PyXLL User Guide

Release 3.3.2

PyXLL Ltd.

Jan 03, 2018

CONTENTS

1 PyXLL Documentation **1**

1.1 What's new in PyXLL 3 . . . 1
- 1.1.1 Automatic Array Resizing . . . 2
- 1.1.2 Keyboard Shortcuts . . . 2
- 1.1.3 Customizing the Ribbon . . . 2
- 1.1.4 RTD (Real Time Data) Functions . . . 2
- 1.1.5 Function Signatures and Type Annotation . . . 2
- 1.1.6 Default Keyword Arguments . . . 2
- 1.1.7 Easier Setup for Anaconda and Virtualenvs . . . 3
- 1.1.8 Deep Reloading . . . 3
- 1.1.9 Error Caching . . . 3
- 1.1.10 Python Functions for Reload and Rebind . . . 3
- 1.1.11 Better win32com and comtypes Support . . . 4
- 1.1.12 IntelliSense . . . 4

1.2 Getting Started . . . 4
- 1.2.1 Installing with Enthought Canopy . . . 4
- 1.2.2 Installing with other Python Distributions . . . 5
- 1.2.3 Calling a Python Function in Excel . . . 6
- 1.2.4 Next Steps . . . 8

1.3 User Guide . . . 8
- 1.3.1 Configuration . . . 8
- 1.3.2 Worksheet Functions . . . 15
- 1.3.3 Menu Functions . . . 26
- 1.3.4 Customizing the Ribbon . . . 29
- 1.3.5 Macro Functions . . . 33
- 1.3.6 Real Time Data . . . 35

1.4 API Reference . . . 38
- 1.4.1 Function Decorators . . . 38
- 1.4.2 Utility Functions . . . 42
- 1.4.3 Ribbon Functions . . . 46
- 1.4.4 Event Handlers . . . 48
- 1.4.5 Excel C API Functions . . . 50
- 1.4.6 Classes . . . 53

1.5 Examples . . . 55
- 1.5.1 Worksheet Functions . . . 55
- 1.5.2 Custom Types . . . 58
- 1.5.3 Menu Functions . . . 62
- 1.5.4 Macros and Excel Scripting . . . 64
- 1.5.5 Event Handlers . . . 67
- 1.5.6 Object Cache . . . 69

1.5.7 Developer Tools 78

Index **84**

CHAPTER

ONE

PYXLL DOCUMENTATION

Looking for the PyXLL 2.x docs?

PyXLL 2 Documentation

This documentation is for versions of PyXLL 3.0 and greater.

If you don't find what you're looking for, or need any extra help please contact us.

1.1 What's new in PyXLL 3

PyXLL 3 introduces a number of new features that make it even easier to make rich, interactive Excel spreadsheets in Python.

Internally a lot has changed to bring you these new features and ensure that PyXLL retains the high quality and reliability that you're used to from previous versions.

- *Automatic Array Resizing*
- *Keyboard Shortcuts*
- *Customizing the Ribbon*
- *RTD (Real Time Data) Functions*
- *Function Signatures and Type Annotation*
- *Default Keyword Arguments*
- *Easier Setup for Anaconda and Virtualenvs*
- *Deep Reloading*
- *Error Caching*
- *Python Functions for Reload and Rebind*
- *Better win32com and comtypes Support*
- *IntelliSense*

1.1.1 Automatic Array Resizing[1]

When returning an array PyXLL can automatically resize the range of the array formula in Excel so it expands or contracts to match the size of the array being returned.

Array functions can be made to automatically resize by setting the keyword argument *auto_resize=True* to `xl_func`, or it can be enabled for all array functions in the config file by setting

```
[PYXLL]
auto_resize_arrays = 1
```

1.1.2 Keyboard Shortcuts[1]

Macros and menus written using `xl_macro` and `xl_menu` can now be assigned keyboard shortcuts using the *shortcut* keyword argument, or set in the *config*.

1.1.3 Customizing the Ribbon

Ever wanted to write an addin that uses the Excel ribbon interface? Previously the only way to do this was to write a COM addin, which requires a lot of knowledge, skill and perseverance! Now you can do it with PyXLL by defining your ribbon as an XML document and adding it to your PyXLL config. All the callbacks between Excel and your Python code are handled for you.

See *Customizing the Ribbon* for more detailed information or try the example included in the download.

1.1.4 RTD (Real Time Data) Functions

PyXLL can stream live data into your spreadsheet without you having to write any extra services or register any COM controls. Any Python function exposed to Excel through PyXLL can return a new `RTD` type that acts as a ticking data source; Excel updates whenever the returned `RTD` publishes new data.

See *Real Time Data* for more detailed information or try the example included in the download.

1.1.5 Function Signatures and Type Annotation

`xl_func` and `xl_macro` need to know the argument and return types to be able to tell Excel how they should be called. In previous versions that was always done by passing a 'signature' string to these decorators.

Now in PyXLL 3 the signature is entirely optional. If a signature is not supplied PyXLL will inspect the function and determine the signature for you.

If you use Python type annotations when declaring the function, PyXLL will use those when determining the function signature. Otherwise all arguments and the return type will be assumed to be *var*.

1.1.6 Default Keyword Arguments

Python functions with default keyword arguments now preserve their default value when called from Excel with missing arguments. This means that a function like the one below when called from Excel with *b* or *c* missing will be invoked with the correct default values for *b* and *c*.

[1] New in PyXLL 3.1

```
@xl_func
def func_with_kwargs(a, b=1, c=2):
    return a + b + c
```

1.1.7 Easier Setup for Anaconda and Virtualenvs

Many problems with installing PyXLL come down to the fact that Python isn't installed as the default Python installation on the PC. This means that PyXLL can't find the Python dll or the Python packages as their locations are not set globally.

This affects any distribution that doesn't set itself up as the default Python install (most notably Anaconda, PortablePython and any virtualenv).

There were ways to get around this and make it work, but it always should have been easier. Now it is!

To set which Python to use with PyXLL now all you have to do is set it up in your pyxll.cfg file

```
[PYTHON]
executable = <your chosen python.exe here>
```

When this is set PyXLL will try to find the right Python DLL to load and try to locate where the Python libraries are installed. If it still can't find either of those you can tell it by setting *dll* and *pythonhome* in the same section of the config.

1.1.8 Deep Reloading

If you've used PyXLL for a while you will have noticed that when you reload PyXLL only the modules listed in your pyxll.cfg file get reloaded. If you are working on a project that has multiple modules and not all of them are added to the config those won't get reloaded, even if modules that are listed in the config file import them.

PyXLL can now track all the imports made by each module listed in the config file, and when you reload PyXLL all of those modules will be reloaded in the right order.

This feature is enabled in the config file by setting

```
[PYXLL]
deep_reload = 1
```

1.1.9 Error Caching

Sometimes it's not convenient to have to pick through the log file to determine why a particular cell is failing to calculate.

The new function *get_last_error* takes an *XLCell* or a COM Range and returns the last exception (and traceback) to have occurred in that cell.

This can be used in menu functions or other worksheet functions to give end users better feedback about any errors in the worksheet.

1.1.10 Python Functions for Reload and Rebind

PyXLL can now be reloaded or it can rebind its Excel functions using the new Python functions *reload* and *rebind*.

1.1.11 Better win32com and comtypes Support

PyXLL has always had some integration with the pythoncom module, but it required some user code to make it really useful. It didn't have any direct integration with the higher level win32com package or the comtypes package.

The new function *xl_app* returns the current Excel Application instance either as a pythoncom PyIDispatch instance, a win32com.client.Dispatch instance, a wrapped comtypes POINTER(IUnknown) instance or an xlwings.App instance[1].

You may specify which COM library you want to use with PyXLL in the pyxll.cfg file

```
[PYXLL]
com_package = <win32com (default), comtypes or pythoncom>
```

1.1.12 IntelliSense[1]

PyXLL integrates with the Excel-DNA IntelliSense Excel Addin to provide function completion and argument help as you enter formulas in Excel.

By adding the Excel-DNA IntelliSense addin to Excel (as well as PyXLL) your custom Python functions will automatically be recognized and it will give you IntelliSense for your functions.

The Excel-DNA IntelliSense addin can be downloaded from https://github.com/Excel-DNA/IntelliSense/releases.

Note: The Excel-DNA IntelliSense addin is currently in beta.

1.2 Getting Started

- *Installing with Enthought Canopy*
- *Installing with other Python Distributions*
- *Calling a Python Function in Excel*
- *Next Steps*

1.2.1 Installing with Enthought Canopy

Have Canopy 1.6 or lower?

You must upgrade to Canopy v1.7.4 or 2.x for PyXLL to install correctly. You can upgrade Canopy under the Help menu.

Enthought Canopy provides an integrated PyXLL *egg* that you can install via the Package Manager. You must upgrade to Canopy v1.7.4 or 2.x for PyXLL to install correctly. You can upgrade Canopy under the *Help menu*.

1. Launch the *Package Manager* from the *Canopy Welcome Screen*.
2. Select *Available Packages* and search for *pyxll*.
3. Select the *pyxll* package from the list and click the *Install* button to the right to have Canopy download and install PyXLL for you.

4. Launch a *Canopy Command Prompt* from the *Start -> Enthought Canopy* menu.
5. Launch Canopy and open the *pyxll.cfg* file by entering:

```
edit_pyxll_configuration
```

This is where you can update your configuration with any Python modules you want to load, and any other settings (see *Configuration*).

Once configured, start Excel and the PyXLL addin will be loaded.

1.2.2 Installing with other Python Distributions

1. **Download the standalone PyXLL zip file from here.** Make sure you download the correct version depending on the versions of Python and Excel you want to use.
2. **Unpack the zipfile.** PyXLL can be used with any Python distribution. Depending on how you have Python installed on your system you may need to configure PyXLL so it knows where to find your Python installation.

   PyXLL is packaged as a zip file. Simply unpack the zip file where you want PyXLL to be installed, there is no installer to run.
3. **Edit the config file** Once you've unzipped the PyXLL download the next thing to do is to edit the *pyxll.cfg* file. Any text editor will do for this. The default configuration may be fine for you while you're getting started, and you can come back to it later if you want to make any changes.

   If your Python installation is not configured to be the default Python installation (this is common with Anaconda and PortablePython distributions) you will need set the following in your config file

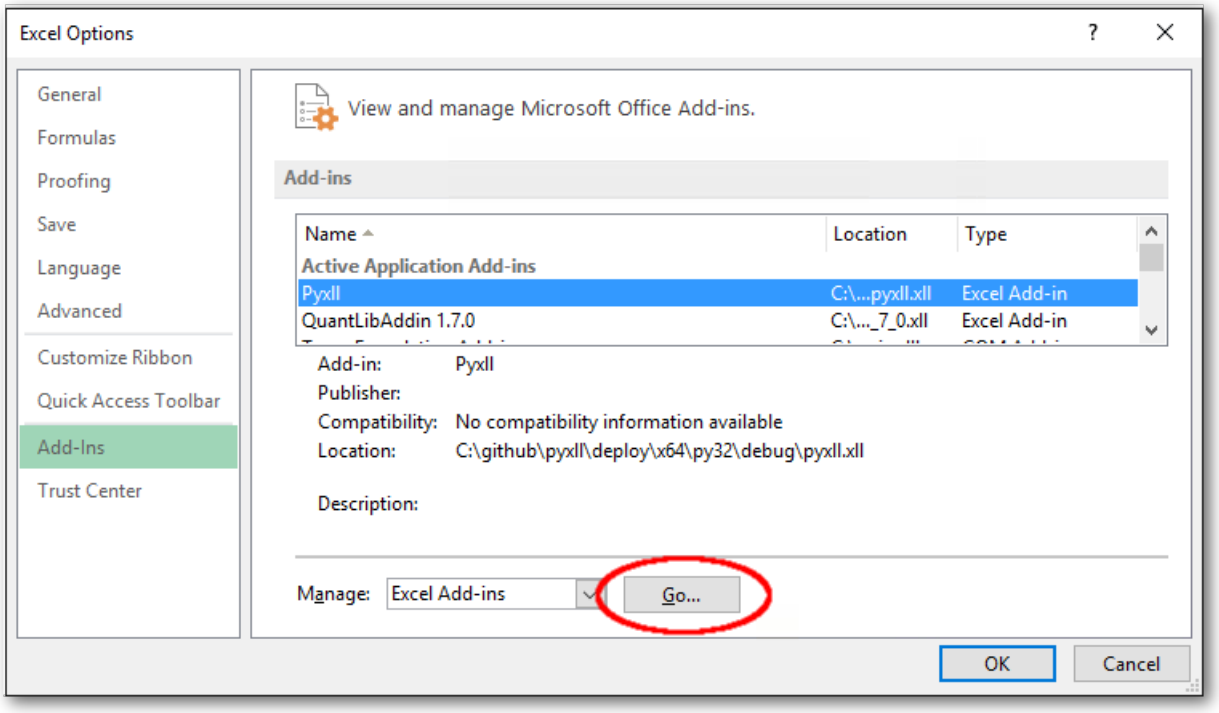
```
[PYTHON]
executable = <path to your python.exe>
```

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 also set the dll location in the config to tell PyXLL where to find it.

4. **Install the addin** Once you're happy with the configuration you can install the addin in Excel by following the instructions below.
   - **Excel 2010 - 2016** 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.



5. ***Optional*** **Install the PyXLL stubs package** If you are using a Python IDE that provides autocompletion or code checking, because the pyxll module isn't installed, you may find it complains and won't be able to provide any autocompletion for functions imported from the pyxll module.

   In the downloaded zip file you will find a *.whl* file. The exact filename varies depending on the version of PyXLL you've downloaded. That's a Python Wheel containing a dummy pyxll module that you can use for the purposes of keeping your IDE from complaining, and for writing code that depends on the pyxll module that you want to use 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:

   ```
   > pip install {pyxll wheel filename}
   ```

   The real pyxll module is compiled into the pyxll.xll addin.

   If you are using a version of Python that isn't supported by pip all you need to do is to unzip the wheel file into your Python site-packages folder (the wheel file is simply a zip file with a different file extension).

1.2.3 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 your new module here>

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

Reload PyXLL by going to the *Addins* menu in Excel and selecting *PyXLL -> Reload*.

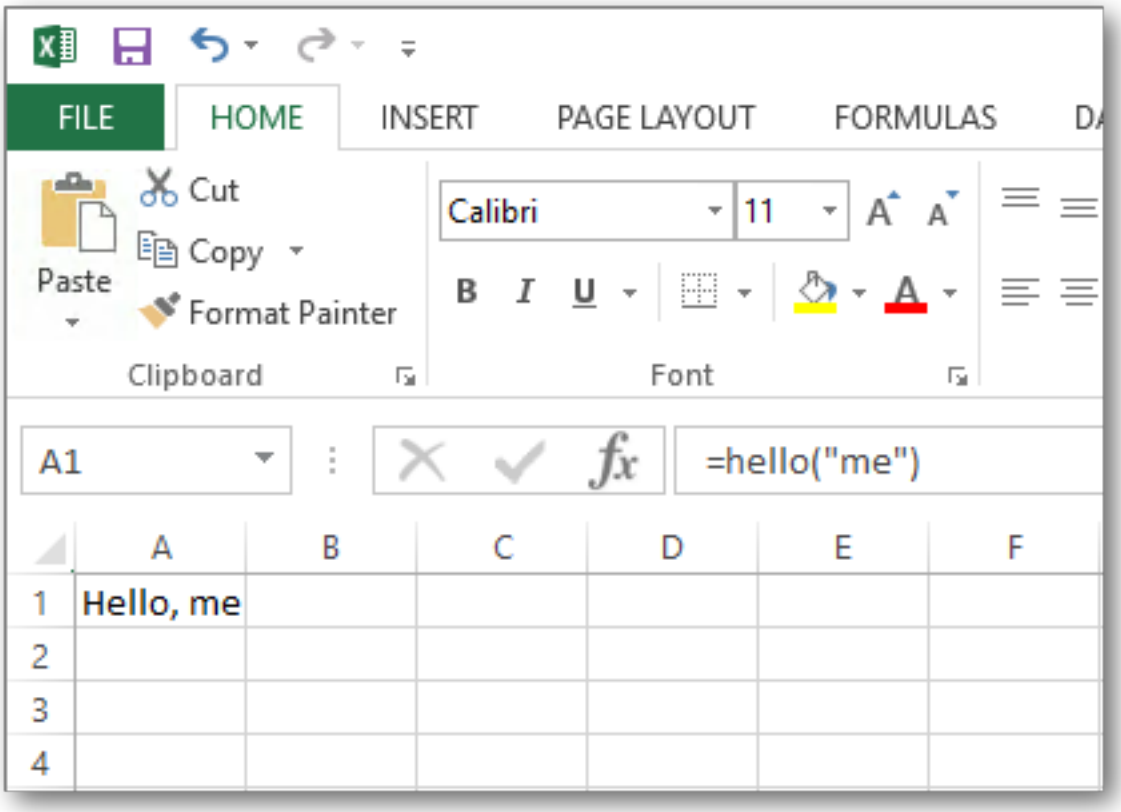
Now in a worksheet you can type a formula using your new Python function.:

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

Using PyCharm or Eclipse?

You can interactively debug Python code running in PyXLL with Eclipse or PyCharm by attaching the PyDev debugger. See this example 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 and necessary changes to your code and reload PyXLL again.



1.2.4 Next Steps

The *user guide* has comprehensive coverage of how to use all the features of PyXLL and a quick look through there will quickly bring you up to speed on how to do most things with PyXLL.

Many of PyXLL's features are 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 user guide please contact us and we will do our best to help.

1.3 User Guide

1.3.1 Configuration

Getting at the config from your code

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.

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

If you wish to load the config file from an alternative location, that can be configured 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 absolue or relative paths. Any relative paths should be relative to the config file.

Config values may contain environment variable substitutions. To substitute an environment variable into your value use *%(envvar_name)s*, e.g.

```
[LOG]
path = %(TEMP)s
file = %(LOG_FILE)s
```

Python Settings

```
[PYTHON]
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
```

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

If you don't specify anything in the Python section then your system default Python settings will be used. Depending on how Python was installed on your system this may be fine, but sometimes you may want to specify different options 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** Usually setting the Python executable will be enough so that PyXLL can find the dll without further help, but if your installation is non-standard or you need to tell PyXLL to use a specific dll for any reason then this setting may be used for that.

  ```
  [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 will usually be determined from with the system default Python installation or by looking for them relative to the Python executable.

  If for any reason the standard libraries are not installed relative to the chosen Python 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.

  ```
  [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.

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
error_cache_size = maximum number of exceptions to cache for failed function calls
external_config = paths of additional config files to load
name = name of the addin visible in Excel
allow_abort = 1 or 0 to set the default value for the allow_abort kwarg
auto_resize_arrays = 1 or 0 to enable automatic resizing of all array functions
quiet = 1 or 0 to disable all start up messages
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_disable = 1 or 0 to disable all deep reloading functionality
disable_com_addin = 1 or 0 to disable the COM addin component of PyXLL
```

- **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 locations of these modules must be on the *pythonpath*, which can be set in the *[PYTHON]* config section.

- **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.

- **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.

- **external_config** This setting may be used to reference another config file (or files) located elsewhere.

  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.

- **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'.

- **allow_abort** 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.

- **auto_resize_arrays** 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. If unset the default is 0.

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

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

- **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 excluded from being reloaded.

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

- **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 it can be added to this list to be ignored.

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

  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_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.

  Changing this setting requires Excel to be restarted.

- **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.

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 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*.

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)
```

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.

Environment Variables

For some python modules it can be helpful to set some environment variables before they are imported. Usually this would be done in the environment running the python script, but in Excel it's more complicated as it would require either changing the global environment variables on each PC, or using a batch script to launch Excel.

For this reason, it's possible to set environment variables in the *[ENVIRONMENT]* section of the config file.

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

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

Menu Ordering

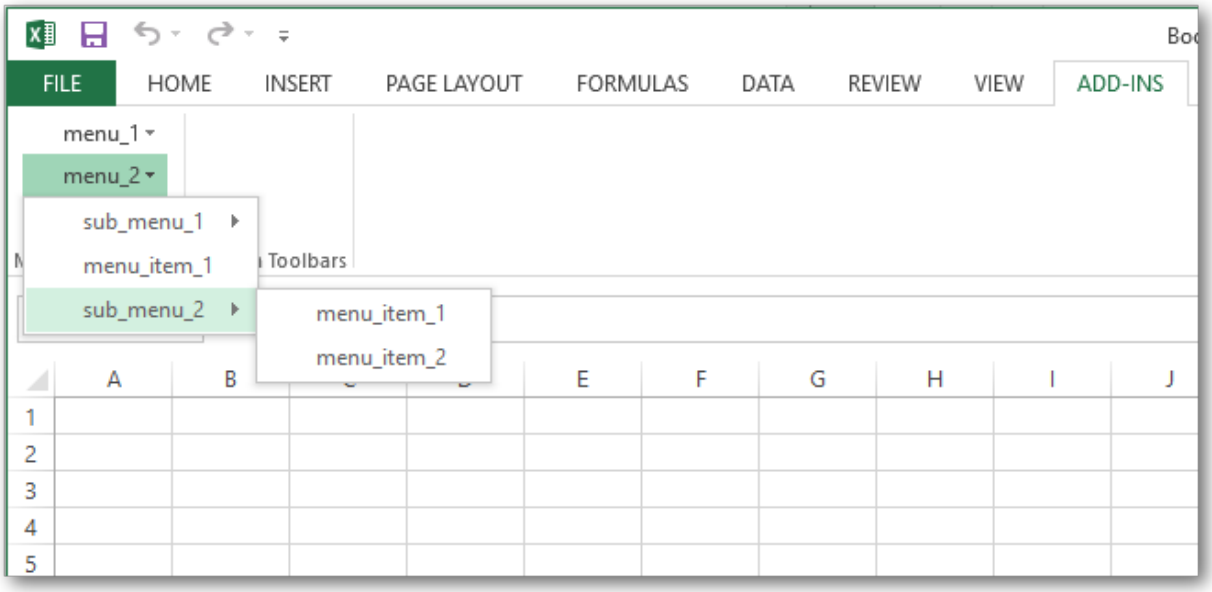
Menu items added via the *xl_menu* decorator can specify what order they should appear in the menus. This can be also be set, or overridden, in the config file.

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:



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.

1.3.2 Worksheet Functions

Tip: You don't have to restart Excel

Use the 'Reload' menu item under the PyXLL menu to reload your Python code without restarting Excel

Calling functions written in Python using PyXLL in Excel is exactly the same as calling any other Excel function written in VBA or as part of another Excel addin. They are called from formulas in an Excel worksheet in the same way, and appear in Excel's function wizzard.

Here's a simple example of a worksheet function written in Python

```
from pyxll import xl_func

@xl_func
```

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

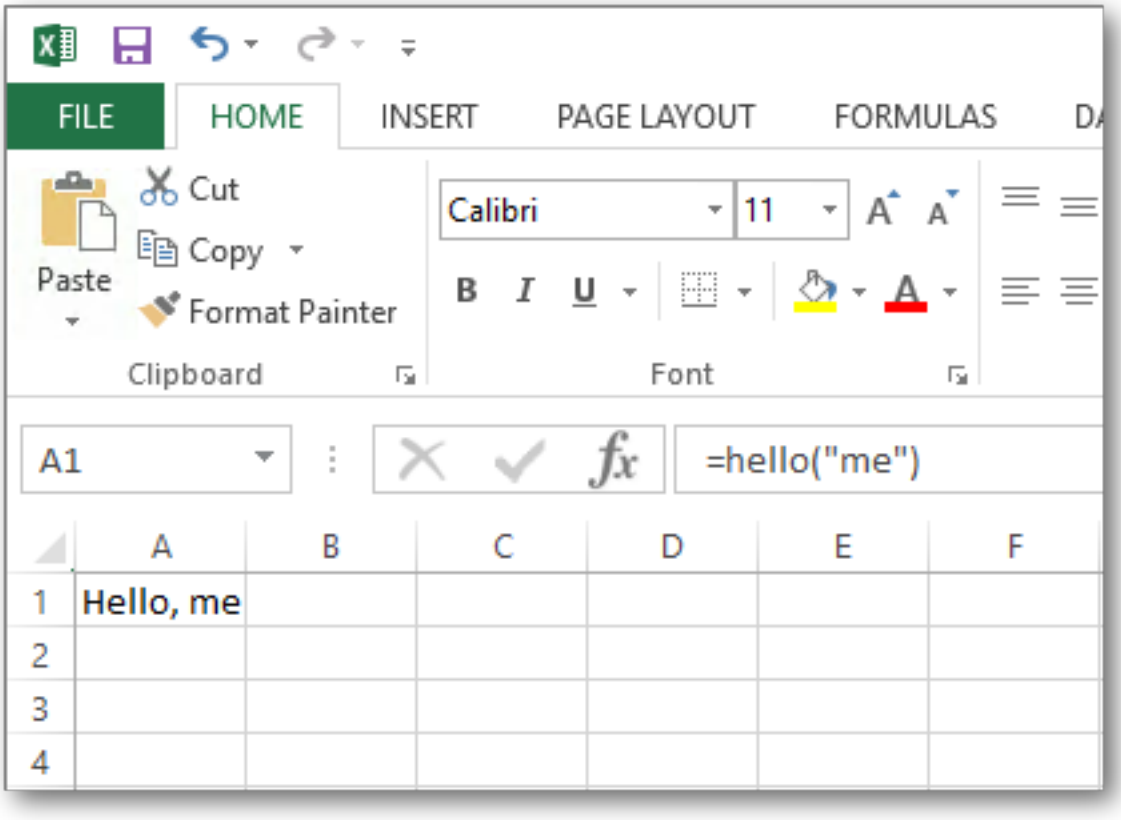
The decorator *xl_func* tells PyXLL to register that Python function as a worksheet function in Excel.

Once that code is saved it can be added to the *pyxll.cfg* config file:

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

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

When you reload the PyXLL addin or restart Excel the function you have just added will be available to use in a formula in an Excel worksheet.

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



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

Documenting Functions

When a python function is exposed to Excel the docstring of that function is visible in Excel's function wizard dialog.

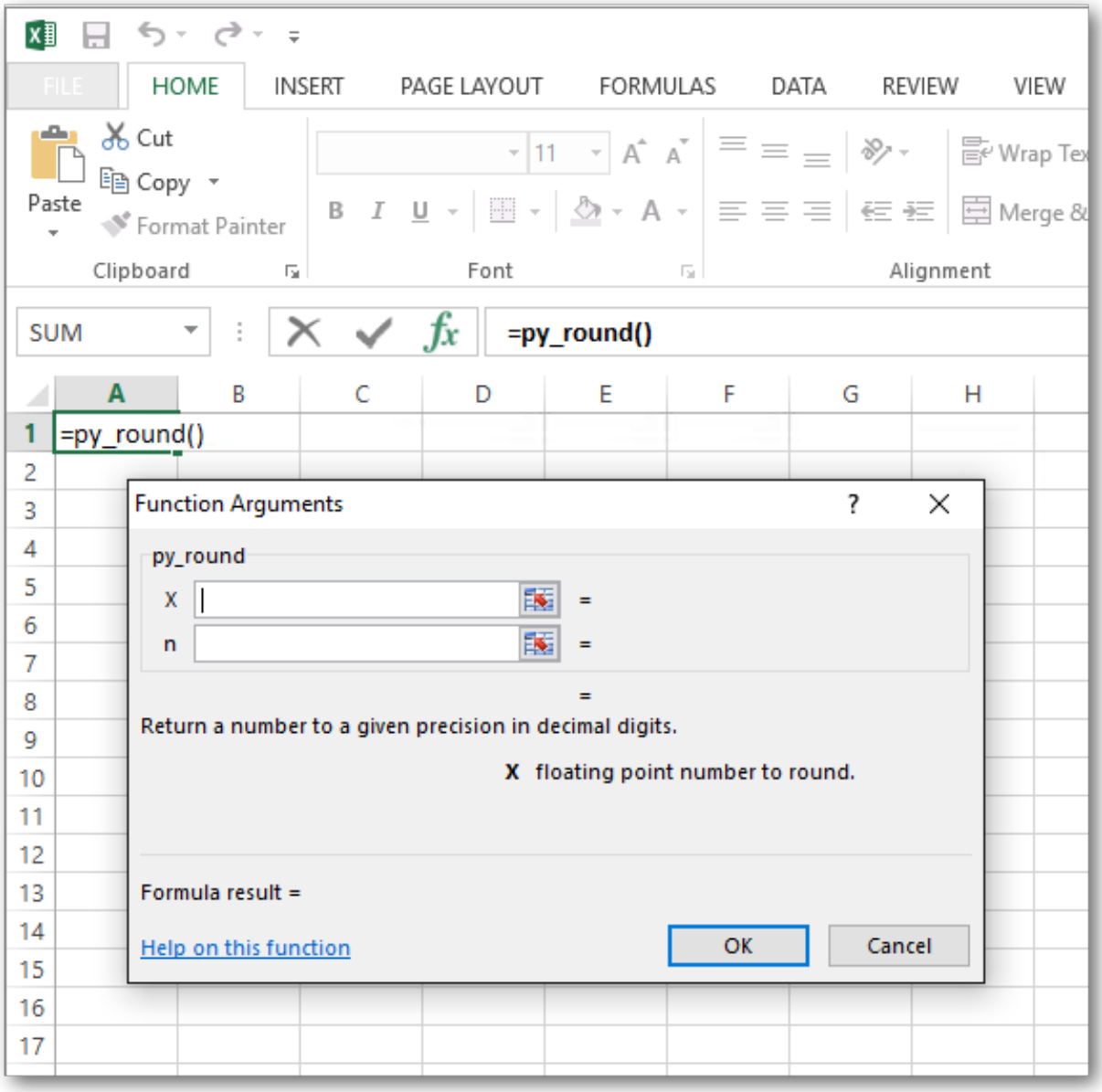
Parameter documentation may also be provided help the user know how to call the function. The most convenient way to add parameter documentation is to add it to the docstring as shown in the following example:

```
from pyxll import xl_func

@xl_func
def py_round(x, n):
    """
    Return a number to a given precision in decimal digits.

    :param x: floating point number to round
    :param n: number of decimal digits
    """
    return round(x, n)
```

Here PyXLL will automatically detect that the last two lines of the docstring are parameter documentation. 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.

One or more of any of the characters *:*, - or = may be used to separate the parameter name from it's description, and the Sphinx style *:param x: description* is also recognized.

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.



Function Signatures

When a Python function is registered in Excel it's possible to specify what types the arguments to that function are expected to be, and what the return type is.

This can be useful to avoid having to do type checking in the function itself, and in many cases it can helpfully do any necessary type conversion for you before your function is called.

One common example of this is how Excel handles dates. Internally, an Excel date is just a number. If you call a function with no type information with a date then that argument will just be a floating point number when it is passed to your Python function. Rather than convert from a float to a date in every function that expects a date you can annotate your Python function to tell PyXLL and Excel what type you expect and have the conversion done automatically.

To add type information to a function you must provide a signature string as the first argument to the *xl_func* decorator, or if you are using Python 3 you may use type annotations.

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.

Here is an example function that takes a date and an integer and returns another date.

Using a function signature string:

```
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)
```

And the same example using type annotations in Python 3:

```
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)
```

Standard Types

Several standard types may be used in the signature specifed when exposing a Python worksheet function. It is also possible to pass arrays and custom types, which are discussed later.

Below is a list of the standard types. Any of these can be specified as an argument type or return type in a function signature. If a type passed from Excel or returned from Python is not (or cannot be converted to) the Python type in this list an error will be written to the log file and NaN will be returned to Excel if possible.

PyXLL type	Python type
var	`object`
int	`int`
float	`float`
string	`str`
unicode	`unicode`[1]
bool	`bool`
datetime	`datetime.datetime`
date	`datetime.date`
time	`datetime.time`
xl_cell	*`XLCell`*
rtd	*`RTD`*

The Var Type

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

[1] 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.

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.

Using Arrays

Ranges of cells can be passed from Excel to Python as a 2d array, represented in python as a list of lists.

Any type can be used as an array type by appending *[]*, as shown in the following example:

```
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

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

    return total
```

Tip: 1d arrays

If you want to pass rows and columns of data see *Custom types and arrays* in the example *Custom Types*.

Arrays can be used as return values as well. When returning an array remember that it has to be a list of lists. This means to return a row of data you would return *[[1,2,3,4]]*, for example. To enter an array forumla 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[]* requires the least marshalling between Excel and python and is therefore the fastest of the array types.

If you use the *var* type in your function signature (or if there is no signature) then an array type will be used if you return a list of lists, or if the argument to your function is a range of data.

Using NumPy arrays

To be able to use `numpy` arrays you must have `numpy` installed and in your pythonpath.

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. Only up to 2d arrays are supported, as higher dimension arrays don't fit well with how data is arranged in a spreadsheet.

The most common type of `numpy` array to use is a 2d array of floats, for which the type to use in the function signature is *numpy_array*. For 1d arrays, the types *numpy_row* and *numpy_column* may be used.

Types other than floating point arrays are supported too, and are listed below for *numpy_array*. The same applies to the 1d array types.

PyXLL type	Python type
numpy_array	`numpy.array` of `float`
numpy_array<float>	`numpy.array` of `float`
numpy_array<int>	`numpy.array` of `int`
numpy_array<bool>	`numpy.array` of `bool`

Resizing Array Formulas

When returning an array, PyXLL can automatically resize the range used by the formula. To have PyXLL do this the *auto_resize* option to *xl_func* should be to the *True*, e.g:

```
from pyxll import xl_func

@xl_func("int rows, int cols: int[]", auto_resize=True)
def make_array(rows, cols, value):
    # create a 2d array of size (rows x cols)
    array = []
    for i in range(rows):
        row = []
        for j in range(cols):
            row.append(i * cols + j)
        array.append(row)
    return array
```

The default setting for *auto_resize* can be set in the *config*.

Passing Errors as Values

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

PyXLL has two different ways of doing this.

The first is to use the *var* type, which passes Excel errors as Python exception objects. Below is a table that shows how Excel errors are converted to python exception objects when the *var* type is used.

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

The second is to use the special type: *float_nan*.

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 *1.#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 also a slight performance penalty for using the *float_nan* type when compared to a plain *float*.

Errors can also be returned to Excel using instances of python exception types. This way, it is possible to return arrays where some values are errors but some aren't.

Retrieving Error Information

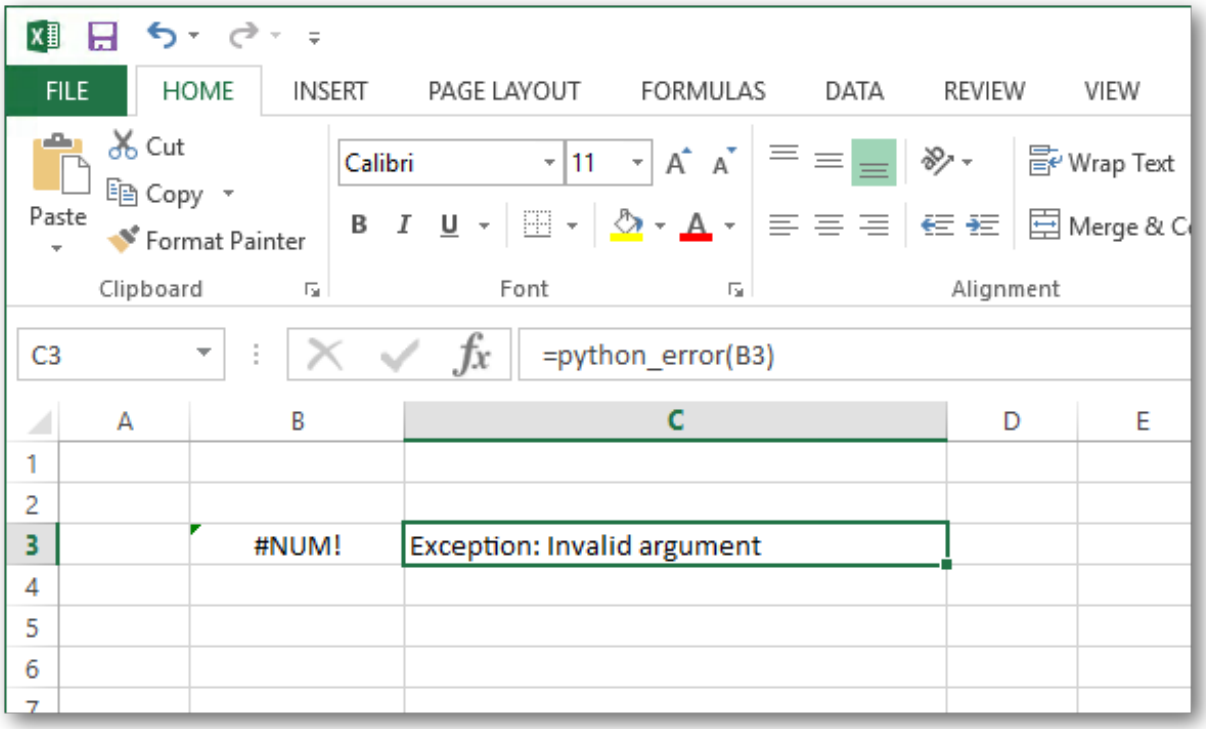
When a Python function is called from an Excel worksheet, if an uncaught exception is raised PyXLL caches the exception and traceback as well as logging it to the log file.

The last exception raised while evaluating a cell can be retrieved by calling `get_last_error`.

`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. This is because doing so would add additional performance overhead to every function call.




```
from pyxll import xl_func, xl_menu, xl_version, get_last_error
import traceback

@xl_func("xl_cell: string")
def python_error(cell):
    """Call with a cell reference to get the last Python error"""
```

```
    exc_type, exc_value, exc_traceback = get_last_error(cell)
    if exc_type is None:
        return "No error"

    return "".join(traceback.format_exception_only(exc_type, exc_value))

@xl_menu("Show last error")
def show_last_error():
    """Select a cell and then use this menu item to see the last error"""
    selection = xl_app().Selection
    exc_type, exc_value, exc_traceback = get_last_error(selection)

    if exc_type is None:
        xlcAlert("No error found for the selected cell")
        return

    msg = "".join(traceback.format_exception(exc_type, exc_value, exc_traceback))
    if xl_version() < 12:
        msg = msg[:254]

    xlcAlert(msg)
```

Custom Types

As well as the standard types listed above, it's also possible to define your own argument and return types that can then be used in your function signatures.

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

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

Here's an example of a simple custom type:

```
from pyxll import xl_arg_type

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)
```

CustomType can now be used as an argument type in a function signature. The Excel UDF will take a string, but before your Python function is called the conversion function will be used to 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 is decorated with the *xl_return_type* decorator.

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

```
from pyxll import xl_return_type, xl_func

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

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

Any recognized type can be used as a base type. That can be a standard 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.

There are more examples of custom types included in the PyXLL download.

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 source and target types by name 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))
```

Asynchronous Functions

In Excel 2010 Microsoft introduced asynchronous functions. Instead of returning a value immediately an asynchronous function receives a handle which is later used, from any thread, to return a value to Excel after the main function has returned.

Asynchronous functions can be used for non-CPU intensive tasks[2] such as requesting data or a calculation from a remote server. When the result is ready *xlAsyncReturn* is called to return the value to Excel. By using an asynchronous function Excel can continue to calculate other functions while the asynchronous function continues in the background.

PyXLL makes registering an asynchronous function very simple. By using the type `async_handle` in the function signature passed to *xl_func* the function automatically gets registered as an asynchronous function.

The `async_handle` parameter will be a unique handle for that function call and must be used to return the result when it's ready. The `async_handle` type should be considered opaque and any functions using that type shouldn't return a value.

The `async_handle` 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[3]

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

class MyThread(Thread):
    def __init__(self, async_handle, x):
        Thread.__init__(self)
        self.__async_handle = async_handle
        self.__x = x

    def run(self):
        # here would be your call to a remote server or something like that
        time.sleep(5)
        xlAsyncReturn(self.__async_handle, self.__x)

# no return type required as 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 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
```

[2] 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.

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

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* while a Python function is being called.

This will usually cause the function to exit, but if the `KeyboardInterrupt` exception is caught then it will not. 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.

The *allow_abort* feature can be enabled for all functions by setting it in the configuration.

```
[PYXLL]
allow_abort = 1
```

It is not enabled by default as it can interfere with the operation of some remote debugging tools as it uses the same Python trace mechanism used by them.

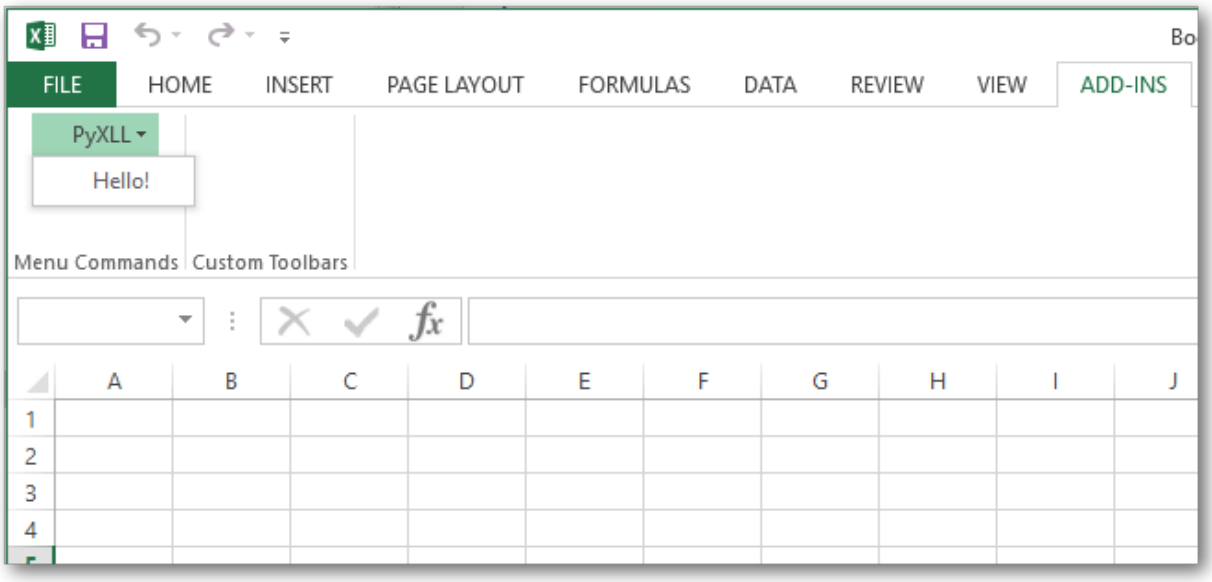
1.3.3 Menu Functions

The *xl_menu* decorator is used to expose a python function as a menu callback. PyXLL creates the menu item for you, and when it's selected your python function is called. That python function can call back into Excel using `win32com` or `comtypes` to make changes to the current sheet or workbook.

Different menus can be created and you can also create submenus. The order in which the items appear is controlled by optional keyword arguments to the *xl_menu* decorator.

Here's a very simple example that displays a message box when the user selects the menu item:

```
from pyxll import xl_menu, xlcAlert

@xl_menu("Hello!")
def on_hello():
    xlcAlert("Hello!")
```



Menu items may modify the current workbook, or in fact do anything that you can do via the Excel COM API. This allows you to do anything in Python that you previously would have had to have done in VBA.

Below is an example that uses *xl_app* to get the Excel Application COM object and modify the current selection. You will need to have `win32com` or `comtypes` installed for this.

```
from pyxll import xl_menu, xl_app

@xl_menu("win32com menu item")
def win32com_menu_item():
    # get the Excel Application object
    xl = xl_app()

    # get the current selected range
    selection = xl.Selection

    # set some text to the selection
    selection.Value = "Hello!"
```

New Menus

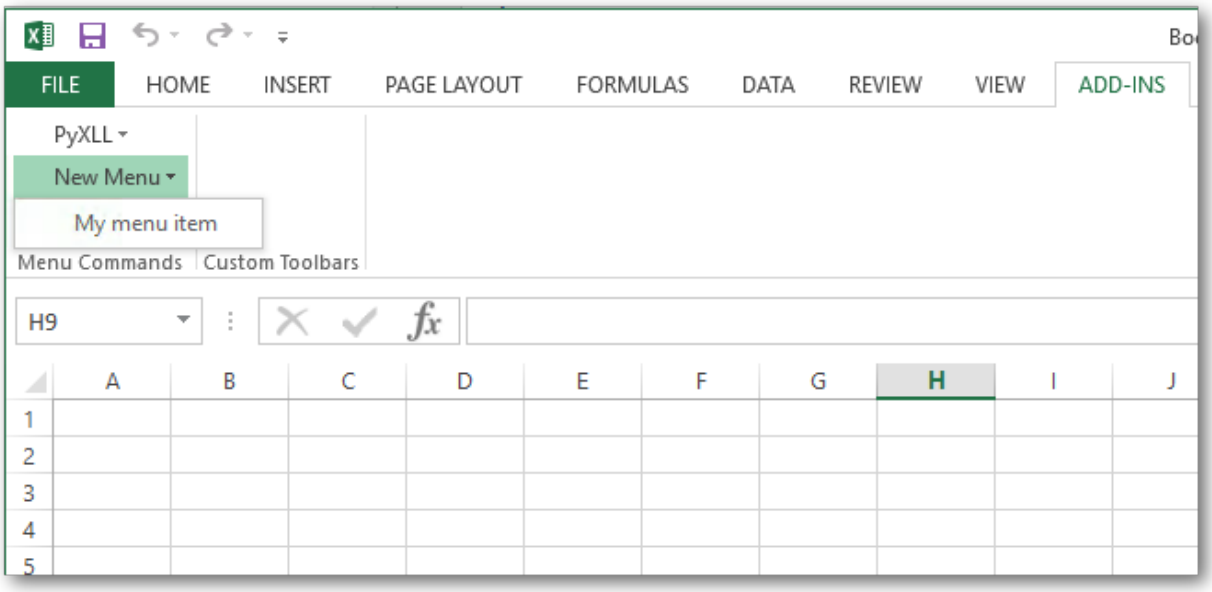
As well as adding menu items to the main PyXLL addin menu it's possible to create entirely new menus.

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")
```



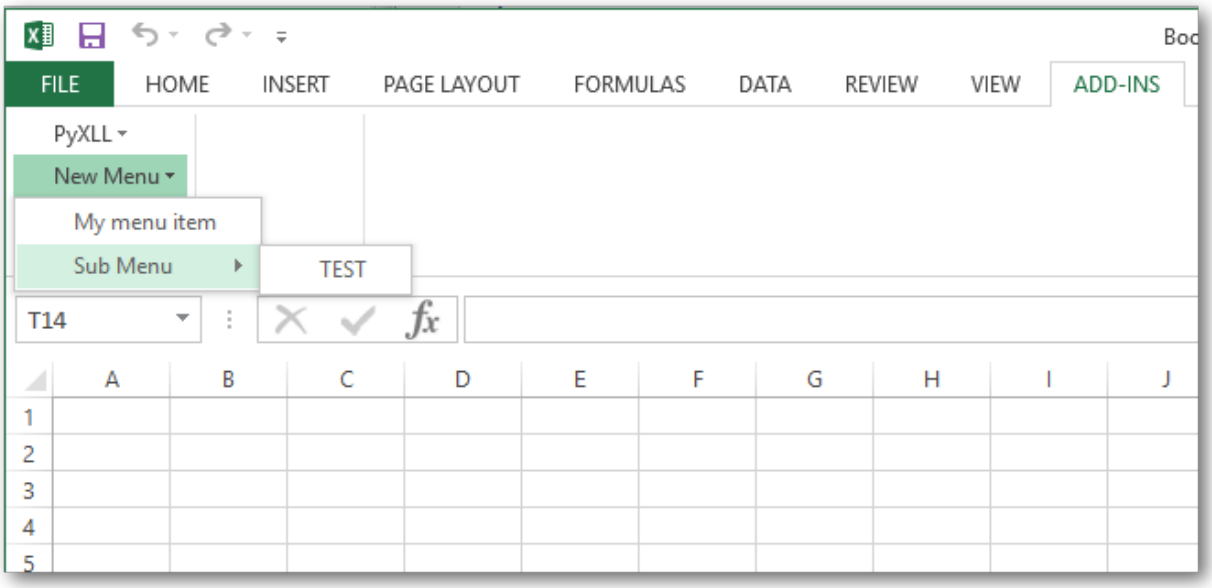
Sub-Menus

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")
```



1.3.4 Customizing the Ribbon

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 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.

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>
```

[1] Microsoft Ribbon Resources

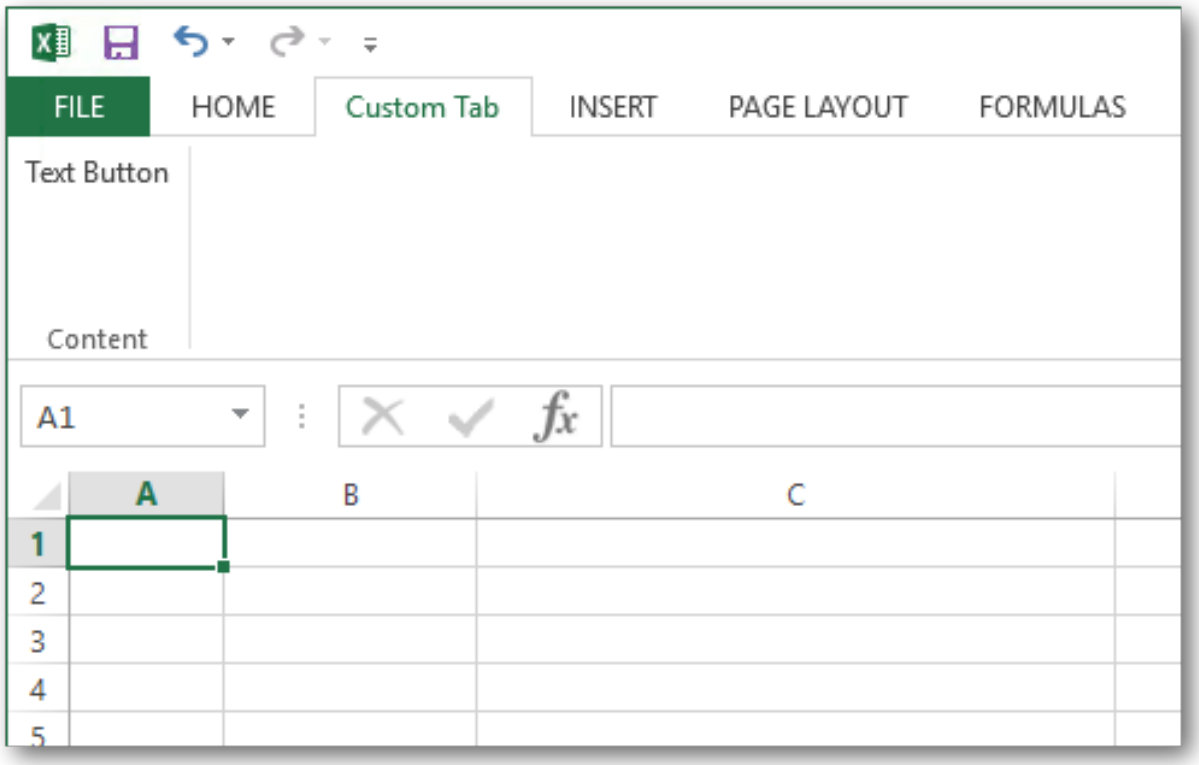
- Ribbon XML
- Walkthrough: Creating a Custom Tab by Using Ribbon XML
- XML Schema Reference

```
            </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.

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 and add the *on_text_button* function[2].

```
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.

Using Images

Some controls can use an image to give the ribbon whatever look you like. These controls have an *image* attribute and a *getImage* attribute.

The *image* attribute is set to the filename of an image you want to load. The *getImage* attribute is a function that will return a COM object that implements the *IPicture* interface.

PyXLL provides a function, `load_image`, that loads an image from disk and returns a COM Picture object. This can be used instead of having to do any COM programming in Python to load images.

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

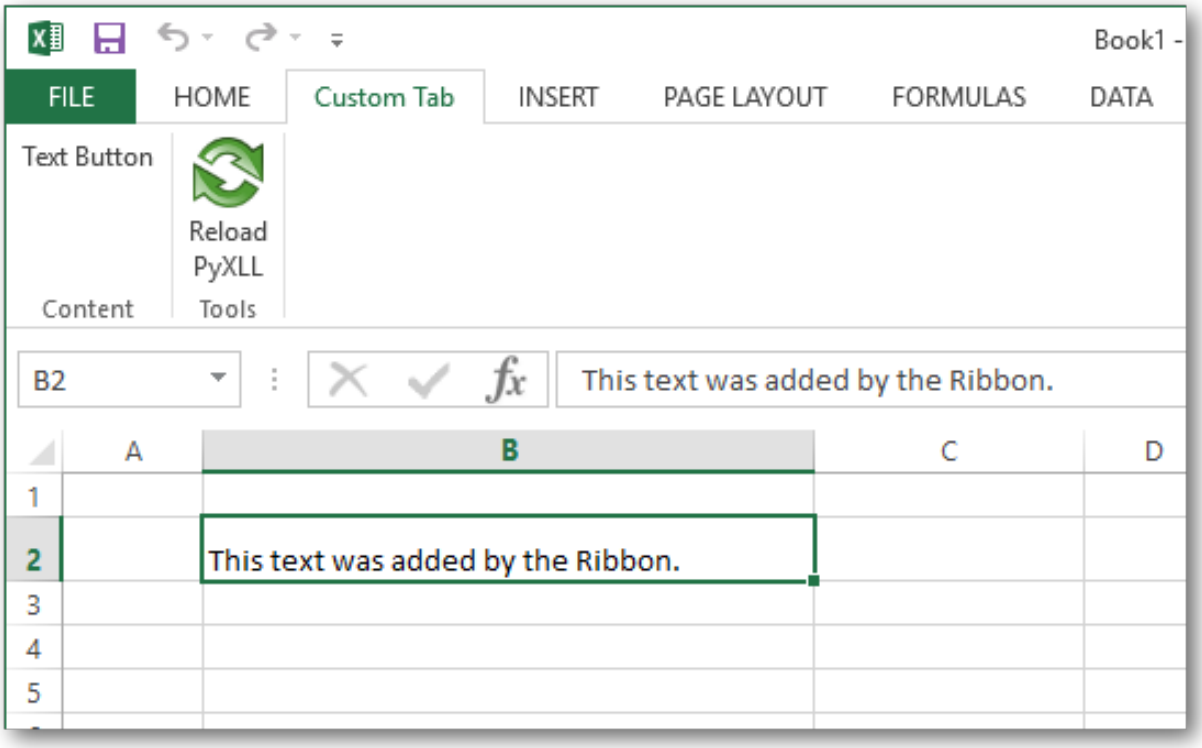
The following shows the example above with a new button added and the *loadImage* handler set.

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/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"
```

[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.

```
                            onAction="pyxll.reload"
                            image="reload.png"/>
                </group>
            </tab>
        </tabs>
    </ribbon>
</customUI>
```



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.

1.3.5 Macro Functions

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.

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)
```

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):

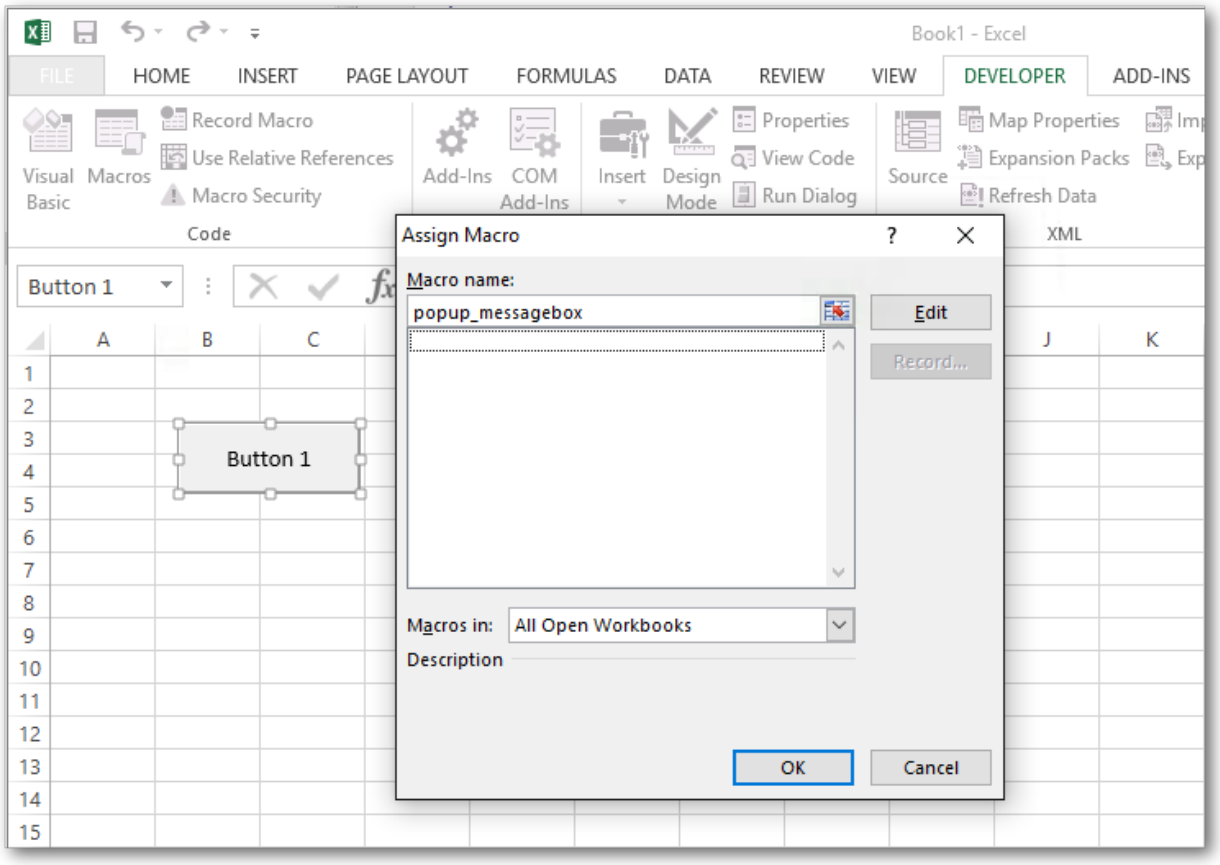
- Backspace

- Break
- CapsLock
- Clear
- Delete
- Down
- End
- Enter
- Escape
- Home
- Insert
- Left
- NumLock
- PgDn
- PgUp
- Right
- ScrollLock
- Tab

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.



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
```

1.3.6 Real Time Data

Real Time Data (or *RTD*) is data that updates according to it's own schedule, not 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 it's current value and updates the value displayed.

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*.

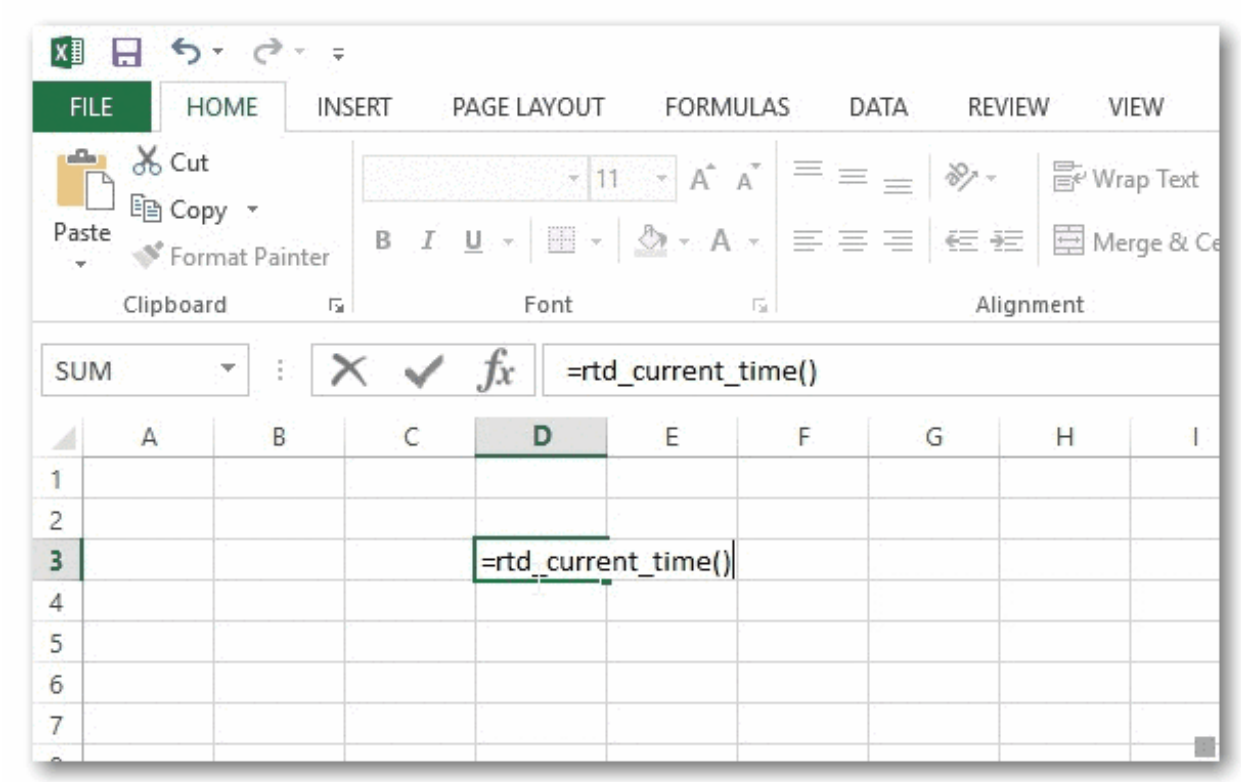
RTD functions have the return type *rtd*.

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.

Example Usage

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



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

```
from pyxll import xl_func, RTD

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()
```

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
```

1.4 API Reference

1.4.1 Function Decorators

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

- *xl_func*
- *xl_menu*
- *xl_macro*
- *xl_arg_type*
- *xl_return_type*

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*)

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 wizzard.

Parameters

- **signature** (*string*) – string specifying the argument types and, optionally, their names and the return type. If the return type isn't specified the var type is assumed. eg:

  `"int x, string y: double"` for a function that takes two arguments, x and y and returns a double.

  `"float x"` or `"float x: var"` for a function that takes a float x and returns a variant type.

  If no signature is provided the argument and return types will be inferred from any type annotations, and if there are no type annotations then the types will be assumed to be `var`.

  See *Standard Types* for the built-in types that can be used in the signature.

- **category** (*string*) – String that sets the category in the Excel function wizard the exposed function will appear under.
- **help_topic** (*string*) – Path of the help file (.chm) 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*).
- **volatile** (*boolean*) – if True the function will be registered as a volatile function, which means it will be called everytime 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.

Example usage:

```
from pyxll import xl_func

@xl_func
def hello(name):
    """return a familiar greeting"""
    return "Hello, %s" % name

# Python 3 using type annotations
@xl_func
def hello2(name: str) -> str:
    """return a familiar greeting"""
    return "Hello, %s" % name

# Or a signature may be provided as string
@xl_func("int n: int", category="Math", thread_safe=True)
def fibonacci(n):
    """naive iterative implementation of fibonacci"""
    a, b = 0, 1
    for i in xrange(n):
        a, b = b, a + b
    return a
```

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")
```

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*).

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)
```

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

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

1.4.2 Utility Functions

- *reload*
- *rebind*
- *xl_app*
- *xl_version*
- *async_call*
- *get_config*
- *get_dialog_type*
- *get_last_error*
- *get_type_converter*
- *load_image*

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.

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()
```

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.

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

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
```

get_config

get_config()

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

See also *Configuration*.

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

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)
```

get_type_converter

get_type_converter(*src_type*, *dest_type*)

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*) – name of type to convert from
- **dest_type** (*string*) – name of type to convert to

Returns function to convert from src_type to dest_type

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*.

1.4.3 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.

- *get_ribbon_xml*
- *set_ribbon_xml*
- *set_ribbon_tab*

- *remove_ribbon_tab*

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.

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.

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).

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.

1.4.4 Event Handlers

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

- *xl_on_open*
- *xl_on_reload*
- *xl_on_close*
- *xl_license_notifier*

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 ...
```

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()
```

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...")
```

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...
```

1.4.5 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].

- *xlfCaller*
- *xlfSheetId*
- *xlfGetWorkspace*
- *xlfGetWorkbook*
- *xlfGetWindow*
- *xlfWindows*
- *xlfVolatile*
- *xlcAlert*
- *xlcCalculation*
- *xlcCalculateNow*
- *xlcCalculateDocument*
- *xlAsyncReturn*
- *xlAbort*
- *xlSheetNm*

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]

xlfSheetId

xlSheetId(*sheet_name*)

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

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*.

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

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

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

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]

xlcAlert

xlcAlert(*alert*)
Pops up an alert window.

Callable from a macro or menu function only[1]

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

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

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]

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]

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

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.

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

1.4.6 Classes

- *RTD*
- *XLCell*
- *XLRect*

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.

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.

XLCell

class XLCell
XLCell represents the data and metadata for a cell in Excel passed as an `xl_cell` argument to a function registered with *xl_func*.

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].

value
value of the cell argument, passed in the same way as the `var` type.

Must be called from a 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 sheet equivalent function[1]

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 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 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.

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,
```

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

```
                cell.formula,
                cell.note)
```

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.

1.5 Examples

1.5.1 Worksheet Functions

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)
```

1.5.2 Custom Types

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.
"""

from pyxll import xl_func

import sys
if sys.version_info[0] > 2:
    from functools import reduce

#
# 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_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()


#
# 2) Custom types and arrays
#

#
# Array types may be used as the base types for custom types
# in the same way as any other type.
#

#
# This example shows how to reduce a range of data (list of
# lists) to a single list for use by a function.
#
# It also shows how it's possible to use multiple xl_arg_type
# decorators for the same function without duplicating code.
#

@xl_arg_type("int_list", "int[]")
@xl_arg_type("float_list", "float[]")
```

```
@xl_arg_type("custom1_list", "custom1[]")
def flatten(x):
    return reduce(lambda a,b: a + b, x, [])


@xl_func("float_list x: string")
def customarray_pyxll_function_1(x):
    # x is list of floats
    total = sum(x, 0)
    return "sum=%f : %s" % (total, x)


@xl_func("custom1_list x: string")
def customarray_pyxll_function_2(x):
    # x is a list of CustomType1 objects
    return "Hello %s" % (", ".join([c.name for c in x]))
```

1.5.3 Menu Functions

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)
```

1.5.4 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 from another
# thread, 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 will
# trigger a recalculation so you have to take care not to
# cause an infinite loop.
#
# 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 some
    # versions of Excel by updating the worksheet from a worksheet function
    def update_func():
        xl = xl_app()
        range = xl.Range(address)

        # 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
```

1.5.5 Event Handlers

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
    errors = []
    for modulename, module, exc_info in import_info:
        if module is None:
            exc_type, exc_value, exc_traceback = exc_info
            errors.append("Error loading '%s' : %s" % (modulename, exc_value))

    if errors:
        # report any errors to the user
        xlcAlert("\n".join(errors) + "\n\n(See callbacks.py example)")


@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 which modules didn't import correctly
    errors = []
    for modulename, module, exc_info in import_info:
        if module is None:
            exc_type, exc_value, exc_traceback = exc_info
            errors.append("Error loading '%s' : %s" % (modulename, exc_value))

    if errors:
        # report any errors to the user
        xlcAlert("\n".join(errors) + "\n\n(See callbacks.py example)")
    else:
        # report everything reloaded OK
        xlcAlert("PyXLL Reloaded OK\n(See callbacks.py example)")

    # recalcuate 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")
```

1.5.6 Object Cache

Advanced example

This is an advanced example but it's fine to just use this code as it is.

All you have to do is include this code in your project and use the custom type *cached_object* as the return type of functions returning Python objects and as the argument type for functions expecting those objects.

Look at the functions `cached_object_return_test` and `cached_object_arg_test` in the code below.

This examples shows how Python objects can be passed *on the Excel grid*.

Using this example you can declare Python functions that return complex Python objects and functions that accept them as arguments *without* converting to and from basic types by storing the complex objects in an object cache.

A custom type `cached_object` is used to add the returned Python object to an object cache and return a string key into that cache that's displayed in Excel.

When the custom type `cached_object` is used as an argument to a function it looks up that string key in the cache and retreives the cached object.

The function *`xlfCaller`* is used to determine which cell 'owns' the Python object and if that cell is updated the cache will remove its reference to that object and the new one is inserted in the cache.

Excel COM event handlers are used to monitor changes to the workbooks and worksheets so the object cache can be kept up to date as cells change, workbooks are closed and sheets are deleted.

All examples are included in the PyXLL download.

`Plain text version`

```
"""
PyXLL Examples: Object Cache

Excel cells hold basic types (strings, numbers, booleans etc) but sometimes
it can be useful to have functions that take and return objects and to be
able to call those functions from Excel.

This example shows how a custom type ('cached_object') and an object cache
```

```
can be used to pass objects between functions using PyXLL.

It also shows how COM events can be used to remove items from
the object cache when they are no longer needed.
"""
import logging
from pyxll import (
    xlfCaller,
    xl_arg_type,
    xl_return_type,
    xl_func,
    xl_on_open,
    xl_on_close,
    xl_on_reload,
    xl_app
)

_log = logging.getLogger(__name__)

# win32com is required for the code that cleans up the cache in response to Excel␣
↪events.
# The basic ObjectCache and related code will work without these modules.
try:
    import win32com.client
    _have_win32com = True
except ImportError:
    _log.warning("*** win32com.client could not be imported           ***")
    _log.warning("*** some of the objectcache examples will not work  ***")
    _log.warning("*** to fix this, install the pywin32 extensions     ***")
    _have_win32com = False


class ObjectCacheKeyError(KeyError):
    """
    Exception raised when attempting to retrieve an object from the
    cache that's not found.
    """
    def __init__(self, key):
        KeyError.__init__(self, key)


class ObjectCache(object):
    """
    ObjectCache maintains a cache of objects returned to Excel
    and the cells referring to those objects.

    As xl functions return objects they update the cache and
    any previously cached objects are removed from the cache
    when they are no longer referred to by any cells.

    Custom functions don't reference this class directly,
    instead they use the custom type 'cached_object' which
    is registered with PyXLL after this class.
    """

    def __init__(self):
        # dict of workbooks -> worksheets -> cell to object ids
        self.__cells = {}
```

```
        # dict of object ids to (object, {[referring (wb, ws, cell)] -> None})
        self.__objects = {}

        # set of open workbooks and opening workbooks
        self.__workbooks = set()
        self.__pending_workbooks = set()

    def __len__(self):
        """returns the number of cached objects"""
        return len(self.__objects)

    @staticmethod
    def _get_obj_id(obj):
        """returns the id for an object stored in the cache"""
        # the object id must be unique for objects within the cache
        cls_name = getattr(obj, "__class__", type(obj)).__name__
        return "<%s instance at 0x%x>" % (cls_name, id(obj))

    def add_workbook(self, workbook):
        """
        Called when a Workbook opens.
        Clears any cached objects from any previous workbooks with the same name.

        :param workbook: Workbook instance.
        """
        workbook_name = str(workbook.Name)

        if workbook_name not in self.__pending_workbooks:
            self.delete_all(workbook_name)

        self.__workbooks.add(workbook_name)
        self.__pending_workbooks.discard(workbook_name)

        # Would like to delete invalid cells since we may have closed/reopened
        # but need access to individual sheet objects
        self.delete_invalid(workbook)

    def update(self, workbook_name, sheet_name, cell, value):
        """
        Updates the cached value for a workbook, sheet and cell and returns the cache␣
↪id.

        :param str workbook_name: Workbook name.
        :param str sheet_name: Worksheet name.
        :param cell: (row, col) tuple.
        :param value: Value to be cached.
        """
        obj_id = self._get_obj_id(value)

        # remove any previous entry in the cache for this cell
        self.delete(workbook_name, sheet_name, cell)

        _log.debug("Adding entry %s to cache at (%s, %s, %s)" % (
            obj_id, workbook_name, sheet_name, cell))

        # update the object cache to include this cell as a referring cell
        # (a dict is used instead of a set to be compatible with older python␣
↪versions)
```

```
        unused, referring_cells = self.__objects.setdefault(obj_id, (value, {}))
        referring_cells[(workbook_name, sheet_name, cell)] = None

        # update the cache of cells to object ids
        self.__cells.setdefault(workbook_name, {}).setdefault(sheet_name, {})[cell] =␣
↪obj_id

        # note that this workbook is opening if it's not in our set of workbooks
        if workbook_name not in self.__workbooks:
            self.__pending_workbooks.add(workbook_name)

        # return the id for fetching the object from the cache later
        return obj_id

    def get(self, obj_id):
        """
        Return an object stored in the cache by the object id returned
        from the update method.

        :param obj_id: Identifier returned by `update`.
        """
        try:
            return self.__objects[obj_id][0]
        except KeyError:
            raise ObjectCacheKeyError(obj_id)

    def delete(self, workbook_name, sheet_name, cell):
        """
        Deletes the cached value for a workbook, sheet and cell.

        :param str workbook_name: Workbook name.
        :param str sheet_name: Worksheet name.
        :param cell: (row, col) tuple.
        """
        try:
            obj_id = self.__cells[workbook_name][sheet_name][cell]
        except KeyError:
            # Nothing cached for this cell.
            return

        _log.debug("Removing entry %s from cache at (%s, %s, %s)" % (
            obj_id, workbook_name, sheet_name, cell))

        # Remove this cell from the object's referring cells and remove the
        # object from the cache if no more cells are referring to it.
        obj, referring_cells = self.__objects[obj_id]
        del referring_cells[(workbook_name, sheet_name, cell)]
        if not referring_cells:
            del self.__objects[obj_id]

        # Remove the entries from the __cells dict.
        wb_cache = self.__cells[workbook_name]
        ws_cache = wb_cache[sheet_name]
        del ws_cache[cell]
        if not ws_cache:
            del wb_cache[sheet_name]
        if not wb_cache:
            del self.__cells[workbook_name]
```

```
    def delete_all(self, workbook_name, sheet_name=None):
        """
        Delete all references in the cache by workbook, worksheet.

        :param str workbook_name: Workbook name.
        :param str sheet_name: Worksheet name.
        """
        wb_cache = self.__cells.get(workbook_name)
        if wb_cache is not None:
            if sheet_name is not None:
                sheet_names = [sheet_name]
            else:
                sheet_names = list(wb_cache.keys())

            for sheet_name in sheet_names:
                ws_cache = wb_cache.get(sheet_name)
                if ws_cache is not None:
                    for cell in list(ws_cache.keys()):
                        self.delete(workbook_name, sheet_name, cell)

    def delete_invalid(self, workbook, sheet=None):
        """
        Deletes all invalid references in the cache by workbook, worksheet.

        References are considered invalid when the cell contents no longer
        matches the cached object identifier of the cache object for that
        cell.

        Cached values can become invalid if a sheet or workbook is deleted,
        or if the contents of a cell that contained an object reference is
        overwritten.
        """
        workbook_name = workbook.Name
        wb_cache = self.__cells.get(workbook_name)
        if wb_cache is None:
            return

        if sheet is not None:
            sheet_names = [sheet.Name]
        else:
            sheet_names = wb_cache.keys()

        for sheet_name in sheet_names:
            ws_cache = wb_cache.get(sheet_name)
            if ws_cache is None:
                continue

            try:
                if sheet is None:
                    sheet = workbook.Worksheets(sheet_name)
                check_cell_contents = True
            except:
                check_cell_contents = False

            for cell, obj_id in list(ws_cache.items()):
                if check_cell_contents:
                    # The cell tuple is zero offset, but Cells expects them starting␣
↪from one.
```

```
                    row, col = cell
                    cell_value = sheet.Cells(row+1, col+1).Value
                    if cell_value == obj_id:
                        continue

                # Either the sheet doesn't exist or the cell value
                # doesn't match the object id, so delete it from
                # the cache.
                self.delete(workbook_name, sheet_name, cell)


#
# There's one global instance of the cache.
#
_global_cache = ObjectCache()


#
# Here we register the functions that convert the cached objects to and
# from more basic types so they can be used by PyXLL Excel functions.
#

@xl_return_type("cached_object", "string", allow_arrays=False, thread_safe=False)
def cached_object_return_func(x):
    """
    Custom return type for objects that should be cached for use as
    parameters to other xl functions.
    """
    global _global_cache
    # This requires the function to be registered as a macro sheet equivalent
    # function because it calls xlfCaller, hence macro=True in
    # the xl_return_type decorator above.
    #
    # As xlfCaller returns the individual cell a function was called from, it's
    # not possible to return arrays of cached_objects using the cached_object[]
    # type in a function signature. allow_arrays=False prevents a function from
    # being registered with that return type. Arrays of cached_objects as an
    # argument type is fine though.

    if _have_win32com:
        # _setup_event_handler creates an event handler for Excel events to
        # ensure the cache is kept up to date with cell changes.
        _setup_event_handler(_global_cache)

    # Get the calling sheet and cell.
    caller = xlfCaller()
    sheet = caller.sheet_name
    cell = (caller.rect.first_row, caller.rect.first_col)

    # Check the function isn't being used as an array function.
    assert cell == (caller.rect.last_row, caller.rect.last_col), \
        "Functions returning objects should not be used as array functions"

    # The sheet name will be in "[book]sheet" format.
    workbook = None
    if sheet.startswith("[") and "]" in sheet:
        workbook, sheet = sheet.strip("[").split("]", 1)
```

```
    # Update the cache and return the cached object id.
    return _global_cache.update(workbook, sheet, cell, x)


@xl_arg_type("cached_object", "string")
def cached_object_arg_func(x, thread_safe=False):
    """
    Custom argument type for objects that have been stored in the
    global object cache.
    """
    # Lookup the object in the cache by its cached object id.
    global _global_cache
    return _global_cache.get(x)


#
# Utility function to check how many objects are in the cache.
#
@xl_func(": int", volatile=True)
def cached_object_count():
    """Return the number of cached objects"""
    global _global_cache
    return len(_global_cache)


#
# So far we can cache objects and keep the cache up to date as
# functions are called and the return values change.
#
# However, if a cell is changed from a function that returns a cached
# object to something that doesn't there will be a reference
# left in the cache - and so references can be leaked. Or, if a workbook
# or worksheet is deleted objects will be leaked.
#
# We can hook into some of Excel's Application and Workbook events to
# detect when references to objects are no longer required and remove
# them from the cache.
#

class EventHandlerMetaClass(type):
    """
    A meta class for event handlers that don't repsond to all events.
    Without this an error would be raised by win32com when it tries
    to call an event handler method that isn't defined by the event
    handler instance.
    """
    @staticmethod
    def null_event_handler(*args, **kwargs):
        return None

    def __new__(mcs, name, bases, dict):
        # Construct the new class.
        cls = type.__new__(mcs, name, bases, dict)

        # Create dummy methods for any missing event handlers.
        cls._dispid_to_func_ = getattr(cls, "_dispid_to_func_", {})
        for dispid, name in cls._dispid_to_func_.iteritems():
            func = getattr(cls, name, None)
```

```
            if func is None:
                setattr(cls, name, EventHandlerMetaClass.null_event_handler)
        return cls


class ObjectCacheApplicationEventHandler(object):
    """
    An event handler for Application events used to clean entries from
    the object cache that would otherwise be missed.
    """
    __metaclass__ = EventHandlerMetaClass

    def __init__(self):
        # We have an event handler per workbook, but they only get
        # created once set_cache is called.
        self.__wb_event_handlers = {}
        self.__cache = None

    def set_cache(self, cache):
        self.__cache = cache

        # Create event handlers for all of the current workbooks.
        for workbook in self.Workbooks:
            wb = win32com.client.DispatchWithEvents(workbook,␣
↪ObjectCacheWorkbookEventHandler)
            wb.set_cache(cache)
            self.__wb_event_handlers[workbook.Name] = wb

    def OnWorkbookOpen(self, workbook):
        # This workbook can't have anything in the cache yet, so make
        # sure it doesn't (it's possible a workbook with the same name
        # was closed with some cached entries and this one was then
        # opened).
        if self.__cache is not None:
            self.__cache.add_workbook(workbook)

            # Create a new workbook event handler for this workbook.
            wb = win32com.client.DispatchWithEvents(workbook,␣
↪ObjectCacheWorkbookEventHandler)
            wb.set_cache(self.__cache)

            # Delete any previous handler now rather than possibly wait for the GC.
            if workbook.Name in list(self.__wb_event_handlers):
                del self.__wb_event_handlers[workbook.Name]
                self.__wb_event_handlers[workbook.Name] = wb

    def OnWorkbookActivate(self, workbook):
        # Remove any workbooks that no longer exist.
        wb_names = [x.Name for x in self.Workbooks]
        for name in list(self.__wb_event_handlers.keys()):
            if name not in wb_names:
                # It's gone so remove the cache entries and the wb handler.
                if self.__cache is not None:
                    self.__cache.delete_all(str(name))
                del self.__wb_event_handlers[name]

        # Add in any new workbooks, which can happen if a workbook has just been␣
↪renamed.
```

```
        if self.__cache is not None:
            for wb in self.Workbooks:
                if wb.Name not in self.__wb_event_handlers:
                    wb = win32com.client.DispatchWithEvents(wb,␣
↪ObjectCacheWorkbookEventHandler)
                    wb.set_cache(self.__cache)
                    self.__wb_event_handlers[wb.Name] = wb


class ObjectCacheWorkbookEventHandler(object):
    """
    An event handler for Workbook events used to clean entries from
    the object cache that would otherwise be missed.
    """
    __metaclass__ = EventHandlerMetaClass

    def __init__(self):
        # Keep track of sheets we know about for when sheets get deleted or renamed.
        self.__sheets = [x.Name for x in self.Sheets]
        self.__cache = None

    def set_cache(self, cache):
        self.__cache = cache

    def OnWorkbookNewSheet(self, sheet):
        # This work can't have anything in the cache yet.
        if self.__cache is not None:
            self.__cache.delete_all(str(self.Name), str(sheet.Name))

        # Add it to our list of known sheets.
        self.__sheets.append(sheet.Name)

    def OnSheetActivate(self, sheet):
        # Remove any worksheets that not longer exist.
        ws_names = [x.Name for x in self.Sheets]
        for name in list(self.__sheets):
            if name not in ws_names:
                # It's gone so remove the cache entries an the reference.
                if self.__cache is not None:
                    self.__cache.delete_all(str(self.Name), str(name))
                self.__sheets.remove(name)

        # Ensure our list includes any new names due to renames.
        self.__sheets = ws_names

    def OnSheetChange(self, sheet, change_range):
        if self.__cache is not None:
            self.__cache.delete_invalid(self, sheet)


_event_handlers = {}
def _setup_event_handler(cache):
    # Only setup the app event handler once.
    if cache not in _event_handlers:
        xl = xl_app(com_package="win32com")
        app_handler = win32com.client.DispatchWithEvents(xl,␣
↪ObjectCacheApplicationEventHandler)
        app_handler.set_cache(cache)
```

```
        _event_handlers[cache] = app_handler


@xl_on_open
def _startup(*args):
    _setup_event_handler(_global_cache)


@xl_on_reload
@xl_on_close
def _delete_event_handlers(*args):
    # Make sure the event handles are deleted now as otherwise they could still
    # exist for a while until the GC gets to them, which can stop Excel from closing
    # or result in old event handlers still running if this module is reloaded.
    #
    # If you never wanted to reload this module, you could just import it from another
    # module loaded by pyxll and remove it from the pyxll.cfg and remove the
    # @xl_on_reload callback.
    #
    global _event_handlers
    handlers = _event_handlers.values()
    _event_handlers = {}
    while handlers:
        handler = handlers.pop()
        del handler
```

1.5.7 Developer Tools

Reloading and Importing Modules

All examples are included in the PyXLL download.

Plain text version

```
"""
PyXLL Examples: reload.py

This script can be called from outside of Excel to load and
reload modules using PyXLL.

It uses win32com (part of pywin32) to call into Excel to two built-in
PyXLL Excel macros ('pyxll_reload' and 'pyxll_rebind') and another
macro 'pyxll_import_file' defined in this file.

The PyXLL reload and rebind commands are only available in developer mode,
so ensure that developer_mode in the pyxll.cfg configuration is set to 1.

Excel must already be running for this script to work.

Example Usage:

# reload all modules
python reload.py

# reload a specific module
python reload.py <filename>
```

```
"""
import sys
import os
import cPickle
import logging

_log = logging.getLogger(__name__)

def main():
    # pywin32 must be installed to run this script
    try:
        import win32com.client
    except ImportError:
        _log.error("*** win32com.client could not be imported          ***")
        _log.error("*** tools.reload.py will not work                   ***")
        _log.error("*** to fix this, install the pywin32 extensions.    ***")
        return -1

    # any arguments are assumed to be filenames
    # of modules to reload
    filenames = None
    if len(sys.argv) > 1:
        filenames = sys.argv[1:]

    # this will fail if Excel isn't running
    xl_app = win32com.client.GetActiveObject("Excel.Application")

    # load the modules listed on the command line by
    # calling the macro defined in this file.
    if filenames:
        for filename in filenames:
            filename = os.path.abspath(filename)
            print "re/importing %s" % filename
            response = xl_app.Run("pyxll_import_file", filename)
            response = cPickle.loads(str(response))
            if isinstance(response, Exception):
                raise response

        # once all the files have been imported or reloaded
        # call the built-in pyxll_rebind macro to update the
        # Excel functions without reloading anything else
        xl_app.Run("pyxll_rebind")
        print "Rebound PyXLL functions"

    else:
        # call the built-in pyxll__reload macro
        xl_app.Run("pyxll_reload")
        print "Reloaded all PyXLL modules"

#
# in order to be able to reload particular files we add
# an Excel macro that has to be loaded by PyXLL
#
try:
    from pyxll import xl_macro

    @xl_macro
    def pyxll_import_file(filename):
```

```
        """
        imports or reloads a python file.

        Returns an Exception on failure or True on success
        as a pickled string.
        """
        # keep a copy of the path to restore later
        sys_path = list(sys.path)
        try:
            # insert the path to the pythonpath
            path = os.path.dirname(filename)
            sys.path.insert(0, path)

            try:
                # try to load/reload the module
                basename = os.path.basename(filename)
                modulename, ext = os.path.splitext(basename)
                if modulename in sys.modules:
                    module = sys.modules[modulename]
                    reload(module)
                else:
                    __import__(modulename)

            except Exception, e:
                # return the pickled exception
                return cPickle.dumps(e)

        finally:
            # restore the original path
            sys.path = sys_path

        return cPickle.dumps(True)

except ImportError:
    pass

if __name__ == "__main__":
    sys.exit(main())
```

Debugging with Eclipse and PyDev

All examples are included in the PyXLL download.

`Plain text version`

```
"""
PyXLL Examples: eclipse_debug.py

PyDev can be used to interactively debug Python code running
in Excel via PyXLL.

Before using this script you must have Eclipse and PyDev
installed:

http://www.eclipse.org/
http://pydev.org/
```

```
To be able to attach the PyDev debugger to Excel and you
Python code open the PyDev Debug perspective in Eclipse
and start the PyDev server by clicking the toolbar
button with a bug and a small P on it (hover over for the
tooltip).

Any python process can now attach to the PyDev debug
server by importing the 'pydevd' module included as part
of PyDev and calling pydevd.settrace()

This module adds an Excel menu item to attach to the
PyDev debugger, and also an Excel macro so that this
script can be run outside of Excel and call PyXLL to
attach to the PyDev debugger.

See http://pydev.org/manual_adv_remote_debugger.html
for more details about remote debugging using PyDev.

"""
import sys
import os
import logging
import time
import glob

_log = logging.getLogger(__name__)

##
## UPDATE THIS TO MATCH WHERE YOU HAVE ECLIPSE AND PYDEV INSTALLED
##
## The following code tries to guess where Eclipse is installed
eclipse_roots = [r"C:\"Program Files*\Eclipse"]
if "USERPROFILE" in os.environ:
    eclipse_roots.append(os.path.join(os.environ["USERPROFILE"],
                                      ".eclipse",
                                      "org.eclipse.platform_*"))

for eclipse_root in eclipse_roots:
    pydev_src = os.path.join(eclipse_root, r"plugins\org.python.pydev.debug_*\pysrc")
    paths = glob.glob(pydev_src)
    if paths:
        paths.sort()
        _log.info("Adding PyDev path '%s' to sys.path" % paths[-1])
        sys.path.append(paths[-1])
        break

def main():
    import win32com.client

    # get Excel and call the macro declared below
    xl_app = win32com.client.GetActiveObject("Excel.Application")
    xl_app.Run("attach_to_pydev")

#
# PyXLL function for attaching to the debug server
#
try:
    from pyxll import xl_menu, xl_macro, xlcAlert
```

```
    # if this doesn't import check the paths above
    try:
        import pydevd
        import pydevd_tracing
    except ImportError:
        _log.warn("pydevd failed to import - eclipse debugging won't work")
        _log.warn("Check the eclipse path in %s" % __file__)
        raise

    try:
        import threading
    except ImportError:
        threading = None

    # this creates a menu item and a macro from the same function
    @xl_menu("Attach to PyDev")
    @xl_macro
    def attach_to_pydev():
        # remove any redirection from previous debugging
        if getattr(sys, "_pyxll_pydev_orig_stdout", None) is None:
            sys._pyxll_pydev_orig_stdout = sys.stdout
        if getattr(sys, "_pyxll_pydev_orig_stderr", None) is None:
            sys._pyxll_pydev_orig_stderr = sys.stderr

        sys.stdout = sys._pyxll_pydev_orig_stdout
        sys.stderr = sys._pyxll_pydev_orig_stderr

        # stop any existing PyDev debugger
        dbg = pydevd.GetGlobalDebugger()
        if dbg:
            dbg.FinishDebuggingSession()
            time.sleep(0.1)
            pydevd_tracing.SetTrace(None)

        # remove any additional info for the current thread
        if threading:
            try:
                del threading.currentThread().__dict__["additionalInfo"]
            except KeyError:
                pass

        pydevd.SetGlobalDebugger(None)
        pydevd.connected = False
        time.sleep(0.1)

        _log.info("Attempting to attach to the PyDev debugger")
        try:
            pydevd.settrace(stdoutToServer=True, stderrToServer=True, suspend=False)
        except Exception, e:
            xlcAlert("Failed to connect to PyDev\n"
                     "Check the debug server is running.\n"
                     "Error: %s" % e)
            return

        xlcAlert("Attatched to PyDev")

except ImportError:
```

```
    pass

if __name__ == "__main__":
    sys.exit(main())
```

INDEX

A

address (XLCell attribute), 54
async_call() (in module pyxll), 43

C

connect() (RTD method), 53

D

disconnect() (RTD method), 53

F

first_col (XLRect attribute), 55
first_row (XLRect attribute), 55
formula (XLCell attribute), 54

G

get_config() (in module pyxll), 44
get_dialog_type() (in module pyxll), 44
get_last_error() (in module pyxll), 45
get_ribbon_xml() (in module pyxll), 47
get_type_converter() (in module pyxll), 45

I

is_calculated (XLCell attribute), 54

L

last_col (XLRect attribute), 55
last_row (XLRect attribute), 55
load_image() (in module pyxll), 46

N

note (XLCell attribute), 54

R

rebind() (in module pyxll), 42
rect (XLCell attribute), 54
reload() (in module pyxll), 42
remove_ribbon_tab() (in module pyxll), 47
RTD (class in pyxll), 53

S

set_ribbon_tab() (in module pyxll), 47
set_ribbon_xml() (in module pyxll), 47
sheet_id (XLCell attribute), 54
sheet_name (XLCell attribute), 54

V

value (RTD attribute), 53
value (XLCell attribute), 54

X

xl_app() (in module pyxll), 43
xl_arg_type() (in module pyxll), 41
xl_func() (in module pyxll), 38
xl_license_notifier() (in module pyxll), 49
xl_macro() (in module pyxll), 40
xl_menu() (in module pyxll), 39
xl_on_close() (in module pyxll), 49
xl_on_open() (in module pyxll), 48
xl_on_reload() (in module pyxll), 48
xl_return_type() (in module pyxll), 41
xl_version() (in module pyxll), 43
xlAbort() (in module pyxll), 53
xlAsyncReturn() (in module pyxll), 52
xlcAlert() (in module pyxll), 52
xlcCalculateDocument() (in module pyxll), 52
xlcCalculateNow() (in module pyxll), 52
xlcCalculation() (in module pyxll), 52
XLCell (class in pyxll), 54
xlfCaller() (in module pyxll), 50
xlfGetDocument() (in module pyxll), 53
xlfGetWindow() (in module pyxll), 51
xlfGetWorkbook() (in module pyxll), 51
xlfGetWorkspace() (in module pyxll), 50
xlfVolatile() (in module pyxll), 51
xlfWindows() (in module pyxll), 51
XLRect (class in pyxll), 55
xlSheetId() (in module pyxll), 50
xlSheetNm() (in module pyxll), 53