
PyXLL User Guide

Release 1.3.3

PyXLL Ltd.

March 17, 2012

CONTENTS

1	Quick start	1
2	Config file	3
2.1	PyXLL settings	3
2.2	Logging	4
2.3	Environment Variables	4
2.4	License key	5
3	Writing User Defined Functions (UDFs)	6
3.1	Exposing functions as UDFs	6
3.2	Standard argument and return types	7
3.3	The var type	7
3.4	Using arrays	7
3.5	Using NumPy arrays	8
3.6	Passing errors as values	9
3.7	Custom types	9
3.8	Type conversion (New in PyXLL 1.3)	11
3.9	Passing cell metadata	11
3.10	Asynchronous functions	12
4	Creating custom menu items	14
4.1	Basic menu items	14
4.2	New menus	15
4.3	Sub-menus	15
5	Writing Excel macros	16
5.1	Exposing functions as macros	16
5.2	Calling macros from Excel	16
6	Calling back into Excel from Python	18
6.1	Macro sheet and command functions	18
6.2	Other non-macro functions	20
6.3	Automation / COM	21
7	PyXLL callbacks	22
8	Developer Tools (New in PyXLL 1.3)	24
8.1	Reloading	24
8.2	Debugging	25

9	Examples	26
9.1	Simple worksheet functions	26
9.2	Custom types	31
9.3	Menu functions	34
9.4	Macros and Excel Automation	36
9.5	PyXLL callbacks	39
9.6	Object cache	42
9.7	Developer Tools	50
	Index	55

QUICK START

Unpack the archive you've downloaded and open `pyxll.cfg` in a text editor. Set the `log_path` and `file` to wherever you want the log to go and change `pythonpath`¹ to where you're going to put your Python modules (this may be the same as where you've extracted the archive to for now), and set `modules` to your Python module. We'll write that Python module next.

If you have a license key you should copy it to the license section (see the [config section](#) for more details). If you do not have a license key you may still use PyXLL for non-commercial or evaluation purposes, but you will see a PyXLL pop-up dialog each time you start Excel.

Example config file:

```
[LOG]
verbosity = debug
path = c:/logs
file = pyxll.log
format = %(asctime)s - %(levelname)s : %(message)s

[PYXLL]
pythonpath = c:/my_python_modules
modules = my_xl_addin_module,my_other_xl_addin_module
developer_mode = 1

[LICENSE]
key = **put your license key here if you have one**
file = **or the path of your license file here**
```

Now create the python module that you specified in the config file using a text editor (in my example, the file will be called `my_xl_addin_module.py`).

First of all import `xl_func` from the `pyxll` module. The `pyxll` module is compiled into the PyXLL addin so does not need to be included in the `pythonpath`. Write a simple function that takes a string and returns a string, as shown in the example file below.

The python function needs to be decorated using the `xl_func` decorator in order for the PyXLL addin to expose it to Excel. The decorator takes one mandatory argument and several option arguments (as described in the `xl_func` section). The mandatory argument is the function signature which should specify the argument types and names (names are optional) and the return type. The return type is optional, and if not specified the default return type `var` will be used. `var` is a special type that can be used to pass any of the standard types. For more information about types see [Standard argument and return types](#).

Example python module:

¹ Your system `pythonpath` must also be setup so PyXLL can find the standard modules. The `pythonpath` in the config is only for additional paths.

```
from pyxll import xl_func

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

Now we're ready to use that function in Excel! Start Excel and add the PyXLL addin. If you're using Excel 97-2003 go to Tools -> Add-Ins -> Browse and locate pyxll.xll and add it. If you're using Excel 2007-2010 click the top left circle and go to options -> Add-Ins -> Manage Excel Addins and browse for the pyxll.xll file and add it.

New in PyXLL 1.3

PyXLL looks for the python dll in the same folder as itself first before trying to find it in the system path.

If at this stage you get an error saying that Python is not installed or the Python dll can't be found, the most likely problem is that you don't have the correct Python version installed on your PC. The Python version you have installed must match whatever version of PyXLL you downloaded (currently versions 2.3 to 2.7 are available). Install the correct version and try again. It's important that python2x.dll (where x is whatever version of python you're using) is in your system path (if you use one of the standard binary distributions of python it will already have been automatically installed somewhere in your system path).

You should now be able to enter the formula =hello("me") and see the calculated value "Hello, me". If you look in the function browser in Excel you should see your function in the PyXLL category with your docstring as the help text for the function. Using other categories for your functions is covered in the `xl_func` section.

Using Eclipse and PyDev?

You can interactively debug Python code running in PyXLL with Eclipse and PyDev by attaching the PyDev debugger. See this [example](#) for details.

If your function doesn't work, check the log file, which you will find in the directory you specified in the config file. If you have made any errors in your python module you will see those in the log file. Correct those errors and select the 'Reload PyXLL' menu item from the PyXLL Addin menu. This will either be in the main menu bar if you're using a version of Excel before 2007, or in the AddIns ribbon menu for 2007 and later versions.

CONFIG FILE

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 used to control settings for PyXLL. It's 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 do not have a config file, or it's not readable by PyXLL you will get an error when starting Excel or adding the addin and PyXLL will not work.

Paths used in the config file may be absolute 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 `%(your_envvar_name)s`, eg `verbosity = %(MY_PYXLL_LOG_VERBOSITY)s`.

The config file is broken down into several sections, each of which is documented below.

2.1 PyXLL settings

Settings that determine the basic behavior of the PyXLL addin are put in the `[PYXLL]` config section:

```
[PYXLL]
pythonpath = semi-colon or new line delimited list of directories
modules = comma or new line delimited list of python modules
developer_mode = (optional) 1 or 0 indicating whether or not to use the developer mode
external_config = (optional) paths of additional config files to load
name = (optional) name of the addin visible in Excel
```

New in PyXLL 1.3

`pythonpath` and `modules` may now span multiple lines.

The standard `pythonpath` is appended with whatever paths you specify in the `pythonpath` setting. This is useful so you can tell PyXLL where to look for the python modules containing your addin code.

When PyXLL is started (or re-loaded) it will import all modules listed in the config file. It's these modules that expose worksheet functions and menu functions.

The `developer_mode` setting can be used to put PyXLL in a developer mode, which is useful when you're developing your addin. In addition to whatever menu items you have added, when the developer mode is used there will be an addition menu item to reload the addin. This can be used to allow you to develop your python modules and test changes without having to restart Excel. If unset, the default is to not use the developer mode.

The `external_config` setting may be used to reference another config file somewhere else. 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 specifying them as a comma separated list. Values in external config files override what's in the parent config file, apart from `pythonpath`, `modules` and `external_config` which get appended to.

The `name` setting is optional and only has an effect when a valid license for PyXLL is found. It is used to change 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'.

2.2 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]
verbosity = logging level (debug, info, warning, error or critical)
path = directory of where to write the log file
file = filename of the log file
format = format string
```

The items in the config file may include substitution values that will be replaced with other section values or the config defaults setup by PyXLL.

Config default	Explanation
pid	process id
date	current date
xlversion	Excel version

These may be used, for example, to create the log file name `file = pyxll.%(date)s.%(pid)s.%(xlversion)s.log`. For more information about variable substitution in the config please see `ConfigParser.get`.

The format string is used by the logging module to format any log messages. An example format string is `"%(asctime)s - %(name)s - %(levelname)s - %(message)s"`. For more information about log formatting, please see the logging module documentation.

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

2.4 License key

If you have licensed PyXLL for commercial or evaluation use, you should put the license key in the [LICENSE] section of the config file.

If you do not have a license key, you may omit this section. Unlicensed versions of PyXLL will display a dialog box each time the addin is loaded which must be acknowledged before Excel will continue.

For convenience for users with site licenses it's also possible to specify a license file instead of a license key. The license file is a text file containing the site license key that must be readable by all users of PyXLL. This simplifies the process of updating a site license as it only needs to be updated in one shared file, rather than every installed instance of PyXLL.

```
[LICENSE]
key = (optional) license key
file = (optional) path to shared license key file
In the event that both key and file are specified, key takes precedence.
```

WRITING USER DEFINED FUNCTIONS (UDFS)

3.1 Exposing functions as UDFs

Python functions to be exposed as UDFs are decorated with the `xl_func` decorator, imported from the `pyxll` module. The `pyxll` module is compiled into the `pyxll.xll` addin so it doesn't have to be on the pythonpath.

```
pyxll.xl_func (signature [, category=PyXLL] [, help_topic="" ] [, thread_safe=False] [, macro=False] [,
               volatile=False] [, disable_function_wizard_calc=False] [, disable_replace_calc=False])
xl_func is a function that returns a decorator for exposing python functions to Excel.
```

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.
- **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 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`
- **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.

Example usage:

```
from pyxll import xl_func

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

@xl_func("int n: int", category="fibonacci", 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
```

3.2 Standard argument and return types

Several standard types may be used in the signature specified when exposing a Python UDF. 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
int	int
float	float
string	str
bool	bool
datetime	datetime.datetime
date	datetime.date
time	datetime.time
unicode	unicode ¹

3.3 The var type

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

3.4 Using arrays

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

¹Unicode 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.

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 formula in Excel you select the cells, enter the formula and then press `Ctrl+Shift+Enter`. Please refer to the Excel documentation for more information about array formula.

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 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.

3.5 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 returning 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
<code>numpy_array</code>	<code>numpy.array</code> of float
<code>numpy_array<float></code>	<code>numpy.array</code> of float
<code>numpy_array<int></code>	<code>numpy.array</code> of int
<code>numpy_array<bool></code>	<code>numpy.array</code> of bool

3.6 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, the value passed to python will be `float('nan')` or `l.#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 the same types. This way, it is possible to return arrays where some values are errors but some aren't.

3.7 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.

```
pyxll.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

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("custom", "string")
def string_to_customtype(x):
    return CustomType(x)

@xl_func("custom 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)
```

`custom` 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.

```
pyxll.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

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

```
from pyxll import xl_return_type, xl_func

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

@xl_func("string x: custom")
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.

3.8 Type conversion (New in PyXLL 1.3)

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.

```
pyxll.get_type_converter(src_type, dest_type)
```

Returns a function to convert objects of type `src_type` to `dest_type`.

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`

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.

Here's an example that shows how to get a `datetime` from a `var` parameter:

```
from pyxll import xl_func, get_type_converter
from datetime import datetime

@xl_func("var x: string")
def var_datetime_func(x):
    var_to_datetime = get_type_converter("var", "datetime")
    dt = var_to_datetime(x)
    # dt is now of type 'datetime'
    return "%s : %s" % (dt, type(dt))
```

3.9 Passing cell metadata

As well as passing arguments from Excel to Python by value, it's also possible to pass data about the cell being passed to your function using the `xl_cell` type.

When you use the `xl_cell` type you get passed an object of type `XLCell`.

class `pyxll.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`.

value

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

address

string representing the address of the cell, or `None` if a value was passed to the function and not a cell reference.

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.

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.

```
from pyxll import xl_func

@xl_func("xl_cell cell: string")
def xl_cell_test(cell):
    return "[value=%s, address=%s, formula=%s, note=%s]" % (
        cell.value,
        cell.address,
        cell.formula,
        cell.note)
```

3.10 Asynchronous functions

In Excel 2010 Microsoft introduced asynchronous functions. Instead of returning a value immediately, an asynchronous function initiates a potentially slow calculation in another thread, or perhaps via a server request, and returns immediately. When the result of the calculation is ready `xlAsyncReturn` is called to inform Excel.

PyXLL makes registering an asynchronous function very simple. By using the type `async_handle`, in the function signature passed to `xl_func`, the function will automatically get 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.

Here's an example of an asynchronous function ²

```
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 long running function or a call to a server
```

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

```
    # 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 long calculation in another thread
    thread = MyThread(h, x)
    thread.start()

    # return immediately, the real result will be returned by the thread function
    return
```

CREATING CUSTOM MENU ITEMS

4.1 Basic menu items

New menu items can be added to the PyXLL Excel Addin menu easily by using the `xl_menu` decorator.

```
pyxll.xl_menu (name [, menu=None] [, sub_menu=None] [, order=0] [, menu_order=0])
```

`xl_menu` is a function that returns a decorator for creating menu items that call Python functions.

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 lexicographically. If the item is a sub-menu item, this order influences where the sub-menu will appear in the main menu
- **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

To add a new menu entry to the PyXLL menu you use the `xl_menu` decorator with the label you want to use for that menu item, and a function that takes no arguments that will be called when that menu item is selected.

Here is an example of a simple menu item that uses `win32api` to display a message box to the user:

```
from pyxll import xl_menu
import win32api

@xl_menu("My menu item")
def my_menu_item():
    win32api.MessageBox(0, "Hello from PyXLL", "Menu button example")
```

Menu items may modify the current workbook, or in fact do anything that you can do via the Excel automation 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 the `win32com` module to call back into Excel from a menu item:

```
from pyxll import xl_menu, get_active_object
import win32com.client
```

```
@xl_menu("win32com menu item")
def win32com_menu_item():
    # get the Excel application object
    xl_window = get_active_object()
    xl_app = win32com.client.Dispatch(xl_window).Application

    # get the current selected range
    selection = xl.Selection

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

4.2 New menus

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

To create a new menu, use the `menu` keyword argument to the `xl_menu` decorator.

In addition, if you want to control the order in which menus are added you may use the `menu_order` integer keyword argument. The higher the value, the later in the ordering the menu will be added.

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
import win32api

@xl_menu("My menu item", menu="New Menu")
def my_menu_item():
    win32api.MessageBox(0, "Hello from PyXLL", "new menu example")
```

4.3 Sub-menus

Sub-menus may also be created. To add an item to a sub-menu, use the `sub_menu` keyword argument to the `xl_menu` decorator.

All sub-menu items share the same `sub_menu` argument. The ordering of the items within the submenu is controlled by the `sub_order` integer keyword argument. In the case of sub-menus, the `order` keyword argument controls the order of the sub-menu within the parent menu.

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
import win32api

@xl_menu("TEST", menu="My Menu", sub_menu="Sub Menu")
def my_submenu_item():
    win32api.MessageBox(0, "Hello from PyXLL", "sub menu example")
```

WRITING EXCEL MACROS

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.

5.1 Exposing functions as macros

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

```
pyxl1.xl_macro([signature="" ])
```

`xl_macro` is a function that returns a decorator for exposing python functions to Excel as macros.

Parameters `signature` (*string*) – 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 supplied, it is assumed the function takes no arguments and the return value is not used.

Example usage:

```
from pyxl1 import xl_macro
import win32api

@xl_macro()
def popup_messagebox():
    """pops up a message box"""
    win32api.MessageBox(0, "Hello", "Hello")

@xl_macro("string x: int")
def py_strlen(n):
    """returns the length of x"""
    return len(x)
```

5.2 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
```

On their own macros might seem a bit limited. When you consider you can call back into Excel from macros using the Excel COM API to do everything you previously would have done in VBA they suddenly become a lot more useful.

There are more examples of macros called from controls in the examples supplied with PyXLL.

CALLING BACK INTO EXCEL FROM PYTHON

While worksheet functions don't usually require to call back into Excel, it's usual that menu items and macros do.

Occasionally worksheet functions may also want to call back into Excel. For them to do so, they must be registered as macro sheet equivalent functions (see `x1_func`).

6.1 Macro sheet and command functions

The Excel C API provides a number of functions for querying information from Excel. A few of those are also available to call from PyXLL.

In order to call these functions from an Excel worksheet function the function must be registered as a macro sheet equivalent function (see *Exposing functions as UDFs*).

`pyxll.xlfCaller()`

Returns address of cell calling the current function, as a string

`pyxll.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`

`pyxll.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`

`pyxll.xlfGetWorkbook(arg_num[, workbook])`

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`

`pyxll.xlfGetWindow(arg_num[, window])`

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`

`pyxll.xlAllWindows` (*[match_type] [,mask]*)

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

`pyxll.xlAlert` (*alert*)

Pops up an alert window.

Callable from a macro or menu function only

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

`pyxll.xlcCalculation` (*calc_type*)

set the calculation type to automatic or manual.

Callable from a macro or menu function only

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

or `xlCalculationSemiAutomatic`

or `xlCalculationManual`

`pyxll.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

`pyxll.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

`pyxll.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

`xlCalculationAutomatic = 1`

```
xlCalculationSemiAutomatic = 2
```

```
xlCalculationManual = 3
```

The functions above have the same signature as their C counterparts and are simply wrappers of them. For more detailed information on their behavior refer to the Excel C API documentation.

Of these functions, the most likely to be found useful is `xlCaller`. It returns the address of the calling cell when called from within a macro sheet equivalent worksheet function. There is an example of its use in the automation example provided with PyXLL.

Some additional functions are provided that return information about the current state of Excel but are not simply wrappers of the Excel C API functions.

```
pyxll.get_active_object()
```

`pythoncom` must be installed for this function to work

Returns the PyIUnknown Excel COM object associated with the current window

```
pyxll.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
```

6.2 Other non-macro functions

The following functions also return information about Excel and the current state of PyXLL, but are not macro sheet functions and may be called from anywhere.

```
pyxll.get_config()
```

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

See also *Config file*.

```
pyxll.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

6.3 Automation / COM

Excel has a well known COM API that makes programming macros and menu functions in Python as familiar as writing them in VBA.

PyXLL provides a function `get_active_object` that returns the Excel Window COM object for the current Excel instance.

The code below shows how to get the Excel Application instance using `get_active_object` and `win32com`:

```
from pyxll import get_active_object
import win32com.client

def xl_app():
    xl_window = get_active_object()
    xl_app = win32com.client.Dispatch(xl_window).Application
    return xl_app
```

It's better to use `get_active_object` than `win32com.client.GetActiveObject("Excel.Application")` as it will always give you the current instance of Excel, whereas getting it by name will return you any running instance which may not be the same if you're running more than one instance of Excel.

Using the Excel COM API to modify the current worksheet is something that you might do from a menu item or from a macro. In some cases you may also want to do it from a worksheet function. Some versions of Excel will block calls to the COM API while calling a worksheet function and so if you try you will cause Excel to hang, even if using a macro sheet equivalent worksheet function.

Because of the deadlock problems associated with calling back into Excel from a worksheet function, PyXLL has a function `async_call`¹. It takes a callable object and calls it in a background thread. It returns immediately and so doesn't block waiting for the call back to Excel.

```
pyxll.async_call(func [, *args] [, **kwargs])
    async_call schedules func to be called from a background thread.
```

If args or kwargs contain any COM objects they will be marshalled across the thread boundary automatically.

Returns immediately.

Parameters

- **func** – callable object (e.g. a function) to be called in the background thread
- **args** (*tuple*) – passed to func when it's called
- **kwargs** (*dict*) – passed to func when it's called

When calling back into Excel and modifying the current sheet from a macro sheet equivalent function it's possible to end up with a circular dependency. Macro sheet equivalent functions with any `xl_cell` arguments are considered volatile by Excel, so any change to the worksheet causes them to be re-evaluated.

To help with this, PyXLL keeps track of the arguments and return values for worksheet functions when it's likely that a circular dependency will occur and prevents that from happening. If you decide not to use `async_call` and to use another thread yourself you need to be careful of circular dependencies.

¹ Not to be confused with asynchronous functions that return their result to Excel asynchronously. `async_call` is simply a function that queues a callable object to be called later. See also *Asynchronous functions*.

PYXLL CALLBACKS

You can register some functions to be called at certain times by PyXLL using a few decorators that may be imported from `pyxll`. This section describes those decorators.

`pyxll.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 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 ...
```

`pyxll.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 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 ...
```

```
# recalculate all open workbooks
xlcCalculateNow()
```

pyxll.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..."
```

pyxll.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...
```

DEVELOPER TOOLS (NEW IN PYXLL 1.3)

8.1 Reloading

When PyXLL is configured in *developer mode* it is possible to reload the modules imported by PyXLL using the *Reload* menu item in the PyXLL Excel addin menu.

To enable the developer mode set *developer_mode=1* in the *PYXLL* section of the config file. See the *config file* documentation for more details.

In addition to the Reload menu item two macro commands are enabled, `pyxll_reload` and `pyxll_rebind`.

These commands can be called via Excel's COM API, and so may be called from other processes outside of Excel which means they can be called by scripts invoked from an external editor.

`pyxll_reload()`

Reloads all modules listed in the PyXLL config file and updates all the Excel functions, macros and menus.

Calling this macro command is equivalent to selecting the Reload menu item.

`pyxll_rebind()`

Refreshes all the Excel function, macro and menu bindings. This can be used if additional modules have been imported since PyXLL started, or if modules have been reloaded and those modules include Excel functions that need to be updated to refer to the reloaded instances of the Python functions.

They are Excel macro commands, *not* python functions.

They may be called from VBA using the *Run* command, or from Python using the `win32com` module.

Example of calling from VBA:

```
Sub ReloadPyXLL()  
    Run "pyxll_reload"  
End Sub
```

Example of calling from Python (this may be called from *any* Python instance, not just from within Excel):

```
import win32com.client  
xl = win32com.client.GetObject("Excel.Application")  
xl.Run("pyxll_reload")
```

See the *developer tools examples* for more information.

8.2 Debugging

It should be possible to use any remote Python debugger to debug code running in Excel using PyXLL.

If you are using Eclipse with PyDev you can use PyDev's remote debugging feature to debug Python code in Eclipse. There is an example included with PyXLL that shows how, and is also available in the *developer tools examples*.

EXAMPLES

All of the following examples are included in the package available for download.

9.1 Simple worksheet functions

These examples show how to expose Python functions as Excel user defined functions that will appear in the Excel function wizard and be callable from Excel worksheet.

They also show some basic usage of the custom type functionality showing how non-standard types can be coerced into types Excel understands.

See also *Writing User Defined Functions (UDFs)*.

```
"""
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

#
# xl_func takes a string argument that is the signature of
# the function to be exposed to excel. This example takes
# three integers and returns an integer.
#

@xl_func("int x, int y, int z: int")
```

```

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

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

#
# 2) The var type
#

#
# Another 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.
#

@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) Arrays
#
#
# Arrays in PyXll are 2d arrays that correspond to the grid in
# Excel. In python, they are represented as lists of lists.
# Arrays of any type can be used, and the var type may be
# an array of vars.
#
# Arrays of floats are more efficient to marshall between
# python and Excel than other array types so should be used
# when possible instead of var.
#
# NumPy arrays are also supported. For those, see the
# next section.
#
@xl_func("float[] x: float")
def array_pyxll_function_1(x):
    """returns the sum of a range of floats"""
    total = 0.0
    # x is a list of lists - iterate through the rows:
    for row in x:
        # each row is a list of floats
        for element in row:
            total += element
    return total

#
# Functions can also return 2d arrays as lists of lists
# in python. These can be used as array formulas in excel
# to return a grid of data.
#
@xl_func("string[] array, string sep: string[]")
def array_pyxll_function_2(x, sep):
    """joins each row by 'sep' and returns a column of strings"""
    # result is a list of lists of strings
    result = []
    for row in x:
        s = sep.join(row)
        # the result is just one column wide
        result_row = [s]
        result.append(result_row)
    return result

#
# the var type may also be used to pass and return arrays, but
# the python function should do any necessary type checking.
#
@xl_func("var x: string[]")

```

```

def array_pyxll_function_3(x):
    """returns the types of the elements as strings"""
    # x may not be an array
    if not isinstance(x, list):
        return [[type(x)]]

    # x is a 2d array - list of lists.
    return [[type(e) for e in row] for row in x]

#
# var arrays may also be used
#

@xl_func("var[] x: string[]")
def array_pyxll_function_4(x):
    """returns the types of the elements as strings"""
    # x will always be a 2d array - list of lists.
    return [[type(e) for e in row] for row in x]

#
# 4) NumPy arrays
#
# the numpy_array type corresponds to the numpy.ndarray
# type.
#
# You must have numpy installed to be able to use the
# numpy_array type.
#

@xl_func("numpy_array x: numpy_array")
def numpy_array_function_1(x):
    # return the transpose of the array
    return x.transpose()

@xl_func("numpy_array<float_nan> x: numpy_array<float_nan>")
def numpy_array_function_2(x):
    # simply return the array to demonstrate how errors from
    # excel may be passed to python as NaN
    return x

#
# As well as 2d arrays, 1d rows and columns may also be used
# as argument and return types.
#

@xl_func("numpy_row x: string")
def numpy_row_function_1(x):
    return str(x)

@xl_func("numpy_row x: numpy_column")
def numpy_row_function_2(x):
    return x.transpose()

@xl_func("numpy_column x: string")
def numpy_col_function_1(x):
    return str(x)

@xl_func("numpy_column x: numpy_row")

```

```

def numpy_col_function_2(x):
    return x.transpose()

#
# 5) 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

#
# 6) 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.
#

@xl_func("xl_cell cell : string")
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)
```

9.2 Custom types

This example shows some basic usage of the custom type functionality that demonstrates how non-standard types can be coerced into types Excel understands.

See also *Custom types* and *Object cache*

```
"""
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

#
# 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 your 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 registered 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]))

```

9.3 Menu functions

These examples show how to create Excel menu items that call your Python code when selected.

See also *Creating custom menu items*.

```
"""
```

```
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 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, xlAlert

#
# 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():
    xlAlert("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():
    xlAlert("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("Wow, a different menu!")

#
# 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=20)
def on_example_submenu_item_1():
    xlcAlert("whoa, a sub-menu!")

#
# A simple menu item to show how to get the PyXLL config
# object and open the log file.
#

@xl_menu("Open log file", sub_menu="More Examples")
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)

```

9.4 Macros and Excel Automation

These examples show how to register macros that can be attached to Excel GUI object and how call back into Excel from Python using the `win32com` module.

See also *Writing Excel macros* and *Calling back into Excel from Python*.

```

"""
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__)

try:
    import win32com.client
except ImportError:
    _log.warning("*** win32com.client could not be imported      ***")
    _log.warning("*** some of the automation examples will not work ***")
    _log.warning("*** to fix this, install the pywin32 extensions.  ***")

#
# Getting the Excel COM object
#
# PyXLL has a function 'get_active_object'. This returns
# a PyIDispatch object for the Excel window instance.
# 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.
#
# The window object can be wrapped as a
# win32com.client.Dispatch object to make it
# easier to use, as shown in these examples.
#
# 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.
```

```

#
def xl_app():
    """returns a Dispatch object for the current Excel instance"""
    # get the Excel application object from PyXLL and wrap it
    xl_window = pyxll.get_active_object()
    xl_app = win32com.client.Dispatch(xl_window).Application

    # it's helpful to make sure the gen_py wrapper has been created
    # as otherwise things like constants and event handlers won't work.
    win32com.client.gencache.EnsureDispatch(xl_app)

    return 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.xlAlert("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. Macro functions with var argument
# types are considered volatile by Excel and so this can
# lead to a circular dependency in the calculation graph.
# PyXLL takes care of this for you if you use the async_call
# function, but if you use your own thread to call back to
# Excel you have to take care not to cause an infinite loop.
#

@xl_func("int rows, int cols, var value", 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 the macro sheet function xlfCaller
    address = pyxll.xlfCaller()

    # 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

```

9.5 PyXLL callbacks

It's possible to register functions for PyXLL to call at certain points. These examples show how to use each of the callback decorators.

See also *PyXLL callbacks*.

```
"""
```

```
PyXLL Examples: Callbacks
```

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

Modules 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, \
                    xlcCalculateNow

import logging
_log = logging.getLogger(__name__)

try:
    import win32api
    import win32con
except ImportError:
    _log.warning( "*** win32api could not be imported          ***")
    _log.warning( "*** the callback examples will not work correctly ***")
    _log.warning( "*** to fix this, install the pywin32 extensions ***")
    win32api = None

@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
        if win32api:
            win32api.MessageBox(0,
                                "\n".join(errors) + "\n\n(See callbacks.py example)",
                                "PyXLL Callbacks Example",
                                win32con.MB_ICONWARNING)
        else:
            _log.info("callbacks.on_open: " + "\n".join(errors))

@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
        if win32api:
            win32api.MessageBox(0,
                                "\n".join(errors) + "\n\n(See callbacks.py example)",
                                "PyXLL Callbacks Example",
                                win32con.MB_ICONWARNING)
        else:
            _log.info("callbacks.on_reload: " + "\n".join(errors))
    else:
        # report everything reloaded OK
        if win32api:
            win32api.MessageBox(0,
                                "PyXLL Reloaded OK\n(See callbacks.py example)",
                                "PyXLL Callbacks Example",
                                win32con.MB_ICONINFORMATION)
        else:
            _log.info("callbacks.on_reload: PyXLL Reloaded OK")

    # recalculate all open workbooks
    xlcCalculateNow()

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

    This is a good time to clean up any globals and stop
    any background threads so that the python interpreter
    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))
    else:
        _log.info("callbacks.license_notifier: "
                  "This copy of PyXLL is for evaluation or non-commercial use only")

```

9.6 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 retrieves 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.

See also *Custom types* and *Calling back into Excel from Python*.

```

"""
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.
"""

from pyxll import xlfcaller, \
                xl_arg_type, \
                xl_return_type, \
                xl_func, \
                xl_on_close, \
                xl_on_reload

import pyxll

import logging
_log = logging.getLogger(__name__)

#
# win32com and automation.xl_app are 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 = {}

    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 update(self, workbook, sheet, cell, value):
        """updates the cached value for a workbook, sheet and cell and returns the cache id"""
        obj_id = self._get_obj_id(value)

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

        _log.debug("Adding entry %s to cache at (%s, %s, %s)" % (obj_id, workbook, sheet, 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, sheet, cell)] = None

# update the cache of cells to object ids
self.__cells.setdefault(workbook, {}).setdefault(sheet, {})[cell] = obj_id

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

def get(self, obj_id):
    """
    returns an object stored in the cache by the object id returned
    from the update method.
    """
    try:
        return self.__objects[obj_id][0]
    except KeyError:
        raise ObjectCacheKeyError(obj_id)

def delete(self, workbook, sheet, cell):
    """deletes the cached value for a workbook, sheet and cell"""
    try:
        obj_id = self.__cells[workbook][sheet][cell]
    except KeyError:
        # nothing cached for this cell
        return

    _log.debug("Removing entry %s from cache at (%s, %s, %s)" % (obj_id, workbook, sheet, 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, sheet, cell)]
    if not referring_cells:
        del self.__objects[obj_id]

    # remove the entries from the __cells dict
    wb_cache = self.__cells[workbook]
    ws_cache = wb_cache[sheet]
    del ws_cache[cell]
    if not ws_cache:
        del wb_cache[sheet]
    if not wb_cache:
        del self.__cells[workbook]

def delete_all(self, workbook, sheet=None, predicate=None):
    """
    deletes all references in the cache by workbook, worksheet.
    If predicate is not None, the cells will only be deleted if
    predicate(cell, obj_id) returns True
    """
    wb_cache = self.__cells.get(workbook)
    if wb_cache is not None:
        if sheet is not None:
            sheets = [sheet]
        else:

```

```

        sheets = wb_cache.keys()

        for sheet in sheets:
            ws_cache = wb_cache.get(sheet)
            if ws_cache is not None:
                cached_cells = ws_cache.items()
                for cell, obj_id in cached_cells:
                    if predicate is None or predicate(cell, obj_id):
                        self.delete(workbook, sheet, 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", macro=True, 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
    """
    # 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 cell in [book]sheet!address format
    caller = xlfCaller()

    # split the cell up into workbook, sheet and cell
    assert "!" in caller, "Calling cell not in [book]sheet!address format: %s" % caller
    wb_and_sheet, cell = caller.split("!", 1)
    wb_and_sheet = wb_and_sheet.strip("'")

    assert wb_and_sheet.startswith("[") and "]" in wb_and_sheet, \
        "Calling cell not in [book]sheet!address format: %s" % caller
    workbook, sheet = wb_and_sheet.strip("[").split("]", 1)
    while "'" in sheet:
        sheet = sheet.replace("'", "")

    # 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
    return _global_cache.get(x)

#
# Example worksheet functions using the object cache
#
# The following examples show how worksheet functions using
# xl_func can use the new 'cached_object' type registered
# above to return and take python objects cached by the
# object cache (appear to be cached on the excel grid).
#

@xl_func(": int", volatile=True)
def cached_object_count():
    """returns the number of cached objects"""
    return len(_global_cache)

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

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

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

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

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

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

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

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

```

```

        total += sum(row)
    return total

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

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

@xl_func("float[]: cached_object")
def make_datagrid(x):
    """returns a MyDataGrid object"""
    return MyDataGrid(x)

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

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

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

#
# 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 respond 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.
    """

    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:
        func = lambda *args, **kwargs: None
        setattr(cls, name, func)

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.delete_all(workbook=str(workbook.Name))

            # 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 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, handler in self.__wb_event_handlers.items():
            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 no 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 and 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, range):
        # delete all the cells from the cache where the cell is in range
        # and the current value is not the cached object id
        def check_cell(cell, obj_id):
            # check this cell is in the range that's changed
            cell = sheet.Range(cell)
            if range.Find(cell) is None:
                return False
            # check the cell's value has changed from obj_id
            return str(cell.Value) != obj_id

        if self.__cache is not None:

```

```

        self.__cache.delete_all(str(self.Name), str(sheet.Name), predicate=check_cell)

def _xl_app():
    """returns a Dispatch object for the current Excel instance"""
    # get the Excel application object from PyXLL and wrap it
    xl_window = pyxll.get_active_object()
    xl_app = win32com.client.Dispatch(xl_window).Application

    # it's helpful to make sure the gen_py wrapper has been created
    # as otherwise things like constants and event handlers won't work.
    win32com.client.gencache.EnsureDispatch(xl_app)

    return xl_app

_event_handlers = {}
def _setup_event_handler(cache):
    # only setup the app event handler once
    if cache not in _event_handlers:
        app_handler = win32com.client.DispatchWithEvents(_xl_app(),
                                                         ObjectCacheApplicationEventHandler)

        app_handler.set_cache(cache)
        _event_handlers[cache] = app_handler

@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

```

9.7 Developer Tools

9.7.1 Debugging with Eclipse and PyDev

This example shows how to attach the PyDev interactive debugger to Python code running in PyXLL.

This module must be loaded by PyXLL by adding it to the modules list in the pyxll config.

```

"""
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

_log = logging.getLogger(__name__)

## UPDATE THIS TO MATCH YOUR VERSION OF PYDEV
_pydev_src = r"plugins\org.python.pydev.debug_2.3.0.2011121518\pysrc"

## UPDATE THIS TO WHERE YOU HAVE ECLIPSE INSTALLED
_eclipse_root = r"C:\Program Files\Eclipse"

# add the pysrc to the path for importing pydev
sys.path.append(os.path.join(_eclipse_root, _pydev_src))

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

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

```

        raise

        # this creates a menu item and a macro from the same function
        @xl_menu("Attach to PyDev")
        @xl_macro()
        def attach_to_pydev():
            pydevd.settrace()

except ImportError:
    pass

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

```

9.7.2 Reloading and importing modules

This example shows how to use the `pyxll_reload` and `pyxll_rebind` commands from outside Excel to reload and import modules in PyXLL. This can be useful when combined with calling this or a similar script from an editor.

This module must be loaded by PyXLL by adding it to the modules list in the `pyxll` config.

```

"""
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>
"""

# pywin32 must be installed to run this script
import win32com.client
import sys
import os
import cPickle

def main():
    # 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("string filename: string")
    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())
```

INDEX

A

address (pyxll.XLCell attribute), 12
async_call() (in module pyxll), 21

F

formula (pyxll.XLCell attribute), 12

G

get_active_object() (in module pyxll), 20
get_config() (in module pyxll), 20
get_dialog_type() (in module pyxll), 20
get_type_converter() (in module pyxll), 11

N

note (pyxll.XLCell attribute), 12

P

pyxll_rebind (C function), 24
pyxll_reload (C function), 24

V

value (pyxll.XLCell attribute), 12

X

xl_arg_type() (in module pyxll), 9
xl_func() (in module pyxll), 6
xl_license_notifier() (in module pyxll), 23
xl_macro() (in module pyxll), 16
xl_menu() (in module pyxll), 14
xl_on_close() (in module pyxll), 23
xl_on_open() (in module pyxll), 22
xl_on_reload() (in module pyxll), 22
xl_return_type() (in module pyxll), 10
xl_version() (in module pyxll), 20
xlAsyncReturn() (in module pyxll), 19
xlAlert() (in module pyxll), 19
xlCalculateDocument() (in module pyxll), 19
xlCalculateNow() (in module pyxll), 19
xlCalculation() (in module pyxll), 19
XLCell (class in pyxll), 11
xlCaller() (in module pyxll), 18

xlGetDocument() (in module pyxll), 18
xlGetWindow() (in module pyxll), 18
xlGetWorkbook() (in module pyxll), 18
xlGetWorkspace() (in module pyxll), 18
xlWindows() (in module pyxll), 19