function to convert from a basic type to our custom type.
#
# xl_arg_type takes two arguments, the new custom type name, and the
# base type.
#

@xl_arg_type("custom1", "string")
def string_to_custom1(name):
    return CustomType1(name)

#
# now the type 'custom1' can be used as an argument type
# in a function signature.
#

@xl_func("custom1 x: string")
def customtype_pyxll_function_1(x):
    """returns x.greeting()"""
    return x.greeting()

#
# To use CustomType1 as a return type for a pyxll function you have
# to register a function to convert from the custom type to a basic type.
#
# xl_return_type takes two arguments, the new custom type name, and
# the base type.
#

@xl_return_type("custom1", "string")
def custom1_to_string(x):
    return x.name

#
# now the type 'custom1' can be used as the return type.
#

@xl_func("custom1 x: custom1")
def customtype_pyxll_function_2(x):
    """check the type and return the same object"""
    assert isinstance(x, CustomType1), "expected an CustomType1 object"""
    return x

#
# CustomType2 is another example that caches its instances
# so they can be referred to from excel functions.
#

class CustomType2:

    __instances__ = {}

    def __init__(self, name, value):
        self.value = value
        self.id = "%s-%d" % (name, id(self))

        # overwrite any existing instance with self
        self.__instances__[name] = self

    def getValue(self):
        return self.value

    @classmethod
    def getInstance(cls, id):
        name, unused = id.split("-")
        return cls.__instances__[name]

    def getId(self):
        return self.id


@xl_arg_type("custom2", "string")
def string_to_custom2(x):
    return CustomType2.getInstance(x)


@xl_return_type("custom2", "string")
def custom2_to_string(x):
    return x.getId()


@xl_func("string name, float value: custom2")
def customtype_pyxll_function_3(name, value):
    """returns a new CustomType2 object"""
    return CustomType2(name, value)


@xl_func("custom2 x: float")
def customtype_pyxll_function_4(x):
    """returns x.getValue()"""
    return x.getValue()

#
# custom types may be base types of other custom types, as
# long as the ultimate base type is a basic type.
#
# This means you can chain conversion functions together.
```

```
#

class CustomType3:

    def __init__(self, custom2):
        self.custom2 = custom2

    def getValue(self):
        return self.custom2.getValue() * 2


@xl_arg_type("custom3", "custom2")
def custom2_to_custom3(x):
    return CustomType3(x)


@xl_return_type("custom3", "custom2")
def custom3_to_custom2(x):
    return x.custom2


#
# when converting from an excel cell to a CustomType3 object,
# the string will first be used to get a CustomType2 object
# via the registed function string_to_custom2, and then
# custom2_to_custom3 will be called to get the final
# CustomType3 object.
#

@xl_func("custom3 x: float")
def customtype_pyxll_function_5(x):
    """return x.getValue()"""
    return x.getValue()
```

5.5 Menu Examples

All examples are included in the PyXLL download.

`Plain text version`

```
"""
PyXLL Examples: Menus

The PyXLL Excel Addin is configured to load one or more
python modules when it's loaded.

Menus can be added to Excel via the pyxll xl_menu decorator.
"""
import traceback
import logging
_log = logging.getLogger(__name__)

# the webbrowser module is used in an example to open the log file
try:
    import webbrowser
except ImportError:
    _log.warning("*** webbrowser could not be imported                ***")
    _log.warning("*** the menu examples will not work correctly      ***")

import os

#
# 1) Basics - adding a menu items to Excel
#

#
# xl_menu is the decorator used for addin menus to Excel.
#
from pyxll import xl_menu, get_config, xl_app, xl_version, get_last_error, xlcAlert

#
# The only required argument is the menu item name.
# The example below will add a new menu item to the
# addin's default menu.
#

@xl_menu("Example Menu Item 1")
def on_example_menu_item_1():
    xlcAlert("Hello from PyXLL")

#
# menu items are normally sorted alphabetically, but the order
# keyword can be used to influence the ordering of the items
# in a menu.
#
# The default value for all sort keyword arguments is 0, so positive
# values will result in the item appearing further down the list
# and negative numbers result in the item appearing further up.
#

@xl_menu("Another example menu item", order=1)
def on_example_menu_item_2():
    xlcAlert("Hello again from PyXLL")

#
# It's possible to add items to menus other than the default menu.
# The example below creates a new menu called 'My new menu' with
# one item 'Click me' in it.
#
# The menu_order keyword is optional, but may be used to influence
# the order that the custom menus appear in.
#

@xl_menu("Click me", menu="PyXLL example menu", menu_order=1)
def on_example_menu_item_3():
    xlcAlert("Adding multiple menus is easy")

#
# 2) Sub-menus
#

# it's possible to add sub-menus just by using the sub_menu
# keyword argument. The example below adds a new sub menu
# 'Sub Menu' to the default menu.
```

```
#
# The order keyword argument affects where the sub menu will
# appear in the parent menu, and the sub_order keyword argument
# affects where the item will appear in the sub menu.
#

@xl_menu("Click me", sub_menu="More Examples", order=2)
def on_example_submenu_item_1():
    xlcAlert("Sub-menus can be created easily with PyXLL")

#
# When using Excel 2007 and onwards the Excel functions accept unicode strings
#
@xl_menu("Unicode Test", sub_menu="More Examples")
def on_unicode_test():
    xlcAlert(u"\u01d9ni\u0186\u020dde")

#
# A simple menu item to show how to get the PyXLL config
# object and open the log file.
#
@xl_menu("Open log file", order=3)
def on_open_logfile():
    # the PyXLL config is accessed as a ConfigParser.ConfigParser object
    config = get_config()
    if config.has_option("LOG", "path") and config.has_option("LOG", "file"):
        path = os.path.join(config.get("LOG", "path"), config.get("LOG", "file"))
        webbrowser.open("file://%s" % path)

#
# If a cell returns an error it is written to the log file
# but can also be retrieved using 'get_last_error'.
# This menu item displays the last error captured for the
# current active cell.
#
@xl_menu("Show last error")
def show_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)
```

5.6 Macros and Excel Scripting

All examples are included in the PyXLL download.

`Plain text version`

```
"""
PyXLL Examples: Automation

PyXLL worksheet and menu functions can call back into Excel
using the Excel COM API*.

In addition to the COM API there are a few Excel functions
exposed via PyXLL that allow you to query information about
the current state of Excel without using COM.

Excel uses different security policies for different types
of functions that are registered with it. Depending on
the type of function, you may or may not be able to make
some calls to Excel.

Menu functions and macros are registered as 'commands'.
Commands are free to call back into Excel and make changes to
documents. These are equivalent to the VBA Sub routines.

Worksheet functions are registered as 'functions'. These
are limited in what they can do. You will be able to
call back into Excel to read values, but not change
anything. Most of the Excel functions exposed via PyXLL
will not work in worksheet functions. These are equivalent
to VBA Functions.

There is a third type of function - macro-sheet equivalent
functions. These are worksheet functions that are allowed to
do most things a macro function (command) would be allowed
to do. These shouldn't be used lightly as they may break
the calculation dependencies between cells if not
used carefully.

* Excel COM support was added in Office 2000. If you are
  using an earlier version these COM examples won't work.
"""

import pyxll
from pyxll import xl_menu, xl_func, xl_macro

import logging
_log = logging.getLogger(__name__)

#
# Getting the Excel COM object
#
# PyXLL has a function 'xl_app'. This returns the Excel application
# instance either as a win32com.client.Dispatch object or a
# comtypes object (which com package is used may be set in the
# config file). The default is to use win32com.
#
# It is better to use this than
# win32com.client.Dispatch("Excel.Application")
# as it will always be the correct handle - ie the handle
# to the correct instance of Excel.
#
# For more information on win32com see the pywin32 project
# on sourceforge.
```

```
#
# The Excel object model is the same from COM as from VBA
# so usually it's straightforward to write something
# in python if you know how to do it in VBA.
#
# For more information about the Excel object model
# see MSDN or the object browser in the Excel VBA editor.
#
from pyxll import xl_app

#
# A simple example of a menu function that modifies
# the contents of the selected range.
#

@xl_menu("win32com test", sub_menu="More Examples")
def win32com_menu_test():
    # get the current selected range and set some text
    selection = xl_app().Selection
    selection.Value = "Hello!"
    pyxll.xlcAlert("Some text has been written to the current cell")

#
# Macros can also be used to call back into Excel when
# a control is activated.
#
# These work in the same way as VBA macros, you just assign
# them to the control in Excel by name.
#

@xl_macro
def button_example():
    xl = xl_app()
    range = xl.Range("button_output")
    range.Value = range.Value + 1


@xl_macro
def checkbox_example():
    xl = xl_app()
    check_box = xl.ActiveSheet.CheckBoxes(xl.Caller)
    if check_box.Value == 1:
        xl.Range("checkbox_output").Value = "CHECKED"
    else:
        xl.Range("checkbox_output").Value = "Click the check box"


@xl_macro
def scrollbar_example():
    xl = xl_app()
    caller = xl.Caller
    scrollbar = xl.ActiveSheet.ScrollBars(xl.Caller)
    xl.Range("scrollbar_output").Value = scrollbar.Value


#
# Worksheet functions can also call back into Excel.
#
```

```
# The function 'async_call' must be used to do the
# actual work of calling back into Excel after Excel has
# finished calculating. Otherwise Excel may lock waiting for
# the function to complete before allowing the COM object
# to modify the sheet, which will cause a dead-lock.
#
# To be able to call xlfCaller from the worksheet function,
# the function must be declared as a macro sheet equivalent
# function by passing macro=True to xl_func.
#
# If your function modifies the Excel worksheet it may trigger
# a recalculation, and so you have to take care not to
# cause an infinite loop that will hang Excel.
#
# Accessing the 'address' property of the XLCell returned
# by xlfCaller requires this function to be a macro sheet
# equivalent function.
#

@xl_func(macro=True)
def automation_example(rows, cols, value):
    """copies value to a range of rows x cols below the calling cell"""

    # Get the address of the calling cell using xlfCaller
    caller = pyxll.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():
        # Get the Excel.Application COM object
        xl  = xl_app()

        # Get an Excel.Range object from the XLCell instance
        range = caller.to_range(com_package="win32com")

        # get the cell below and expand it to rows x cols
        range = xl.Range(range.Resize(2, 1), range.Resize(rows+1, cols))

        # and set the range's value
        range.Value = value

    # kick off the asynchronous call the update function
    pyxll.async_call(update_func)

    return address
```

5.7 Event Handler Examples

All examples are included in the PyXLL download.

Plain text version

```
"""
PyXLL Examples: Callbacks
```

```
The PyXLL Excel Addin is configured to load one or more
python modules when it's loaded.

Moldules can register callbacks with PyXLL that will be
called at various times to inform the user code of
certain events.
"""

from pyxll import xl_on_open,              \
                  xl_on_reload,            \
                  xl_on_close,             \
                  xl_license_notifier,     \
                  xlcAlert,                \
                  xlcCalculateNow

import logging
_log = logging.getLogger(__name__)

@xl_on_open
def on_open(import_info):
    """
    on_open is registered to be called by PyXLL when the addin
    is opened via the xl_on_open decorator.
    This happens each time Excel starts with PyXLL installed.
    """
    # Check to see which modules didn't import correctly.
    for modulename, module, exc_info in import_info:
        if module is None:
            exc_type, exc_value, exc_traceback = exc_info
            _log.error("Error loading '%s' : %s" % (modulename, exc_value))


@xl_on_reload
def on_reload(import_info):
    """
    on_reload is registered to be called by PyXLL whenever a
    reload occurs via the xl_on_reload decorator.
    """
    # Check to see if any modules didn't import correctly.
    errors = 0
    for modulename, module, exc_info in import_info:
        if module is None:
            exc_type, exc_value, exc_traceback = exc_info
            _log.error("Error loading '%s' : %s" % (modulename, exc_value))
            errors += 1

    # Report if everything reloaded OK.
    # If there are errors they will be dealt with by the error_handler.
    if errors == 0:
        xlcAlert("Everything reloaded OK!\n\n(Message from callbacks.py example)")

    # Recalculate all open workbooks.
    xlcCalculateNow()


@xl_on_close
def on_close():
    """
    on_close will get called as Excel is about to close.

    This is a good time to clean up any globals and stop
    any background threads so that the python interpretter
    can be closed down cleanly.

    The user may cancel Excel closing after this has been
    called, so your code should make sure that anything
    that's been cleaned up here will get recreated again
    if it's needed.
    """
    _log.info("callbacks.on_close: PyXLL is closing")


@xl_license_notifier
def license_notifier(name, expdate, days_left, is_perpetual):
    """
    license_notifier will be called when PyXLL is starting up, after
    it has read the config and verified the license.

    If there is no license name will be None and days_left will be less than 0.
    """
    if days_left >= 0 or is_perpetual:
        _log.info("callbacks.license_notifier: "
                  "This copy of PyXLL is licensed to %s" % name)
        if not is_perpetual:
            _log.info("callbacks.license_notifier: "
                      "%d days left before the license expires (%s)" % (days_left,␣
↪expdate))
    elif expdate is not None:
        _log.info("callbacks.license_notifier: License key expired on %s" % expdate)
    else:
        _log.info("callbacks.license_notifier: Invalid license key")
```

Index