

Open Application Development

Developing macros

Version 2.0

Macros, Part 2 - Developing macros

Contents

Developing macros	1
Version 2.0.....	1
Introduction.....	4
Macro Editor Window	5
The Menu Bar	6
The Icon Bar.....	7
The Button Bar.....	7
The Code Window	8
The Watch and Message Area.....	9
The Macro Tool.....	11
Macro Examples	12
Our First Macro.....	12
Getting Acquainted with The Code Window.....	13
Python – Short Introduction.....	15
Python’s Features.....	15
Python Syntax.....	16
Strings in Python.....	17
Flow Control Statements.....	17
Moving Beyond Cut and Paste	23
On Classes and Objects.....	26
Collections	29
Documents.....	29

Macros, Part 2 - Developing macros

ZEN Macro Environment	31
Acquisition	31
Application.....	33
Devices.....	37
Windows.....	38
Processing.....	39
Measurement	42
Analysis.....	43
ZEN Macro Classes.....	45
ZenImage - Methods	46
ZenImage - Properties	47
Multidimensional Images	49
Including Code and Importing Functionality	52
Now you tell me.....	53
Inter-process communication	55
Appendix A – Configuring MTB For the Simulation Mode	58

Macros, Part 2 - Developing macros

Introduction

This documentation will be of interest to users who have mastered the recording and execution of recorded macros (see “Macros - Recording, using and modifying macros”), but miss some features, essential for their problem and its solution, but not available via simple recording of steps. A typical example would be logical structures, such as if-else and loop clauses, which one cannot implement in a simple recording environment.

This document includes a short introduction into the Python¹, furthermore an introduction into the object-oriented programming, covering the basics about classes, their methods, and their properties. This should help the reader master the use of the simple interface, i. e. the framework of ZEN classes, available via the Macro editor functionality. The document ends with an example of using ZEN as a client: using these examples the user will be able to gain access to other .NET services, for instance the Microsoft Office functionality.

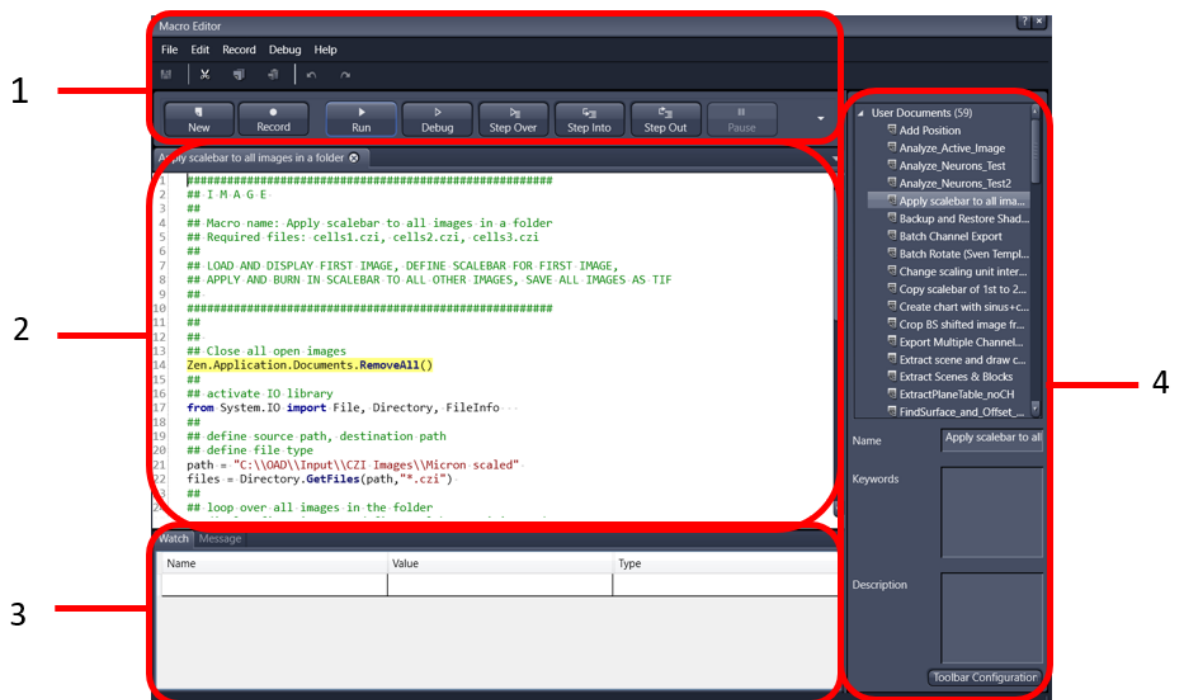
¹ The Python version, used in ZEN, is the IronPython 2.7.11.1000 (2.7.11). see <http://ironpython.net> for specifics.

Macros, Part 2 - Developing macros

Macro Editor Window

Select *Macro > Macro Editor ...* in the ZEN menu to open the Macro Editor window. It combines the following features of a typical integrated development environment (IDE):

- **Source code editor:** it allows you to record, enter and change the macro source code. To simplify and speed up input of source code it incorporates features such as support for *autocomplete* - predicting a word or phrase that the user wants to type - and *pick lists* - i.e. lists of alternative functions, particularly their parameter lists, for the current context. These features appear under the common name *Intellisense* (short for "intelligent sense" or "intelligent code sense") in several applications, for instance in Microsoft Visual Studio.
- **Python interpreter:** the active macro is executed directly in the embedded Python interpreter.
- **Debugger:** helps the user find and fix errors in the created code.



The Macro Editor window includes following parts:

1. The menu and the icon bar, with functions such as to open and save the macro files, to edit and record macros and to run and debug them.

Macros, Part 2 - Developing macros

2. The code window shows the currently active macro, in the above example “Apply scalebar to all images in a folder”
3. The area for messages and debug information
4. Right hand macro area with two folders of available macros: **User Documents** and **Workgroup Documents** - same as the macro window in the ZEN’s right hand tool area.

The Menu Bar



The menu offers the following functions:

File:

New Macro	Opens a new macro in the code window.
Import...	Import an outside macro file in the code window.
Save	Saves the current macro, if it has changed (Ctrl+S).
SaveAs...	Saves the current macro under a different name.
Rename...	Renames the current macro.
Delete	Deletes the current macro.

Edit:

Cut	Cuts out the selected parts of the macro (Ctrl+X, Shift+Del).
Copy	Copies the selected text into the clipboard (Ctrl+C, Ctrl+Ins).
Paste	Pastes the clipboard contents into the macro (Ctrl+V, Shift+Ins).
Find	Finds the string entered (Ctrl+F).
Replace	Replaces the string (Ctrl+H).
Undo	Undo the last operation (Ctrl+Z).
Redo	Redo the operation undone (Ctrl+Y).

Note: the edit commands are standard and so are their shortcuts

Record:

Record	Starts recording.
Stop Recording	Stops recording if recording has been activated

Note: Record and Stop Recording have the function, similar to “Record” and “Stop” buttons in the macro window of ZEN’s right hand tool area. When the button Stop

Macros, Part 2 - Developing macros

Recording has been pressed, the code recorded is added to the end of the current macro code.

The Icon Bar



From left to right you have the following functions available:



Save the macro



Cut the selected text



Copy the selected text



Paste the clipboard text

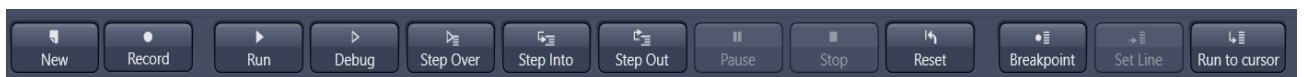


Undo



Redo


The Button Bar



New	Opens a new macro in the code window.
Record	Starts/stops recording the steps.
Run	Executes the macro (Ctrl+F5).
Debug	Executes the macro in the debug mode (F5).
Step Over	Executes the next step in the macro (F10). If the next step is a Python function, executes it.

Macros, Part 2 - Developing macros

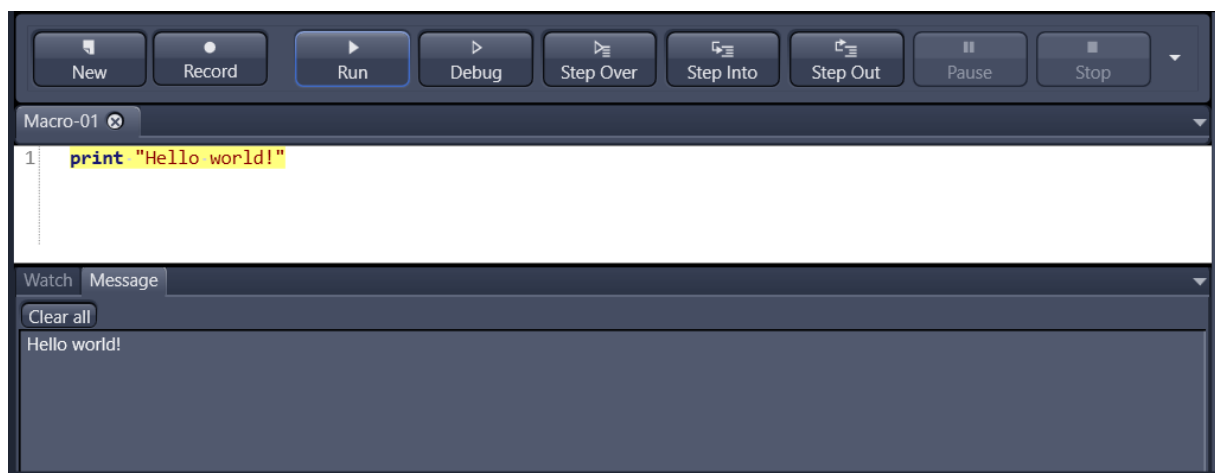
Step Into	Executes the next step in the macro (F11). If it is a Python function, it enters the function and executes its first step.
Step Out	Executes the rest of the current Python function without stepping line by line.
Pause	Pauses the execution of the macro.
Stop	Stops the execution of the macro.
Reset	Resets the stopped/paused macro.
Breakpoint	Sets/turns off a breakpoint at the current line (F9).
Set Line	Sets the line to execute next (F8).
Run to cursor	Continues the execution until the position of the cursor is reached.

Note: you obtain the whole bar by increasing the width of the Macro Editor or by clicking the icon  in the right hand area.

The Code Window

Time to write your first macro, and of course, it will be the hallowed “Hello world”:

- Click **New**
- Type print “Hello world!” into the code window
- Click **Run**
- Activate **Message** tab in the Watch and Message area

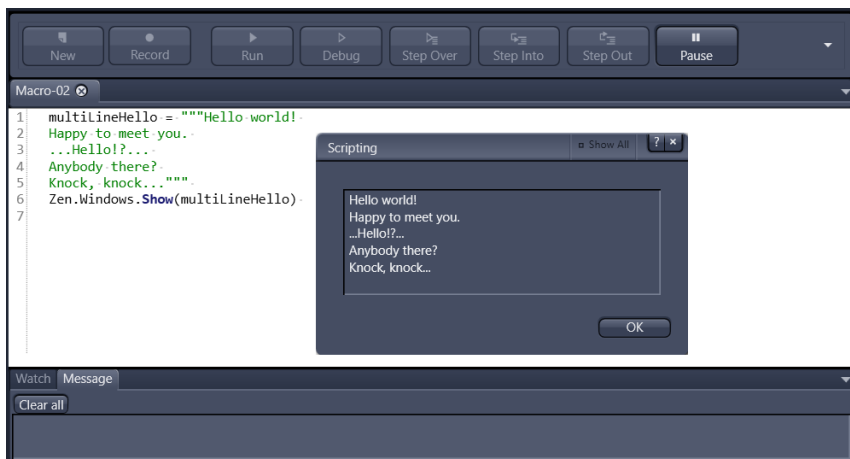


Macros, Part 2 - Developing macros

The command print will be used frequently in the examples below. In some cases, we will want to display something and wait for the user to react – a typical case for a standard message box, as in the example below:

```
multiLineHello = """Hello world!
Happy to meet you.
...Hello!?...
Anybody there?
Knock, knock..."""
Zen.Windows.Show(multiLineHello)
```

After executing the macro, the following will be seen in ZEN:



The Watch and Message Area

Cut and paste the following macro into the code window:

```
N = 6
F = 1
for i in range(1, N+1):
    F = F * i
    print i, F
ThisIsTheEnd = "My friend"
```

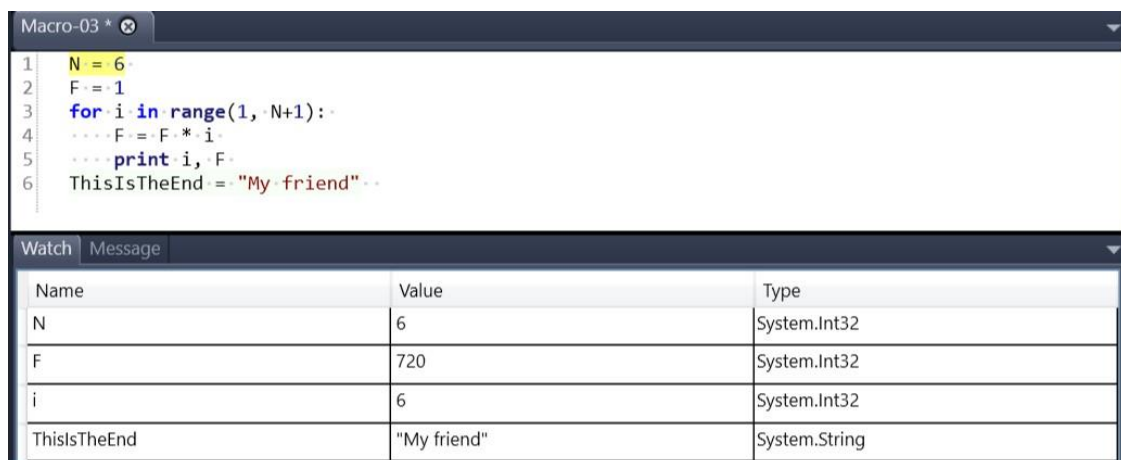
Macros, Part 2 - Developing macros

*Note: the periods at the beginning of the code lines (e.g.F = F * i) indicate the indent. If you type the above code into the macro window, use the tabulator key instead of spaces. Cutting the code from above and then pasting it into the macro editor window is of course simpler and safer.*

- Select the **Watch** tab in **Watch & Message** area
- Highlight **N** in the first line and right-click for the dropdown menu. Select **Add Watch**



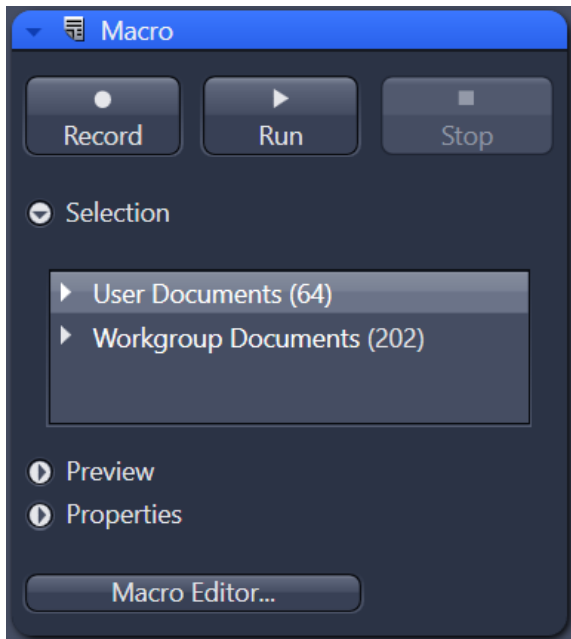
- Repeat the above step for **F** in second line, **i** in the next line and **ThisIsTheEnd** in the last line.
- Click **Reset** and **Run** – or use **F10** to step through the macro while watching **Watch** tab contents
- When the macro stops on the last line, the **Watch** tab contents will look like this:



Macros, Part 2 - Developing macros

- Activate the **Message** tab. It contains six rows (from 1 1 to 6 720) as expected.

The Macro Tool



The Macro tool in the Right Tool Area contains everything you may need to record and run macros. Use the *View > Show Macro Environment* to display/hide it. See Part 1 – recording, using and modifying macros for documentation.

Macros, Part 2 - Developing macros

Macro Examples

Our First Macro

The simplest way to learn about the code window is to start using it. Here is a simple job to do: load 3 images from the disk:

- Press **New**
- Press **Record**
- Select **Windows > Close All** to clear all files in document area
- Select first image from the disk (*File > Open*)
- Select second image from the disk
- Select third image from the disk
- Press **Stop Recording**

The code should look like this:

```
# ***** Recorded Code Block *****

Zen.Application.Documents.RemoveAll()

image1 = Zen.Application.LoadImage ("pic1.czi", True)
Zen.Application.Documents.Add (image1)

image2 = Zen.Application.LoadImage ("pic2.czi", True)
Zen.Application.Documents.Add(image2)

image3 = Zen.Application.LoadImage ("pic3.czi", True)
Zen.Application.Documents.Add(image3)

# ***** End of Code Block *****
```

Instead of pic1.czi, pic2.czi and pic3.czi your macro will of course show the file names of images you have selected.

You may be curious about all those terms and functions, like Zen, Add, LoadImage. The intention of the macro recorder is – similar to the Speed Dial Codes on the telephone, which spare you memorizing 15-digit telephone numbers. Rest assured we will come to Zen, Add, LoadImage a little later for more explanation.

Macros, Part 2 - Developing macros

Getting Acquainted with The Code Window

Here are a few simple tasks to get acquainted with the code window:

1. Saving and loading:
 - a. Save the macro with **File > Save, Ctrl+S** or by clicking on the “**save macro**” icon (i.e. the left-most “floppy” icon in the icon bar).
 - b. Rename the saved macro (for instance call it “Load Three Images”).
 - c. Load the macro again by clicking on it in the **User Documents** or **Workgroup Documents** list in the right hand macro area.
 - d. Click **Run**.
2. Executing single command:
 - a. Click on the line with “image1”.
 - b. Click on the **Set Line** button or press **F8**. The line is high-lighted with yellow background: this is now the current line.
 - c. Click **Step over** (or press **F10**) and select a new, not yet loaded image.
 - d. Click **Step over** (or press **F10**) again to add the selected image to Documents. You should now have four images opened in the document area of ZEN.
 - e. Click on the first line, i.e. line ending with RemoveAll() and press **F8** to activate it.
 - f. Press **F10**. All four images are now gone.
3. Starting in the middle of the macro:
 - a. Run the macro and wait for it to finish.
 - b. Click on the line with “image2”.
 - c. Click on the **Set Line** button, or press **F8**
 - d. Click on **Run** button or press **Ctrl+F5**. You will have to select two images that will be added to Documents.

The Python interpreter executes the macro one line at a time. The line to be executed next is highlighted in yellow. When you press **Reset**, the macro will start again from the top.

You can select the next line to be executed by clicking on **Set Line**.

Pressing **Stop** during the execution of the macro will stop it and highlight the top line as the next line to be executed.

Pressing **Pause** during the execution of the macro will pause it. The highlighted line means the next line to be continued when you press **Run** again.

Macros, Part 2 - Developing macros

If you want the macro to stop on a given line during the execution, select it and mark it as the Breakpoint by clicking the **Breakpoint** button.

Macros, Part 2 - Developing macros

Python – Short Introduction

The intention of this subchapter is to provide some basic concepts that the reader will need later. It is a shortened and abridged version of the “Python in 10 minutes” by Stavros korokithakis². All the examples provided can be cut and pasted into the macro editor. Some points and subjects will be introduced directly in code and only briefly commented on, so keep the ZEN macro editor on hand.

Hint: to learn the basic features of Python and some more, there is a number of excellent sites available. See <http://www.python.org/about/gettingstarted/> for some suggestions.

Python's Features

Python is implicitly, dynamically typed: you do not have to declare a variable, as its type is set down when it receives a value. The variable `i` below looks like an integer, but for Python it is a string, because the first line defines `i` as a string. So, in the second line Python will protest, as `x` is by now known as a string and cannot be divided by two:

```
i = "string"
x = i/2
```

Error just pops up when macro is running. To resolve this problem, we can add a third line: `i = 3`, to the two lines above, then the type of `i` will be changed to integer (as an example for what dynamically typed stands for):

```
i = "string"
i = 6
x = i/2
```

Python is case sensitive: `x` and `X` are two different variables.

Python is object-oriented: to Python everything is an object. So far, you have survived the starting dozen pages or so, of object-oriented programming, so it is evidently not that hard on the user – there will be more later in the chapter “On Classes and Objects”.

² Available online - <http://www.korokithakis.net/tutorials/python/> - or as Ebook on Amazon - <http://www.amazon.com/kindle-store/dp/B0052T2V06> under [Creative Commons Attribution-NonCommercialNoDerivs 3.0 Unported License](https://creativecommons.org/licenses/by-nc-nd/3.0/).

Macros, Part 2 - Developing macros

Python Syntax

Python has no mandatory statement termination characters: the end of the line is the end of statement. There is no BEGIN END or {} brackets for blocks: you specify blocks by indentation: indent to begin a block, dedent to end one. The statements, like loops, if statement, function definitions, that expect an indentation level, end with a colon:

```
for k in range(5):  
    print k
```

Note that the loop will run 5 times, as indicated in the range argument (and k will start at 0 - there will be more on loops later). Comments start with the pound (#) sign and are single-line:

```
# Here a similar example for a loop  
# You will meet again and again in ZEN  
# Files opened in the ZEN interface  
docs = Zen.Application.Documents  
if (docs.Count == 0):  
    print "no documents available"  
else :  
    # Get the number of files opened  
    print "number of documents: ", docs.Count  
  
# Get the filename using a loop  
for i in range(Zen.Application.Documents.Count):  
    print Zen.Application.Documents[i].Name
```

Values are assigned with the equals sign (i = "string"), and so are objects – in fact, objects are bound to the names given, Zen.Application.Documents (an object, to be discussed shortly) has been given the name docs. Note the statement if (docs.Count == 0) above: items are tested for equal using double equal signs. See more on the subject in the subchapter Flow Control statements in the following text.

Macros, Part 2 - Developing macros

Strings in Python

Its strings can use either single or double quotation marks, and you can have quotation marks of one kind inside a string that uses the other kind (i.e. "He said 'hello'." is valid). Multiline strings are enclosed in triple double (or single) quotes ("""). Python supports Unicode out of the box, using the syntax u"string in Unicode". A "raw string literal" (r"...") is a slightly different syntax for a string literal, in which a backslash "\ " is taken as meaning "just a backslash", for instance r"c:\program files\windows". In normal string literals, each backslash must be doubled up to avoid being taken as the start of an escape sequence, forcing one to write "c:\\program files\\windows".

To fill a string with values, you use the % operator and a tuple. Each %s gets replaced with an item from the tuple, left to right, and you can also use dictionary substitutions – see the last example below:

```
name = "Joe"
print "Name: %s\nNumber: %s\nString: %s" % (name, 6, "pack")
print "-----"
print "together: %s%s%s" % (name, 6, "pack")
print "-----"
strString = """This is a 'multiline' string."""
print strString
print "-----"
print "konichiwa wakarimashita"
print u"こんにちは 分かりました"
print "-----"
# WARNING: Watch out for the trailing s in "%(key)s".
print "This %(verb)s a %(noun)s." % {"noun": "test", "verb": "is"}
```

Python uses the \ character to denote escape sequences, such as \n in the second line above for a new line, or \t for a tab. Use \\ to denote the back slash itself, e. g. tempDir = "c:\\temp". You can of course use the raw string literal convention, if applicable.

Python has a number of functions to manipulate strings. See the chapter "On Classes and Objects".

Flow Control Statements

Flow control statements are **if**, **for**, and **while**. Use **for** to enumerate through members of a one-dimensional array or a list:

Macros, Part 2 - Developing macros

```
for number in range(10):
    # Check if number is one of the numbers in the tuple.
    if number in (7, 9):
        # 'break' stops the loop without executing the "elif...else" clause.
        print "number ", number, " in (7, 9) .... break"
        break
    elif number in (2, 3, 4, 5):
        # 'continue' skips a single iteration and starts the next iteration in the
loop
        print "number ", number, " in (2, 3, 4, 5)"
        continue

    else:
        # 'pass' is just a placeholder, does nothing
        pass

print "Done"
```

The pass statement (the second line from the bottom) is just a placeholder, does nothing. It can be used when a statement is required syntactically but the program requires no action. To obtain a list of numbers, use range(<number>). There is no **select**; use **if ... elif ...** instead. You can access array ranges using a colon (:). Leaving the start index empty assumes the first item, leaving the end index assumes the last item. Negative indices count from the last item backwards (thus, -1 is the last item).

Macros, Part 2 - Developing macros

```
# define a one-dimensional array
dwarfs = ("sleepy", "happy", "grumpy", "dopey", "bashfull", "sneezy", "doc")
# print the first item
print dwarfs[0]
# print all items before the item at index 3 (the item at index 3 not included)
print dwarfs[:3]
# print items between index 2 and index 5
# include the item at index 2, but the item at index 5 not included
print dwarfs[2:5]
# print all items after the item at index 3 (include the item at index 3)
print dwarfs[3:]

# print the last item
print dwarfs[-1]
# count from the last item (i.e. index -1) backwards
# print the items between index -5 and index -2
print dwarfs[-5:-2]
# count from the last item backwards
# print all items after the item at index -3 (the item at index -3 not included)
print dwarfs[:-3]
```

The example above uses a one-dimensional array. In the examples below, lists are used. While arrays are initialized with the (...) construct, e.g. (3, 4, 7, 9), the lists are initialized with [...]:

```
# Define a list
mylist = ["List item 1", 2, 3.14]
print mylist

# Update the item at index 0 with the string "List
item 1 again"
mylist[0] = "List item 1 again"
print mylist

# Update the item at last index with the value 6.28
mylist[-1] = 6.28
print mylist
```

Macros, Part 2 - Developing macros

After this excursion into uncharted waters, the following examples, using arrays and lists, will be welcome. The first example shows, how to combine a list of filenames with - by now familiar, to be explained in detail later - items like `Zen.Application.LoadImage` and `Zen.Application.Documents.Add`

```
# Define a one-dimensional array for images to be loaded
myImages = ("Cells1.czi", "Cells2.czi", "Cells3.czi")
# Set the folder path for images
rootDir = "C:\\OAD\\Input\\CZI Images\\Micron Scaled\\"
print myImages

# Loop over all images in the folder
for imgName in myImages:
    # Load the image
    image = Zen.Application.LoadImage(rootDir + imgName)
    # Show the image in the document area
    Zen.Application.Documents.Add(image)
```

The second example is a little more complex: it uses the list of extensions to create an array of filenames in the specified folder. The array is then sorted and used to load images in the folder in their alphabetical order:

Macros, Part 2 - Developing macros

```
# Makes System items Directory and FileInfo known to the macro
from System.IO import Directory, FileInfo

# Get the path of the Images folder
root = Zen.Application.Environment.GetFolderPath(ZenSpecialFolder.Images)
exts = [".bmp", ".zvi", ".jpg", ".czi"]
files = []
for ext in exts:
    # Get the images with the requested extensions from the folder
    currFiles = Directory.GetFiles(root, ext)
    for file in currFiles:
        files.append(file)
# Remove all files in the document area
Zen.Application.Documents.RemoveAll(False)
# Sort all images in the alphabetic order
sortedNames = sorted(files)
for file in sortedNames:
    print file
    # Load all images in the alphabetic order
    Img = Zen.Application.LoadImage(file, False)
    # Show the image in the document area
    Zen.Application.Documents.Add(Img)
```

*Note: you may wonder about functions like “**from** System.IO **import** Directory, FileInfo”, “Zen.Application.Environment.**GetFolderPath**” and “Directory.**GetFiles**”. We are jumping the gun here a little – see the subchapter “Importing” below.*

Importing

External libraries can be added with the import [libname] keyword. You can also use from [libname] import [funcname] for individual functions. Here is an example:

```
# The classes Directory and FileInfo have been imported from the System.IO
from System.IO import Directory, FileInfo

# Get the path of the Images folder
root = Zen.Application.Environment.GetFolderPath(ZenSpecialFolder.Images)
ext = ".czi"
# Get all images with the ".czi" extension
files = Directory.GetFiles(root, ext)
```

Macros, Part 2 - Developing macros

```
if (files.Length == 0):
    print "no files in " + root + " with extension " + ext
else :
    for i in range(0,files.Length):
        # Get the whole path and filename of an image by using FileInfo()
        flInfo = FileInfo(files[i])
        print flInfo
        print flInfo.Name, "\t", flInfo.DirectoryName, "\t", flInfo.Length
        print
```

In the above case, the classes **Directory** and **FileInfo** have been imported from the **System.IO**. As they belong to the intrinsic **.NET** classes, you will notice that code completion is available for them.

Conclusion

This tutorial has never intended to be exhaustive. Python offers much more, but the hope is, that by now you will be able to discover it all at your own leisure. We do hope that these introductory chapters have made your transition in Python easier.

Macros, Part 2 - Developing macros

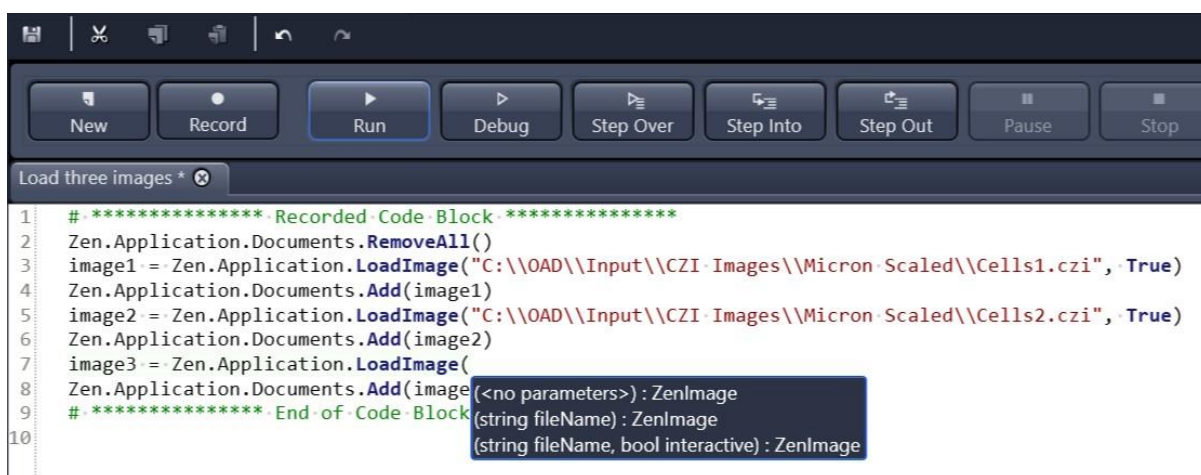
Moving Beyond Cut and Paste

Until now, we have just cut and pasted. Let us start using the keyboard for entering some text as well.

Changing function parameters:

- Load the macro, created in the chapter “Our First Macro” that loads three images (see below for its contents).
- Press **Ctrl+H/Ctrl+F**, the “Find And Replace” window opens
- Enter **True** in “Text to find”, and **False** in “Replace with”:
- Click on “Replace all”
- Click on **Run** - the three images will be loaded without the “Open image” dialog
- Press **Ctrl+Z** in the edit window to change **False** back to **True**
- Click on **Run** – the macro runs as before

The parameter **True** or **False** at the end of lines with **LoadImage** is evidently to tell, how to load images: interactively or automatically. This is also what the autocomplete help for this command can indicate (note the Boolean parameter *interactive* below):



Hint: to see the autocomplete text field, go to line 3, then delete and type again the opening parenthesis after LoadImage. You can of course also start on a fresh empty line, type

image1 = Zen.Application.LoadImage(

and see what happens, when you get to the opening parenthesis

Macros, Part 2 - Developing macros

The pop-up field that appears, explains the alternatives the user has at this point. The one taken during recording is evidently the last one the three:

..... (String filename, Boolean interactive): **ZenImage**

The command expects a string as the first parameter (a text, enclosed in double quotes) and a logical variable (**True/False**) as the second parameter and returns an image - strictly speaking an object of **ZenImage** class.

You are invited to experiment now with the other two alternatives. Note that macro recorder is smart enough to record the most expansive alternative. Later you can just eliminate the unnecessary parts or change them to suit you. If you trim down the original code to the top alternative – i.e. **LoadImage()** without any parameters - the macro will shrink considerably to something like this:

```
Zen.Application.Documents.RemoveAll()

image1 = Zen.Application.LoadImage()
Zen.Application.Documents.Add(image1)

image2 = Zen.Application.LoadImage()
Zen.Application.Documents.Add(image2)

image3 = Zen.Application.LoadImage()
Zen.Application.Documents.Add(image3)
```

Save for the top **RemoveAll** command the rest looks like a repeat of the one and the same sequence of (two) commands, that load a given image and show it in the Documents area.

The following example introduces a loop, i.e. something that is not possible to do via recording:

- a. Create a new macro and call it *Load images with a loop*
- b. Copy the first three commands from the listing above
- c. Insert **for i in range(3)** : after the line with **RemoveAll** and before the line with **LoadImage**.
- d. Insert blanks or a tab at the beginning of the **LoadImage()** line.
- e. Insert blanks or a tab at the beginning of the following **Add(image1)** line
- f. Add a line **print image1.name** at the end - do not forget the indent!
- g. The macro should look like this by now

Macros, Part 2 - Developing macros

```
Zen.Application.Documents.RemoveAll()  
for i in range(3):  
    image1 = Zen.Application.LoadImage()  
    Zen.Application.Documents.Add(image1)  
    print image1.Name
```

The line **for i in range(3):** (note the colon at the end) specifies a loop, that runs three times. The code to be repeated is indicated by the indents in the lines following.

Press **Run**. You will be asked three times for a new image file. The **Message** area will show the actual names of the images you have selected.

The macro above has two variables: **i** and **image1**, **i** as a loop counter and **image1** to hold the image loaded. At the end of the macro, just one image variable still exists, but there are three images in the Documents collection. How can one address them programmatically? What means **range(3)** above? Now that we are not tethered to the recorder anymore, a lot more is possible.

Macros, Part 2 - Developing macros

On Classes and Objects

When you program macros, you work constantly with classes and objects. Let us take an image as an example: an image – any image - has certainly a name, it has a given size (i. e. width and height), can be black and white or color; it can be a multichannel image, it can include annotations of a given shape, color, position, and so on. These are the **properties** that an image – any image – will have.

Let us take again the following macro as an example:

```
Zen.Application.Documents.RemoveAll()

image1 = Zen.Application.LoadImage()
Zen.Application.Documents.Add(image1)

image2 = Zen.Application.LoadImage()
Zen.Application.Documents.Add(image2)

image3 = Zen.Application.LoadImage()
Zen.Application.Documents.Add(image3)
```

When the macro is executed, there will be three images in the documents area, each of them is an example of that ethereal, yet to be defined in detail, thing called the **image** class. If we set the watch for the items image1, image2 and image3, we will see the following in the **Watch** area:

Watch Message		
Name	Value	Type
image1	Cells1.czi	Zeiss.Micro.Scripting.ZenImage
image2	Cells2.czi	Zeiss.Micro.Scripting.ZenImage
image3	Cells3.czi	Zeiss.Micro.Scripting.ZenImage

There are three instances (the common expression for an object, created from a given class) of images available, with names image1, image2 and image3. **Zen.Micro.Scripting.ZenImage** is the class in ZEN Macro environment they stem from.

*Hint: think of a class as a cookie cutter, used to create cookies, i. e. objects as instances of the class. **ZenImage** class is the cookie cutter that ZEN uses to create objects like image1, image2 and image3. By the time you are through with this documentation, you will have met a few more cookie cutters.*

Macros, Part 2 - Developing macros

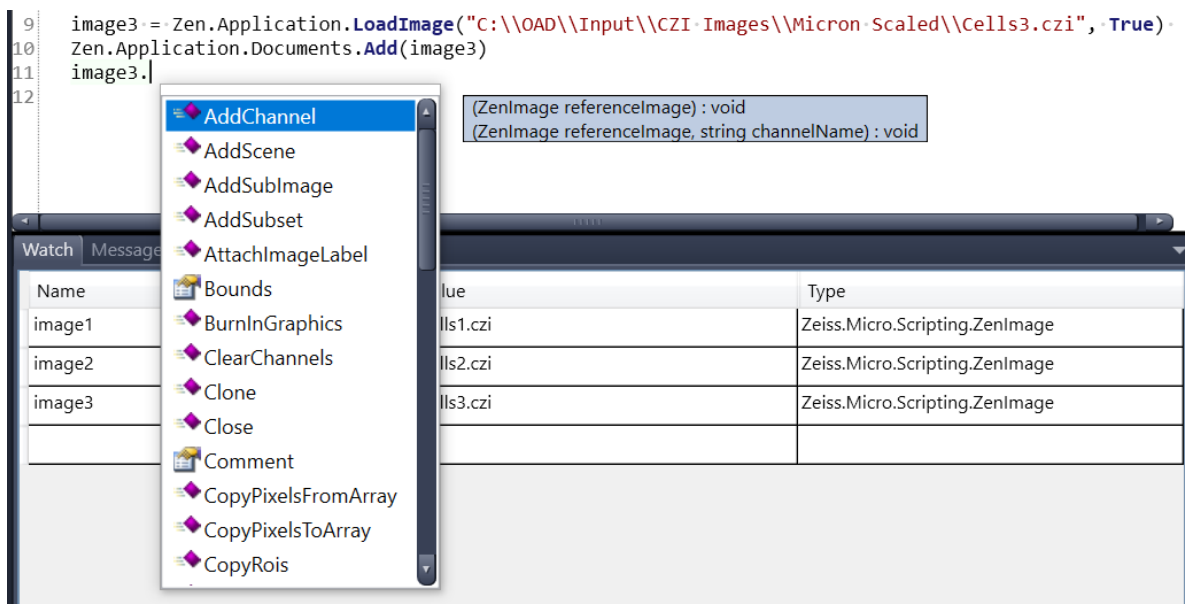
Note that the *name* of an image is just one of the so-called *properties* an image object in ZEN. Add the following two lines to the above example to print out the names of images:

```
for myImage in Zen.Application.Documents:  
    print myImage
```

The name of the **myImage** object can thus be accessed as **myImage.Name**. The convention **<object>. <something>**, as in **image.Name**, is all pervading. It simply means what it says: it is **<something>**, pertaining to the class instance **<object>**.

Taking image class as an example, size that is its width and height for starters, is certainly one of its properties. How do we get our hands on its size? How do we find out, if it is a multichannel image? In essence, what kind of **<something>** has a given object to offer? As such questions are expected to come up again and again, the Macro Editor window includes the so-called **auto completion** functionality: when the user types a name of an existing object and a period, suggestions for **<something>** will be provided. To see how it works, proceed as follows:

- Execute the example on how to load three images
- Go to the empty line at the bottom of the macro
- Type `image3.`
- A drop-down window will appear:



Macros, Part 2 - Developing macros

All the entries in the drop-down list stand for **<something>** of the **image3** object. The property **FileName** in the list is already familiar. There are more properties in the list, such as **DisplaySetting** and **Bounds**.

You can quickly identify properties based on the icon in front of them – note the two types of icons in the drop-down list. Next to the **properties** (**Bounds**, **DisplaySetting** and **FileName**), the drop-down window includes another group of entries, called **methods**, such as the **AddChannel** method at the top of the list. Methods are functions that relate directly to the object. If you scroll further down, you will hit upon the Load method for instance, that should be familiar by now.

Double-click now on the property **Bounds**: the word will get inserted after the period to read **image3.Bounds**. Type period again, and a new drop-down appears, this time for the property **Bounds**. Select **SizeX** in the new drop-down and edit the line as follows:

```
print "SizeX is ", image3.Bounds.SizeX
```

When you execute it, you should get something like “SizeX is 1024” in the Message window. You can also watch the item **image3.Bounds.SizeX** (hint: right click on mouse and activate Watch):

Watch Message		
Name	Value	Type
image3.Bounds.SizeX	2080	System.Int32

When you repeat the steps for **image2**, pretty much the same thing will happen, except that possibly the value of **image2.Bounds.SizeX** will be different. This should come as no surprise: both **image3** and **image2** are instances of the same class, i. e. of the **ZenImage** class, so they possess identical methods and properties. The values of properties can of course differ from one object to another.

Note that a property of given object can be another object: **Bounds** in **image3.Bounds.SizeX** is an instance of the **ZenBounds** class.

Looking back through the examples, you will find **Zen.Application.Documents.Add(..)** among the most popular items. The command adds a document, provided as a parameter, to the object of a **Documents** type. As the plural form of its name indicates, the **documents** object contains several objects of the type **ZenDocument**: the class **ZenDocuments** is just one typical example for the so-called **Collections** in ZEN. Collections share a number of properties and methods, described in the following chapter.

Macros, Part 2 - Developing macros

Collections

An object of a collection type contains other objects, for instance **Devices**, will contain a number – provided as the Count property – of other objects. One can add an item to a collection and remove one of them or several or all of them:

```
# Load three images
for i in range(3):
    image = Zen.Application.LoadImage()
    # Display them in document area
    Zen.Application.Documents.Add(image)
# Print the number of files opened in document area
print "before: " + format(Zen.Application.Documents.Count) + " items"

# Remove all files in document area
Zen.Application.Documents.RemoveAll(False)
# Print the current number of files opened in document area
print "after: " + format(Zen.Application.Documents.Count) + " items"
```

A collection can be enumerable, i. e. you can count one by one through it, like in the following example:

```
# Get all files in document area
myDocuments = Zen.Application.Documents
# Loop over the document area
for i in range(0, myDocuments.Count):
    # Print the name of each file in document
    area
    print myDocuments.Item[i].Name
```

Documents

Because of its visibility, **ZenDocuments** class is explained in a little more detail here. The following list includes methods, offered by the *Documents* collection, that are most often used:

Name	Description
------	-------------

Macros, Part 2 - Developing macros

Add	Adds the specified document to the open documents collection
Contains	Determines whether the open documents collection contains the specified document
GetByFileName	Gets the document by its fileName
GetByName	Gets the document by its name
Remove(ZenDocument)	Removes the specified document
RemoveAll()	Removes all documents
SaveAll	Saves all open documents

Here are the properties of the Documents collection:

Name	Description
ActiveDocument	Gets or sets the active document
Count	Gets the count of all opened documents
Item	Gets the Document at the specified index

The following example removes documents that do not have “lena” in their names from the *Documents* collection. You may find some new useful functions in the example, like extending a list (**<list>.append**) or checking for a existence of a substring within a string – (**<string>.contains**).

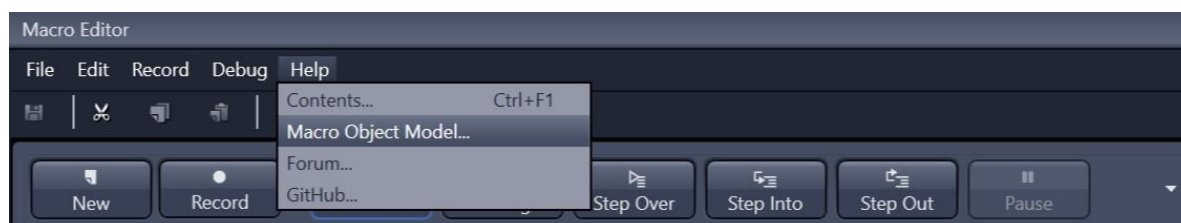
```
# Remove all items from Documents whose name does not contain "lena"
exception = "lena"
docsToRemove = []
for doc in Zen.Application.Documents:
    if (not doc.Name.Contains(exception)):
        docsToRemove.append(doc.Name)

for imgName in docsToRemove:
    img = Zen.Application.Documents.GetByName(imgName)
    # Do not ask to save
    Zen.Application.Documents.Remove(img, False)
    print "removed " + imgName
```

Macros, Part 2 - Developing macros

ZEN Macro Environment

The ZEN macro environment consists of a large variety of components. The complete documentation is available to the user in the submenu **Help > Macro-Object Model...** in the macro editor window:



Here only the most often used elements will be described.

The *ZEN* object is the top node for all objects, available in the ZEN macro environment. It contains the following elements:

Name	Description
Acquisition	Gets the macro details for acquisition handling.
Analyzing	Gets the macro details for Analyzing handling.
Application	Gets the macro details for ZEN application handling.
Devices	Gets the macro details for devices handling.
Measurement	Gets the macro details for measurement handling.
Processing	Gets the macro details for processing handling.
Windows	Gets the macro details for window and dialog handling.

Hint: the names of the Zen class and the subordinate classes in the list do not change.

Acquisition

The **ZenAcquisition** class deals with the image acquisition setup and the acquisition execution.

Acquisition - Methods

Name	Description
AcquireImage	Acquires an image

Macros, Part 2 - Developing macros

AutoExposure	Calculates the exposure time
Execute	Executes the specified experiment
FindAutofocus	Finds the autofocus
StartContinuous	Starts the continuous acquisition
StartLive	Starts the live imaging
StopContinuous	Stops the continuous acquisition
StopLive	Stops the live acquisition
ExecuteMultiBlockImages	Executes a multiBlockImages experiment
FindSurface	Uses the Definite Focus to detect the sample surface and adjust the Z-Position accordingly. The method uses the current Definite Focus settings.
RecallFocus	Tries to recall the previously stored focus (distance to cover glass surface) with the Definite Focus. This is typically the surface with an additional offset.
StoreFocus	Stores the current Z-Position as an offset respective to the sample surface for use by the Definite Focus.
ExecutePanoramaExperiment	Executes a panorama experiment

Note: a number of methods offer optional parameters – in the same manner as the familiar LoadImage. See the Help file for specifics.

The example below shows how to acquire an image:

```
Zen.Application.Documents.RemoveAll()
# Starts the live imaging
Zen.Acquisition.StartLive()
# Acquires an image
image = Zen.Acquisition.AcquireImage()
# Stops the live acquisition
Zen.Acquisition.StopLive()
# Show the acquired image in document area
Zen.Application.Documents.Add(image)
```


Macros, Part 2 - Developing macros

Acquisition - Properties

Name	Description
ActiveCamera	Gets the active camera
CameraSettings	Gets a collection of available camera settings
Experiments	Gets a collection of available experiments
IsContinuousRunning	Returns true if Continues-Mode is running
IsLiveRunning	Returns true if Live-Mode is running

Application

Application - Methods

Name	Description
LoadImage	Loads an image
LoadTable	Loads a table
Save(ZenDocument)	Saves the document
ShowWindow	Shows the ZEN window with the specified window name
ShowWizard	Shows the requested wizard
Wait	Waits for the specified milliseconds
GetOmeTiffFile	Method that asks the user to select a third party microscopy format and converts it to OME-TIFF file with the BioFormats library.
LoadMultiBlockImage	Loads a multi block image document interactively using the file open dialog.
LoadMultiImage	Loads a multiimage document interactively using the file open dialog.

Macros, Part 2 - Developing macros

Pause	Pauses the script and wait for interaction from user. Show a message box with the specified text and a button to continue.
Shutdown	Closes the current ZEN application. Attention! All unsaved changes will be lost. Over TCP you have no possibility to start ZEN again!

Note: the alternatives with optional parameters have been omitted for clarity

Application - Properties

Name	Description
Documents	Collection of documents, currently opened in ZEN
DocumentViews	Collection of views, available for open documents
Environment	ZEN environment, like the names of all relevant folders, etc
ProgressInfo	Gets the controller for a progress info window
Workspaces	Collection of currently available workspaces
ActiveDocument	Gets or sets the active document.
AnalysisSetting	Gets the image analysis setting.
HardwareSettings	Gets a collection with all available hardware settings.
LeftToolArea	Gets the left tool area of the ZEN Blue application.
MacroEditor	Gets script editors details.

The ***GetFolderPath()*** method of the *Environment* object may be of immediate interest. The argument ***ZenSpecialFolder*** can be one of the following special folders (cf. macro help, item ***ZenSpecialFolder*** enumeration):

Member name	Description
Program	Program startup folder, i.e. when ZEN.exe or its variants reside
ProgramDocuments	Root folder for installed (default) documents

Macros, Part 2 - Developing macros

ProgramTemplates	Root folder for installed (default) templates
ProgramResources	Root folder for installed (default) resources
Images	The default storage location for pictures, i.e. the "My Pictures" folder
Documents	The default storage location for documents, i.e. the "My Documents" folder
UserData	The root folder for saving program specific settings and user customizations
UserDocuments	The root folder for saving user specific documents
UserTemplates	The root folder for saving user specific templates
WorkgroupData	The common folder for saving program specific settings and customizations
WorkgroupDocuments	The root folder for workgroup documents
WorkgroupTemplates	The root folder for workgroup templates ¹
ImageAutoSave	The folder for auto-saved images
CameraStreaming	The folder for camera streaming files
WorkflowTemplates	Sandbox folder for workflow templates.
WorkflowResults	Sandbox folder for workflow results - a temporary storage location as defined by the application. Maybe empty if not set.
CommonProgramResources	Root folder for installed (default) resources - common for all programs (EXEs).

The example below lists the subfolders of the *user Documents* folder and then the macros (*.czmac files) in its Macros subfolder:

Macros, Part 2 - Developing macros

```
from System.IO import Directory, File

# Get the user Documents folder
userDocumentsFolder =
Zen.Application.Environment.GetFolderPath(ZenSpecialFolder.UserDocuments)
print userDocumentsFolder

# Lists the subfolders of the user Documents folder
userDocumentsSubFolders = Directory.GetDirectories(userDocumentsFolder)
for userSubFolder in userDocumentsSubFolders:
    print "\t" + userSubFolder.replace(userDocumentsFolder + "\\ ", "")
print

# Get the Macros subfolder
macrosSubFolder = userDocumentsFolder + "\\Macros"
print macrosSubFolder

# Get all macro files, i.e. "*.czmac" files
macros = Directory.GetFiles(macrosSubFolder, "*.czmac")
for macro in macros:
    print "\t" + macro.replace(macrosSubFolder + "\\ ", "")
```

The following windows are known to ZEN and can be opened via the ShowWindow command:

AutofocusWindow	BatchProcessing	CalibrationManager
Diagnostics	DisplaySettingManager	Dosimeter
GeneralLoginScreen	KitchenTimer	LocalizeCheck
ModuleManager	SampleHolderTemplates	Scaling
SystemTestWindow	VivaTomeQuickCalibration	WorkflowDesigner

The following windows are modal, i.e. they force you to deal with them first:

NewDocument	Options
UserAndGroupManagement	ZenPrintPreviewWindow

Macros, Part 2 - Developing macros

Devices

The simplest way to master devices available is via *Settings*. Using the Settings editor one can retrieve the status of the instrument hardware and save it under an appropriate name, or one can load the setting stored under the given name and apply it to the system. A subset of commands available is presented here (see more in the macro help file[1])

Devices - Methods

Name	Description
ApplyHardwareSetting	Applies the specified hardware setting to the underlying hardware.
ReadHardwareSetting	Gets the active hardwareSetting
GetLampByName	Gets the lamp device of the microscope by name

Devices - Properties

Name	Description
HardwareSettings	Gets a collection of all available hardware settings
Lamp	Gets the lamp device
ObjectiveChanger	Gets the objective changer device
Optovar	Gets the optovar changer device
Reflector	Gets the reflector changer device
Stage	Gets the stage device
IsMicroscopeInitialized	Gets a value indicating whether the microscope is completely initialized.

The following example scans through all objective revolver positions, retrieving the name of objectives and their magnifications.

Macros, Part 2 - Developing macros

```
# Get all objective revolver items
objRevolver = Zen.Devices.ObjectiveChanger
print("The number of objective revolver items: " +
str(objRevolver.ItemsCount))
print ' Position mag\tobjective '
print
# Loop over the objective revolver positions
for objRevolver.TargetPosition in range(1, objRevolver.ItemsCount + 1):
    objRevolver.Apply()
    # Get the Position, Magnification und Name of current objective revolver item
    objinfo = "
    objinfo = objinfo + ' ' + format(objRevolver.ActualPosition)
    objinfo = objinfo + '\t' + format(objRevolver.Magnification)
    objinfo = objinfo + '\t' + objRevolver.ActualPositionName
    print objinfo
```

Windows

The user can create and open windows from within the macro using the following methods:

Name	Description
CreateWindow	Creates a window
Show	Creates a window with the specified contents and an OK button
ShowDropDown	Shows a dropdown list box, asking for user input
ShowImage2d	Shows a window with a 2D projection of the image
ShowTextbox	Shows a textbox, asking for user input

A code snippet with an example for the **ShowDropDown** method

Macros, Part 2 - Developing macros

```
# Get all opened files in document area
zapDocs = Zen.Application.Documents
# Create a list to store the filenames without path
zapNames = []
for doc in zapDocs:
    zapNames.append(doc.Name)
# Create a window with a drop down list to show all files
zappedName = Zen.Windows.ShowDropDown("activate document", zapNames, 0)
# Make the selected file as the active file
zapDocs.ActiveDocument = zapDocs.GetByName(zappedName)
# Get the name of active file
fileName = zapDocs.ActiveDocument.FileName
print("Active Document: " + fileName)
```

Processing

The **Processing** object provides the image processing functions and is, due to its size, divided into several subgroups.

Processing - Methods

Often the function parameters are not known in advance, as they depend on the image being processed, so they need to be determined interactively during execution. The parameters once set interactively can be reused later during execution as follows:

```
image2 = Zen.Processing.Transformation.Geometric.Rotate(image1, True)
image4 = Zen.Processing.Transformation.Geometric.Rotate(image3)
```

The command in the second line above will rotate image3 using the set of parameters, defined and used in the first line.

The methods, listed below, will force the parameters remembered to be forgotten.

Processing - Properties

Name	Description
Adjust	Gets a function from the group adjust
Arithmetics	Gets a function from the group arithmetics

Macros, Part 2 - Developing macros

Binary	Gets a function from the group binary
Filter	Gets a function from the group filtering
Morphology	Gets a function from the group morphology
Options	Gets the options for ip processing
Segmentation	Gets a function from the group segmentation
Transformation	Gets a function from the group transformation
Utilities	Gets a function from the group utilities
ObjectClassification	Gets the function for group object classification.
TimeSeries	Gets the function for group for time series.

Here is the list of individual functions³ for each of the groups in the above table.

Processing.Adjust group

BrightnessContrastGamma	ColorBalance	ColorTemperature
HueSaturationLightness	ShadingCorrection WhiteBalance	StackCorrection

Processing.Arithmetics group

AddConstant	Addition	Average
Combine	Division	Exponential
Inversion	Logarithm	Maximum
Minimum	Multiplication	MultiplyConstant
Negation Square	SquareRoot	Subtraction

Processing.Binary group

And	Distance	Exoskeleton
FillHoles	Invert	LabellImage
MarkRegions	Or	Scrap

³ Functions are among properties of a given class and not, as one would expect, among its methods. Think of these entries as *factories* producing the selected function, e. g. *Processing.Adjust* factory creating an instance of the *WhiteBalance* function.

Macros, Part 2 - Developing macros

Thinning

UltimateErode

Xor

Processing.Filter group

The Filter group consists of three subgroups, to filter edges, to sharpen and to smooth the image.

Processing.Filter.Edges subgroup

GradientMax

GradientSum

Highpass

Laplace

LocalVariance

Roberts

Sobel

Processing.Filter.Sharpen subgroup

Delineate

EnhanceContour

Processing.Filter.Smooth subgroup

Binomial

Denoise

Gauss

Lowpass

Median

Rank

Sigma

Processing.Morphology group

Close

Dilate

Erode

Gradient

GrayReconstruction

Open

TopHatBlack

TopHatWhite

Watersheds

Processing.Segmentation group

Canny

Marr

RegionGrowing

Threshold

ThresholdAutomatic

ThresholdDynamic

Valleys

Processing.Transformation.Geometric group

ChannelAlignment

FlipHorizontal

FlipVertical

Mirror

OrthoView

Resample

Rotate

Shift

ZstackAlignment

Macros, Part 2 - Developing macros

Processing.Utilities group

AddChannels	ChangePixelType	ColorConversion
CombineHls	CombineRgb	CopyImage
CreateGrayScaleImage	CreateSubset	ExportMovie
ExportOmeTiff	ExportSingleFile	

Measurement

The **Measurement** object can be used for the interactive measurement in images.

Measurement - Methods

Name	Description
MeasureToTable	Creates a table of measurements for the graphic elements, drawn into the graphic layers of the image
MeasureSequenceInteractive	Measures the given sequence settings in the specified input image.
UpdateMeasurementText	Updates the measurement text of all graphic elements on image.

Measurement - Properties

Name	Description
FeatureSets	Gets a collection with all available feature sets.
FeatureSubsets	Gets a collection with all available feature subsets.
MeasurementSequenceSettings	Gets a collection with all available measurement sequence settings.

The following example should demonstrate its function:

Macros, Part 2 - Developing macros

```
# Clear all files in the document area
Zen.Application.Documents.RemoveAll()
# Load an image
image = Zen.Application.LoadImage()
# Show the image in the document area
Zen.Application.Documents.Add(image)

msg = ""Switch to Graphics view or Graphics menu,
and select a tool, then draw objects.
When done, press continue""

# Create a message box and show the text above
Zen.Application.Pause(msg)

# Create a table of measurements for the graphic elements
# Drawn into the graphic layers of the image
resultsTable = Zen.Measurement.MeasureToTable(image)
# Show the table in the document area
Zen.Application.Documents.Add(resultsTable)
```

Analysis

Analysis - Methods

This chapter gives you a first impression of Analysis features. Please keep in mind that you need the Analysis Model to use these functions.

Name	Description
Analyze	Segments and measures the specified input image.
AnalyzeBatch	Segments and measures the specified input images (.czi only).
AnalyzeInteractive	Starts the wizard for analyzing the image.
AnalyzeToFile	Segments and measures the specified input image and output in a file.
AnalyzeToImage	Labels the image.

Macros, Part 2 - Developing macros

AnalyzeToTable	Segments the input image, measures it and stores data in a data table.
CreateRegionsTable	Creates the regions data table from the image attachment.
CreateRegionTable	Creates the region data table from the image attachment.
CutOutRegions	Masks the image slices according the specified regions.
DefineRegionsMask	Starts the wizard for analyzing the image with creating the interpolate mask setting.
FillRegionsTable	Creates the regions data table from the image attachment.
FillRegionTable	Creates the region data table from the image attachment.
FillTables	Fills the region and regions data tables from the image attachment.
GetImageAnalysisSetting	Gets the image analysis setting.
GetRegions	Gets the regions from the image attachment.
GetTable	Gets the table.
LabelImage	Labels the image.
Measure	Measures the specified input image given the segmented regions in the image attachment.
ResetAnalysisResult	Resets the analyzer result in the images.
Segment	Segments the image and stores the resulting regions in the image attachment.
StartAnalysisSettingWizard	Starts the wizard for editing the image analysis setting.

Macros, Part 2 - Developing macros

ZEN Macro Classes

The classes, described in the previous chapter, all stem from the same generating **ZEN** object, made available by the ZEN application, when the scripting environment gets active. It then allows creating new objects, like new images, using **LoadImage**.

For the classes, listed below, new objects can be created directly inside the macro environment. For instance:

```
# Remove all opened files in document area
Zen.Application.Documents.RemoveAll(False)

# Define parameters of new image
min_greyValue = 20
max_greyValue = 100
generator_pattern = 4
image_width = 1024
image_height = 1024
# Create new image
myImage = ZenImage(image_width, image_height, ZenPixelType.Gray16,
min_greyValue, max_greyValue, generator_pattern)

# Define channels of image
channels = ("DAPI", "GFP", "YFP", "FaKo")
# Define colors of image
colors = (ZenColors.Blue, ZenColors.Red, ZenColors.Green, ZenColors.Yellow)
for iCol in range(0, channels.Count):
    # Set channel of image
    myImage.SetChannelName(iCol, channels[iCol])
    # Set color of image
    myImage.SetChannelColor(iCol, colors[iCol])
# Show the image in document area
Zen.Application.Documents.Add(myImage)
```

Class	Description
ZenBounds	Class to hold bounds information of the image.
ZenCameraSetting	The CameraSetting contains up to 2 camera configurations.

Macros, Part 2 - Developing macros

ZenDisplaySetting	Handles display setting for ZenImages like channel color and histogram values. The Display setting can contain display settings for multiple channels.
ZenDocument	Base class for the Zen documents - ZenImage and ZenTable
ZenExperiment	Experiment for Acquisition
ZenFrequentAnnotationGraphic	Frequent annotation
ZenGraphic	The base object for graphic elements. An element can be added to <i>ZenImage</i> objects.
ZenGraphics	ZenGraphics is a mask with a collection of graphic elements
ZenHardwareSetting	Hardware setting
ZenImage	The Image document for handling images. Creates a new <i>ZenImage</i> object.
ZenImageAnalysisSetting	Analysis setting
ZenImageMetadata	Holds additional information of the image
ZenRegionClass	Class representing the region class
ZenRegionsClass	Class representing the region class
ZenScaleBarGraphic	A scale bar
ZenTable	The Table document for handling tables
ZenTextBoxGraphic	A text box
ZenWindow	A window for custom input

As the **ZenImage** class is central to ZEN, its methods and properties are given below.

ZenImage - Methods

Macros, Part 2 - Developing macros

Name	Description
AddChannel	Adds the specified reference image to a new channel.
AddScene	Adds the image to a new multi-scene image.
AddSubImage	Adds the reference image to specified dimensions. If the reference image has multi-dimension, only the first sub-block will be added.
AddSubset	Adds the specified subset image to an existing Image
AttachImageLabel	Attaches an image as label
BurnInGraphics	Burns all graphics in the image.
ClearChannels	Clears the channels.
Clone	Clones the <i>ZenImage</i> instance.
Close	Closes the document. Don't close an unsaved document. Removes and disposes the document.
CopyPixelsFromArray	Copies the pixels from array.
CopyPixelsToArray	Copies the pixels to array.
CopyRois	Copies the regions of interest (ROI).
CreateSubImage	Creates a subset image.
InsertDefaultScaleBarGraphic	Inserts the default scale bar graphic.
Load	Load document to an object. The document is not opened in GUI.
Save	Save the document.
SetChannelColor	Sets the color of the channel.
SetChannelName	Sets the name of the channel.
SetDisplaySetting	Sets the display setting.
GetPositionLeftTop	Stores necessary information (based on given image) for subsequent coordinate transformation during creation of experiment regions.

ZenImage - Properties

Name	Description
------	-------------

Macros, Part 2 - Developing macros

Bounds	Gets the bounds.
DisplaySetting	Gets the display setting from the image.
Comment	Gets or sets the comment of this image.
Description	Gets or sets the description of this image.
GraphicLayers	Gets the graphic layers of the image document.
HasPyramid	Gets a value indicating whether the image has a pyramid.
IsDisposed	Gets a value indicating whether this instance is disposed.
IsModified	Gets a value indicating whether this document is modified.
IsZenImage	Gets a value indicating whether the document is an image or not.
IsZenMultiBlockImage	Gets a value indicating whether the document is an multiblock image or not.
IsZenTable	Gets a value indicating whether the document is a table or not.
Keywords	Gets or sets the keywords of this image.
NameWithoutExtension	Gets the filename of the document (name without Extension).
Rating	Gets or sets the rating of this image.
Scaling	Gets the scaling of the image document.
Title	Gets or sets the title of this image.
FileName	Gets the filename of the document (name including the path).
Graphics	Gets or sets all graphics for an image. The graphic elements are collected in a mask.
Metadata	Gets the metadata.
ImageAttachments	Allows access to image attachments.

Macros, Part 2 - Developing macros

Multidimensional Images

A powerful case for the use of arrays and lists is multidimensional images. A volume image, imaged as a Z-stack, can be addressed as an array of consecutive Z sections. A movie, obtained from a time series, is an array (or ordered list) of single time frames. Multi-channel images can be viewed as an example of unsorted lists of basic, two-dimensional images.

Note that image processing and analysis methods are homogenous, as regards the multidimensionality of images: median filter will run then same way whatever the dimensionality of the input and will create an output image of the same dimensionality. However, there are cases, where one needs an output of a different dimensionality as the input – a 2D copy of one of the elements in the multidimensional image being a typical case. One can thus expect that some of the methods and properties, described above, will turn up here as well.

The following example creates a multidimensional image (MultiChannel x Z-Stack):

```
# Remove all open files in document area
Zen.Application.Documents.RemoveAll(False)

# List for channels
channels = ["♠", "♥", "♦", "♣"]
# List for colors
colors = [ZenColors.Blue, ZenColors.Red, ZenColors.Green, ZenColors.Yellow]
# List for Z index
z_index = ["1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12", "13"]
# Set width of the new ZenImage
width = 75
# Set height of the new ZenImage
height = 92
# Create new ZenImage
myImage = ZenImage(width, height, ZenPixelType.Gray8, 13, 1, 4)
# Set the name of the new ZenImage
myImage.Name = "fullset"

for i in range(colors.Count):
    for j in range(z_index.Count):
        # Link of subimage:
        # https://www.soolide.com/de/357579/dog-puppy-on-garden-royalty-free-image-1586966191-2
        # Or you can choose any subimage that you like
        filename = "C:\\OAD\\Input\\dog.jpg"
        # Load the subimage
```

Macros, Part 2 - Developing macros

```
subimg = Zen.Application.LoadImage(filename)
# Add the subimage to the new created ZenImage,
myImage.AddSubImage(subimg, 0, i, j)
# Set the channel of the image
myImage.SetChannelName(i,channels[i])
# Set the color of the image
myImage.SetChannelColor(i, colors[i])
# Show the image in document area
Zen.Application.Documents.Add(myImage)

# Get the number of new created images
opendocs = Zen.Application.Documents
print opendocs.Count
```

The following two commands are relevant for the example:

1. ***ZenImage(width, height, pixeltype, zSlices, tSlices, channels)***: the command creates an image document of the given width, height and pixel type, and of the specified dimensionality. In the example, an 8-bit image with 13 Z-positions and 4 channels is created.
2. ***<ZenImage>.AddSubImage(subimg, timeIndex, ChannelIndex, zIndex)***: The command adds the image in the first parameter as a subimage to the target at the specified position, at the provided time, channel and Z index.

An antipod to ***AddSubImage*** is the ***CreateSubImage (string subsetString)*** method. The subset string has the following format: D (values) | D (values) | ... where D is one of the X, Y, C, Z, T, A, S, I... axes. The parameter values are a comma separated list of either individual indices, e. g. Z (1,4,5,6,9), or range of indices, e. g. Z (3-5), or any combination thereof, e. g. Z (1,3,10-20,3,5).

For convenience, all indices in the Subset strings start with 1. Nth element thus has the index N. For instance, to extract the first 4 time points from the image, one can use T (1-4). To extract first 200 time points for Z= 10, the string will look like T (1-200) | Z (10). Dimensions not included in the subset string are not filtered: for instance, if there are channels in the example image as well you do not specify which channel or channels have to be extracted, then all channels will be included.

Here first the list of available dimensions and their meaning:

ID	Info
X	Pixel index / offset in the X direction – available also for tiled images.

Macros, Part 2 - Developing macros

Y	Pixel index / offset in the Y direction– available also for tiled images.
C	Channel in a Multi-Channel data set
Z	Slice index (Z – direction).
T	Time point in a sequentially acquired series of data.
R	Rotation – used in acquisition modes where the data is recorded from various angles.
S	Scene – for clustering items in X/Y direction (data belonging to contiguous regions of interests in a mosaic image).
I	Illumination - illumination direction index (e.g. from left=0, from right=1).
B	(Acquisition) Block index in segmented experiments.
M	Mosaic tile index – to reconstruct the image acquisition order and to be used in conjunction with a global position list and the scaling information to define the pixel offset
H	Phase index – for specific acquisition methods.
V	View index (for multi – view images, e.g. SPIM)

The following code snippet lists the dimensions entries of the images in the gallery

```
for doc in Zen.Application.Documents:
    if (doc.IsZenImage):
        print doc.Metadata.Dimensions + "\t" + doc.Name
```

The property **Dimensions** is a string, for instance “X (0,1200) Y (0,602) C (0,4) Z (0,13) T (0,1)” in case of the image **dog.jpg**. Instead of parsing this string to get at single dimensions and their sizes, you can use the **Bounds** object and its off-springs instead:

```
for d in Zen.Application.Documents:
    if (d.IsZenImage and d.Bounds.IsMultiChannel):
        print "{0}\t{1}\t{2}".format(d.Bounds.StartC, d.Bounds.SizeC, d.Name)
```

Macros, Part 2 - Developing macros

Including Code and Importing Functionality

If you have a library of Python functions, Python allows you to “store it once, use many times” by using the import command. Import has been mentioned and used already, so the information here can be seen as an extended repetition.

Python looks in several places when you try to import a module. Specifically, it looks in all the directories specified in **sys.path**. Python will look through these directories (in this order) for a **.py**⁴ file matching the module name you're trying to import.

```
import sys
for path in sys.path:
    print path
```

You can add a new directory to Python's search path at runtime by appending the directory name to **sys.path**, and then Python will look in that directory as well, whenever you try to import a module. The effect lasts as long as Python is running.

Python has two ways of importing modules. Both are useful, and you should know when to use each. Create two short Python files as follows:

HelloModule.py:

```
def Hello(who):
    print "Hello, " + who + "!"

Hello("world")
```

Save it in one of folders, mentioned in the sys path, for instance in the folder, where ZEN.exe resides. Have a look then at the following example, that uses the module two different ways: first the example using from construct:

⁴ Because not all modules are stored as .py files, the truth is a little more complicated. For instance, modules may stem from the .NET environment (like System.IO), or, like the sys module, have baked right into Python itself.

Macros, Part 2 - Developing macros

```
from HelloModule import Hello
Hello("James")

import HelloModule
HelloModule.Hello("Jane")
```

The first form, using `from`, is already familiar from examples involving classes **Directory**, **FileInfo** in `System.IO`.

Now you tell me...

The statement **`import sys`** looks so simple and elegant, self-explaining – once it has been written. How to proceed when you need some functionality outside ZEN, that you feel must definitely be on hand, somehow, somewhere? Here's some typical cases:

- File and device handling
- Accessing System and Windows properties
- Interfacing to social networks
- Using cloud services

You are welcome to extend the above short list.

The first step, before searching for existing solutions, is to define in few words what you need to have your problem solved. We will take an example: we would like to time the execution of the macro being developed, so, next to *Time* of course, the possible keywords would be

- **System** – to avoid hitting on the *Time* magazine, time measurement and zones in Wikipedia etc.
- **.NET** – to ignore Time/System planners and other calendar stuff, Linux and Windows XP entries etc.

With the keywords “time system .NET” we do not need to go any further than the top hit or two on any search machine in the internet: they point right away into the MSDN (Microsoft Developer Network) pages, documenting the `DateTime` System property.

Taking a gamble that pays, we first import said property and take a peek at it...

```
import System.DateTime
print System.DateTime.Now
```

Macros, Part 2 - Developing macros

... to get the current date and time in the Message area.

From here on it is all peaceful sailing to the following macro, involving some of the functions available via DateTime - note that from the moment we type System., code completion is coming to help:

```
import System.DateTime

# Get the current date and time
timeNow = System.DateTime.Now
print timeNow
print "Current day\t", timeNow.Day, timeNow.DayOfWeek
print "current month\t", timeNow.Month
print "current month\t", timeNow.Year
print "Date in long form\t", timeNow.ToLongDateString()
print

timeBefore = System.DateTime.Now
# Adds 3000 milliseconds to the value of timeBefore
timeAfter = timeBefore.AddMilliseconds(3000)
while (timeAfter > System.DateTime.Now):
    pass
print "starting time\t", timeBefore
print "ending time\t", timeAfter
```

It is worth mentioning that identical data can be obtained via intrinsic Python functionality (look for it by searching for “iron python date time”):

```
import time
print time.gmtime()
print time.time()
```

The reader may wonder “Why two ways of achieving the same? And are why different?”. Note that ZEN is a .NET-based application and looking there first is probably the shortest way to the answer. And, as in the case of DateTime, you will often get the code completion for free with it.

Macros, Part 2 - Developing macros

Inter-process communication

In computing, inter-process communication (IPC) is a set of methods for the exchange of data among multiple threads in one or more processes. There are several reasons for providing an environment that allows process cooperation, such as information sharing, computational speedup, modularity and convenience.

To enable this type of communication, data must be moved between different parts of a computer program or from one program to another using the so-called marshaling. The first line in the example below imports the necessary marshaling module. Once the Marshal object is available, it can be used to connect ZEN with some other independent process, running on the computer, in the example below the Office Excel.

```
from System.Runtime.InteropServices import Marshal
# Make sure that excel has been activated
excel = Marshal.GetObject('Excel.Application')

if excel == None:
    Zen.Windows.Show('Excel has not been started')

else:
    # Set the value of cell(1,1)
    excel.Cells(1,1).Value = "OK"
    for row in range(1, 20):
        # Set the values of the second column
        excel.Cells(row,2).Value = row
```

You will notice there's no code completion – i. e. Intellisense - available for Excel. On the other hand, Intellisense and some more is available in the Excel's Macros environment. You can turn on the recording in Excel and then use the created VBA code as a template for macro code.

Here's one more example to show how data can be transferred from ZEN to Excel:

Macros, Part 2 - Developing macros

```
from System.Runtime.InteropServices import Marshal
from System import *

# Make sure that excel has been activated
excel = Marshal.GetObject('Excel.Application')

Zen.Application.Documents.RemoveAll()

# Load a ZenTable
table = Zen.Application.LoadTable()
print table.Name
Zen.Application.Documents.Add(table)

# Make sure you have a fresh, empty workbook on hand
if (excel.Workbooks.Count == 0):
    excel.Workbooks.add()

for Col in range(0, table.ColumnCount):
    # Get the Caption of ZenTable and store them in excel
    excel.Cells(1, Col+1).Value = table.Columns[Col].Caption
    for Row in range(0, table.RowCount):
        # If the values in ZenTable are type "int" oder "float", get the value directly
        # store them in excel
        if isinstance(table.GetValue(Row, Col), float) or
isinstance(table.GetValue(Row, Col), int):
            excel.Cells(Row+2, Col+1).Value = table.GetValue(Row, Col)
        else:
            # If the values are type "DBNull", let the cells be empty
            if table.GetValue(Row, Col) == DBNull.Value:
                excel.Cells(Row+2, Col+1).Value = ""
            else:
                # For other types, such as datetime, show the values as string
                excel.Cells(Row+2, Col+1).Value = table.GetValue(Row, Col).ToString()
```

Hint: If you just want to have the data table in Excel, then the following single line (additionally to the top two lines that connect ZEN and Excel) would suffice:

```
# alternative: load csv in Excel!
excel.Workbooks.Open(csvFileName)
```


Macros, Part 2 - Developing macros

In the above case, ZEN is the client for the services, provided by the Excel server. The analogous code in Excel VBA could achieve the same communication with the roles reversed, i. e. Excel as client for ZEN services.

Macros, Part 2 - Developing macros

Appendix A – Configuring MTB For the Simulation Mode

The MTB2011 Configuration software is used to generate and manage microscope and stereomicroscope configurations.

Information about microscope components (nosepieces, reflector turrets, shutters etc.) and, if necessary, additional external units (motorized x and y stages, external light sources etc.) is stored in these configurations. Furthermore, the MTB2011 Configuration program can also be used to enter information about microscope components, such as objectives, fluorescence filter cubes etc., in a simple way and to save this information in the microscope (depending on the type of microscope in question). In this case, the information is saved directly in the microscope, allowing it to be displayed on the microscope's **TFT** screen, for example.

Various configurations can be created, of which only one is activated at any time. The active configuration is used by control and imaging software such as ZEN from Carl Zeiss to provide graphic control dialogs for the configured microscope units.

The standard ZEN installation also includes the installation of the MTB2011 Configuration software. If MTB2011 Configuration is installed, you will see the following icon on the Microsoft Windows desktop:



In case MTB2011 Configuration is missing, install it from the ZEN installation DVD. Open the folder **Other** on the DVD.

The MTB2011 Configuration documentation (MTB2011Configuration_Installation + Reference.PDF) can be found in the Manuals/MTB2011 on the DVD.

MTB2011 Configuration program allows you to specify the hardware you have at your disposal. **For** learning and testing purposes, you can request that the configuration is to run in an emulation mode. If you already have hardware configured, proceed as follows:

- Export the configuration and Import it under a different name (Menu File)
- Activate the copied configuration and check the "Simulate Hardware" checkbox
- Press Apply to activate it and OK to exit the MTB2011 Configuration program.

Activate any camera you may have available. Activate the created configuration and, as above, check the "Simulate Hardware" checkbox. Press Apply to activate it and OK to exit the MTB2011 Configuration program.

Macros, Part 2 - Developing macros

