



itom Documentation

Release 69868c9

Institut for Technical Optics (ITO), University Stuttgart

September 02, 2013

CONTENTS

1	Introduction	1
1.1	About us	2
1.2	Licensing	2
2	Installation	5
2.1	Minimum system requirements	5
2.2	Installation from setup	5
2.3	Build from Sources	16
2.4	Plugins, Designer-Plugins	25
2.5	Get this help	25
3	Getting Started	29
3.1	Quick Start	29
3.2	Further Information	38
3.3	Tutorials	38
4	The itom User Interface	39
4.1	Main Window	39
4.2	Editor	48
5	Plots and Figures	51
5.1	Quick tutorial to plots and figures	51
5.2	Figure Management	54
5.3	line plots (1D)	56
5.4	2D image plots	56
5.5	isometric Plot	56
6	Extending the user interface of itom	57
6.1	Customize the menu and toolbars of itom	57
6.2	Show messages, input boxes and default dialogs	60
6.3	Creating advanced dialogs and windows	63
6.4	Implement a more complex GUI in a plugin (C/C++)	75
7	Plugins	77
7.1	Basic concept	77
7.2	How to start a plugin	77
7.3	Development under C++	77
8	Python scripting language	179
8.1	Introduction	179
8.2	Python module itom	213
8.3	Further python packages	217
9	itom Script Reference	221

9.1	itom methods	221
9.2	dataIO	228
9.3	actuator	232
9.4	Algorithms, Widgets and Filters	236
9.5	figure	237
9.6	plotItem	238
9.7	ui-elements (GUI)	238
9.8	dataObject	247
9.9	npDataObject	257
9.10	point	281
9.11	pointCloud	283
9.12	polygonMesh	286
9.13	region	287
10	Miscellaneous	289
10.1	How to use the help	289
10.2	Units and Conventions	289
11	Demo scripts	291
11.1	itom Basics	291
11.2	Plugins	291
11.3	Algorithm / Filter	292
11.4	ui	292
12	Python tutorials	293
12.1	determine lateral image shift and show images by using itom figure plots	293
13	Indices and tables	297
	Python Module Index	299

INTRODUCTION

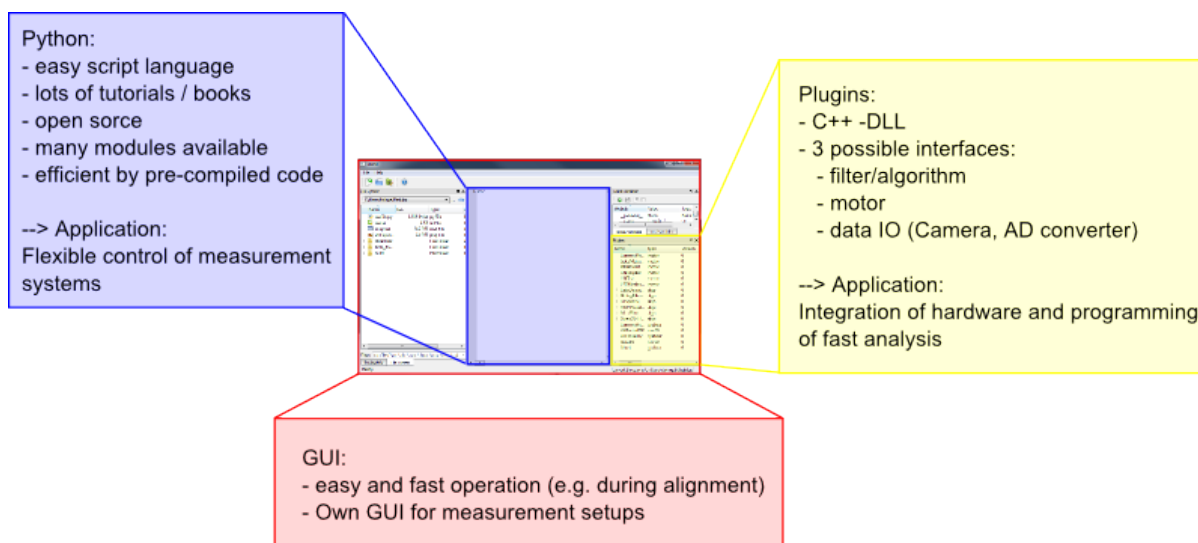
itom is a lab automation and measurement software developed, maintained and provided by the ITO (Institut for Technical Optics, University Stuttgart, Germany). While this software has been designed with an emphasis on developing and running optical systems, it is not limited to this field of application. **itom** represents a versatile tool for building the control software for any measurement setup aimed at a high degree of adaptability to different parameters and variable hardware. To address these requirements it was decided to design a powerful and fast core-program, integrating a scripting language in order to easily modify and transfer code blocks from one part of the program to another and to provide a simple and unitary interface for controlling external hardware components. Beyond the control of the physical setup, **itom** provides a wide variety of features for the analysis and processing of the acquired data. **itom** is best considered as an open source lab automation software whose functionalities lie somewhere between commercial software packages like Matlab or Labview.

The following table shows in which way base requirements to such a software system have influenced the choice of components during the designing process:

What we wanted	How we did it
fast software, development of fast algorithms	C++
modern multi-platform GUI	Qt-Framework
easy integration of different hardware (camera, actuator,...)	Plugin system
fast, robust and easy-to-learn scriptinglanguage	Python

In the figure below you can see the three columns on which **itom** is based:

1. Python
2. Plugins
3. GUI



Based on these three columns you can control measurement applications, basic setups or scripted image processing.

To learn more about how to control **itom** via script language or the GUI proceed to *Getting Started*.

TODO: Add the basic idea and structure of ITOM and explain it in more detail than this above

1.1 About us

Institut für Technische Optik
Universität Stuttgart
Pfaffenwaldring 9
70569 Stuttgart

Report issues at:

<https://bitbucket.org/itom/itom/issues>

This help has been built for itom-version 1.0.14 (SVN/Git-revision 69868c9).

1.2 Licensing

1.2.1 itom Licensing

The core components and the main application of **itom** are covered by the **GNU Library General Public Licence** (GNU LGPL). All components belonging to the SDK of **itom** (e.g. *dataObject*, *pointCloud*, *addInInterface*,...) are additionally covered by an **itom exception**. The main idea of this exception is to allow other libraries (e.g. plugins) to include and link against components of **itom** SDK independent on the specific license model of the respective “other” library. All files belonging to the **itom** SDK are included in the folder **SDK** that is shipped with any setup or included in the build directory (when build from sources).

The full text license of **LGPL** and **itom exception** is also included as file *COPYING* in the source distributions and setups.

All plugins and designer-plugins that can be integrated into **itom** can have their own licensing. Therefore the user is referred to the specific licensing documents or statements of each external library (plugin).

1.2.2 itom Exception

All components (source files) which are part of **itom** SDK are not only covered by the LGPL license but also by the *itom Exception*, that provides further grants and rights for using these components:

ITO LGPL Exception version 1.0

Additional rights granted beyond the LGPL (the "Exception").

As a special exception to the terms and conditions of version 2.0 of the LGPL, ITO hereby grants you the rights described below, provided you agree to the terms and conditions in this Exception, including its obligations and restrictions on use.

Nothing in this Exception gives you or anyone else the right to change the licensing terms of the itom Software.

Below, "Licensed Software" shall refer to the software licensed under the LGPL

and this exception.

- 1) The right to use Open Source Licenses not compatible with the GNU Library General Public License: Your software (hereafter referred to as "Your Software") may import the Licensed Software and/or distribute binaries of Your Software that imports the Licensed Software.
- 2) The right to link non-Open Source applications with pre-installed versions of the Licensed Software: You may link applications with binary pre-installed versions of the Licensed Software.
- 3) The right to subclass from classes of the licensed software: Classes that are subclassed from classes of the licensed software are not considered to be constituting derivative work and therefore the author of such classes does not need to provide source code for this classes.

1.2.3 Package Licences

A standard distribution of **itom** links to the following third-party library (shared linkage):

- [OpenCV](#) by Willow Garage under BSD-license
- [Point Cloud Library](#) by Willow Garage under BSD-license
- [QT-Framework](#) by Nokia under LGPL.
- [QScintilla](#) in by Riverbank Computed Limited under GPL with additional exceptions.
- QScintilla is a port to Qt of Neil Hodgson's [Scintilla](#) C++ editor control.
- [Python](#) by Python Software Foundation under Python-License (similar to BSD license).
- Python-package [NumPy](#) under BSD compatible license.
- Some classes concerning the interaction between **Qt** and **Python** have been inspired by [PythonQt](#) under LGPL license.

Additionally, the following third-party libraries are also used by common plugins or common configurations of **itom**:

- [SciPy](#) under BSD compatible license.
- [Matplotlib](#) under BSD compatible license.
- [QwtPlot](#) by Uwe Rathmann and Josef Wilgen under LGPL with additional [exceptions](#).
- [QPropertyEditor](#) under LGPL
- Google test framework by Google under New BSD-license

Other plugins may also contain further third party packages, e.g.:

- NVidia CUDA SDK by NVidia under NVidia CUDA SDK License
- [OpenMP](#)
- Hardware dependent third party drivers and SDKs with different license models or terms of conditions than the main itom-programm. So please check the specific plugin-license.

INSTALLATION

In this section we want to show you how to get and install **itom** on your computer.

2.1 Minimum system requirements

Before installation, please review the minimum system requirements.

- Operating System
 - Windows XP SP3 *or*
 - Windows Vista SP1 (SP2 and platform update recommended) *or*
 - Windows 7 *or*
 - Windows 8 (not tested yet) *or*
 - Linux based OS (tested with Debian)
- 32 or 64bit processor architecture
- 800MHz processor (dual-core recommended)
- 512MB of RAM
- 1024 x 768 screen resolution
- 200+ MB hard drive space

2.2 Installation from setup

Currently, there are both a *32bit* and a *64bit* setup version of **itom** available (Windows only). For linux users there is no pre-compiled package available. The use of the setup version is recommended for users that mainly want to work with **itom** without developing new algorithms or plugins. The setup can be downloaded from <https://bitbucket.org/itom/itom/downloads>. The setup installs the core application, a number of useful hardware plugins and the designer plugins which provide plotting functionalities to **itom**.

2.2.1 Windows Setup Installation

For your convenience you can download an installer package for Microsoft Windows from [<https://bitbucket.org/itom/itom/downloads>](https://bitbucket.org/itom/itom/downloads). There are several different types of setups available:

- Windows, 32bit, **itom** only
- Windows, 64bit, **itom** only
- Windows, 32bit, **itom**, Python + packages: numpy, scipy, matplotlib, PIL (optional)

- Windows, 64bit, **itom**, Python + packages: numpy, scipy, matplotlib, PIL (optional)

These setups do not contain any plugins or designer-plugins.

In the course of the installation, the following third-party components will be installed along with **itom**:

1. Microsoft Visual C++ 2010 Runtime Libraries (x86 or x64)
2. Python 3.2.3 (optional)
3. Python package *numpy* 1.6.2(optional)
4. Python package *scipy* 0.10.1 (optional)
5. Python package *matplotlib* 1.2.x (optional)
6. Python package *PIL* 1.1.7 (optional)

In the following we will guide you through the installation setup with a couple of screenshots:

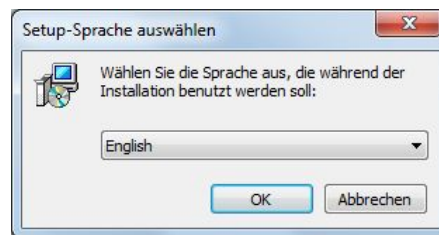


Figure 2.1: Please select your desired language for the setup.

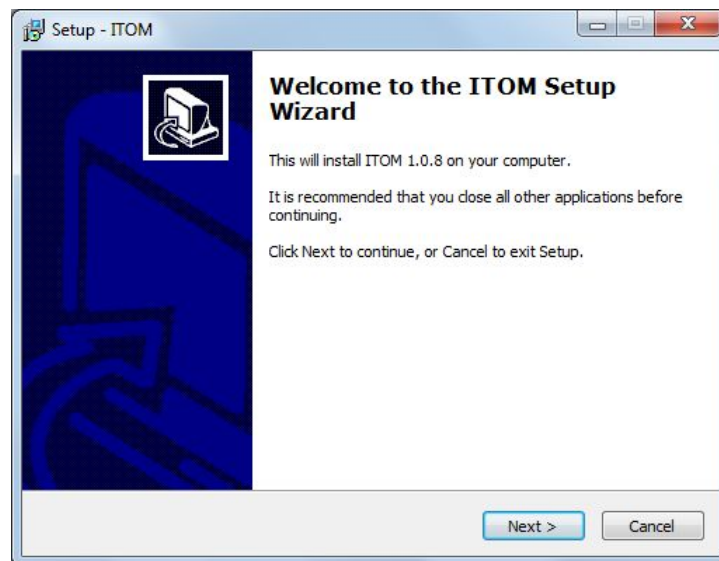


Figure 2.2: The start screen of the setup will appear.

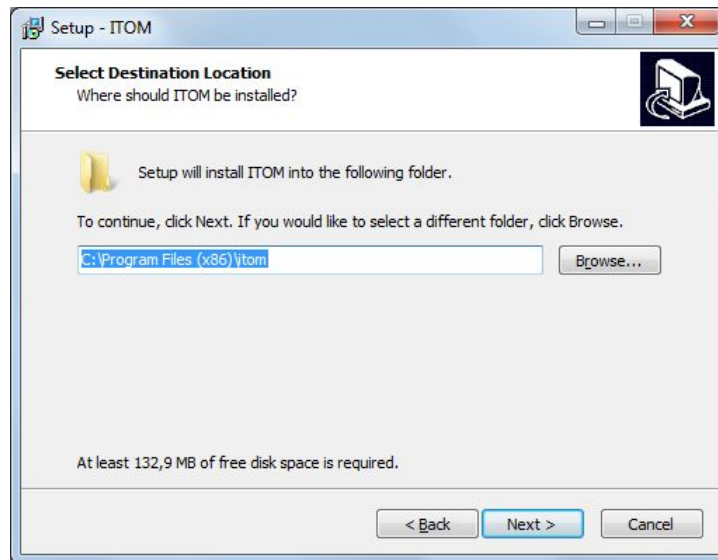


Figure 2.3: Choose where to install **itom** on your file system.

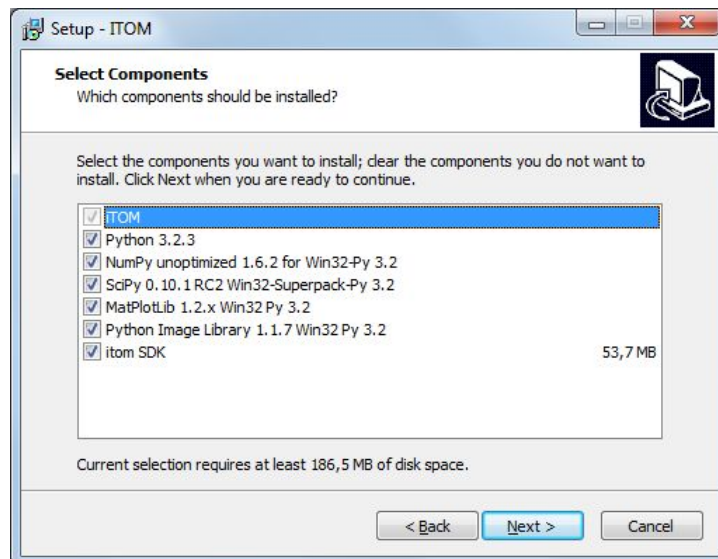


Figure 2.4: Depending on your version of the setup, you now need to (de)select some optional components. The SDK is important if you want to develop your own plugins for **itom**. If you have the extended setup version, you can also select that python including some important packages is directly installed. This is only recommended, if you do not have python in a similar version already installed on your computer. You can also manually install and/or update python or its packages before or after this setup.

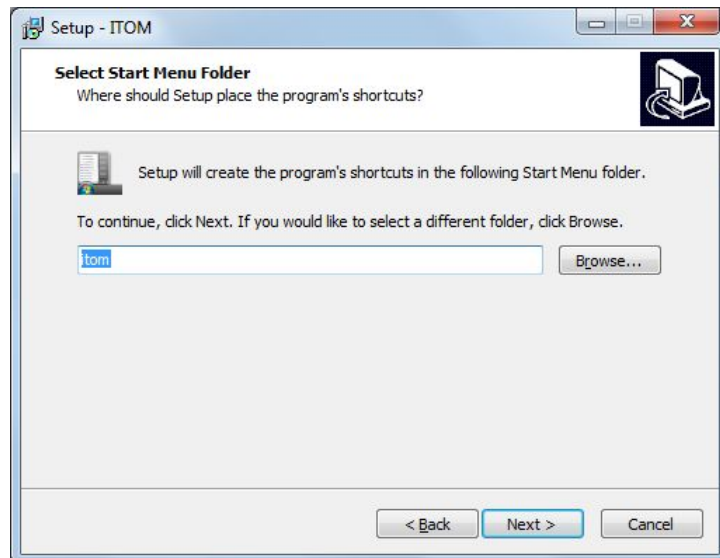


Figure 2.5: Select the name of **itom** in your Windows start menu.

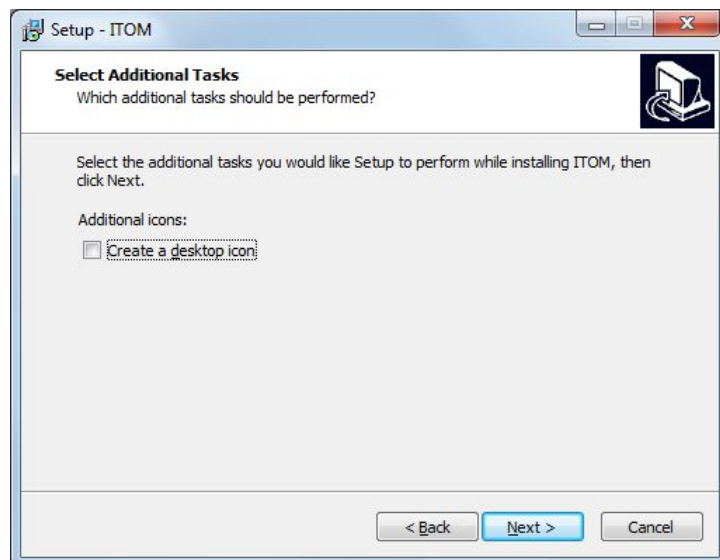


Figure 2.6: Choose whether you want to have an **itom** shortcut on your desktop

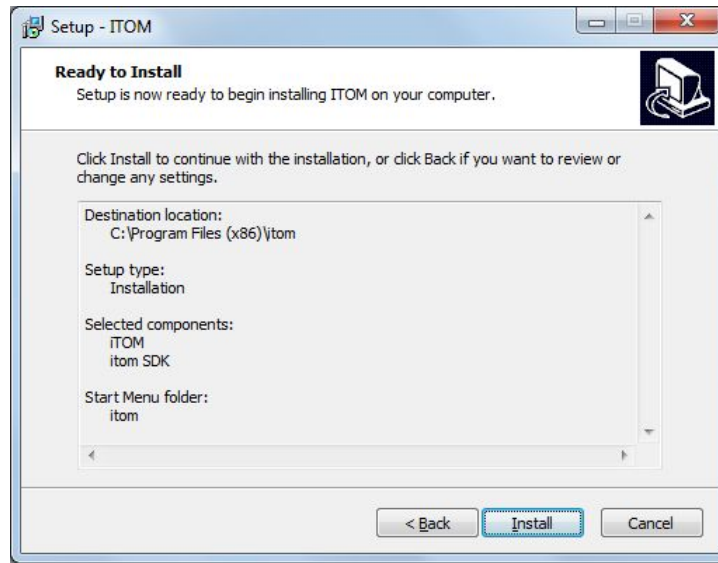


Figure 2.7: Now, a summary of the installation steps is given. Press next if you want to start the installation...

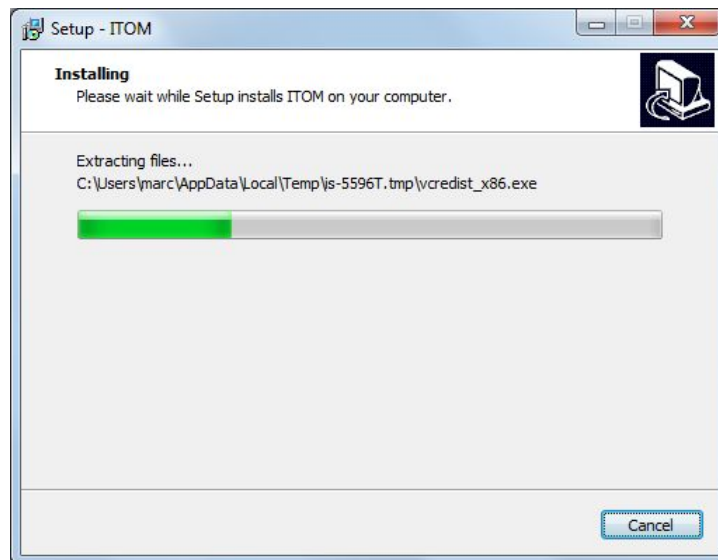


Figure 2.8: The installation is executed now. **itom** is not copying any files in another folder than the indicated program folder (besides python or any python-packages). However this setup creates an application entry in the Windows registry in order to allow an uninstall by the default Windows control panel and to check if any version of **itom** already has been installed. When uninstalling **itom**, the registry entry is removed, too.

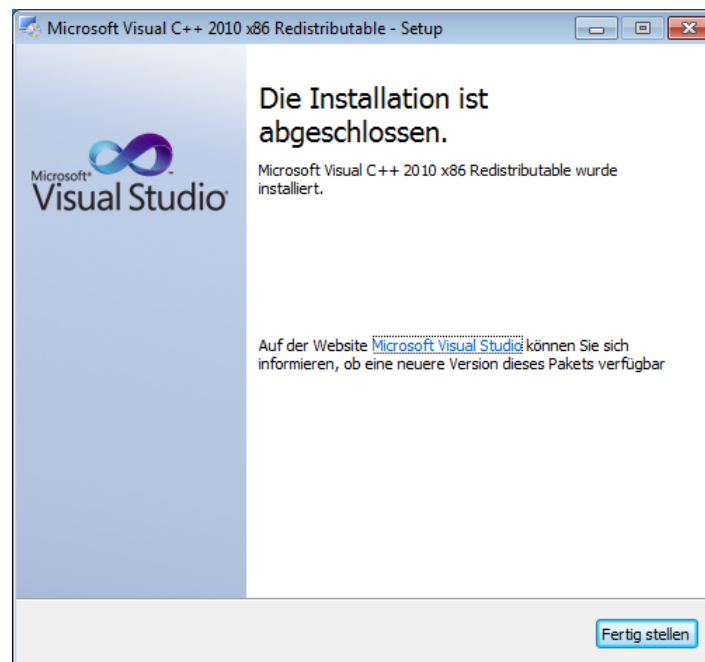
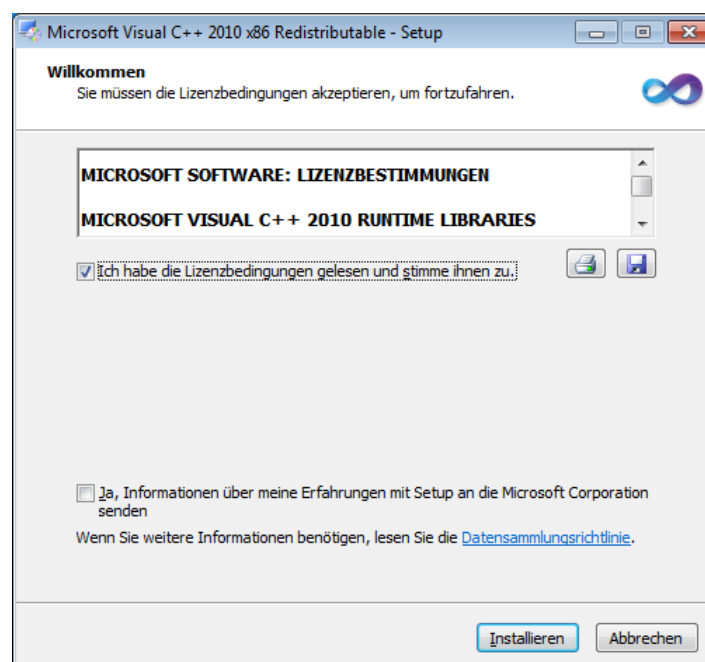


Figure 2.9: If not already available, the Microsoft Visual C++ 2010 Runtime Libraries are installed now.



Depending on your selected components, python and/or any python packages are now installed:



Figure 2.10: Install Python 3.2.1 for the current user or for all users.



Figure 2.11: Customize your Python installation. We recommend leaving everything as is.



Figure 2.12: You've completed the Python installation as well. We're getting getting closer.

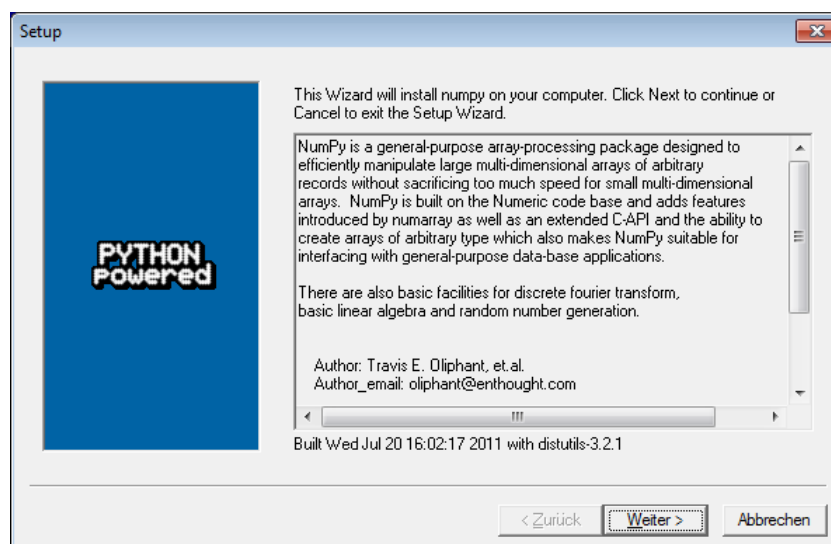


Figure 2.13: Continue with the installation of NumPy.

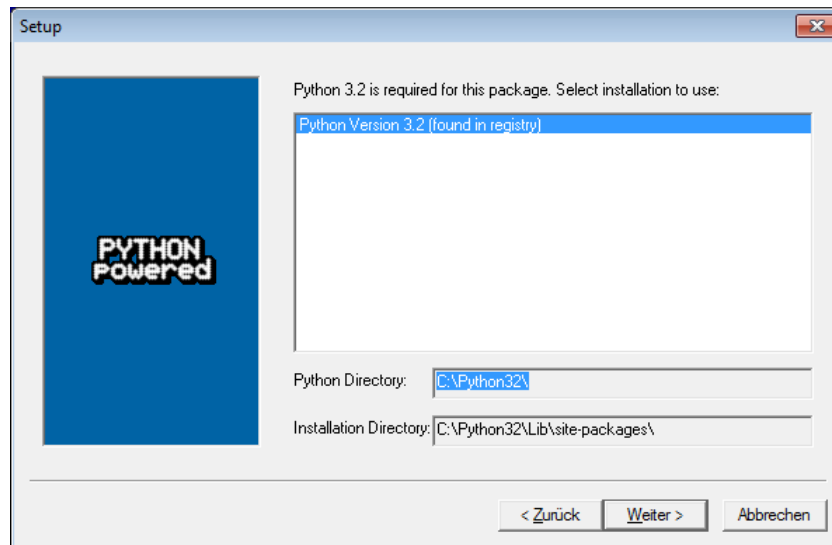


Figure 2.14: You should be able to continue right away.

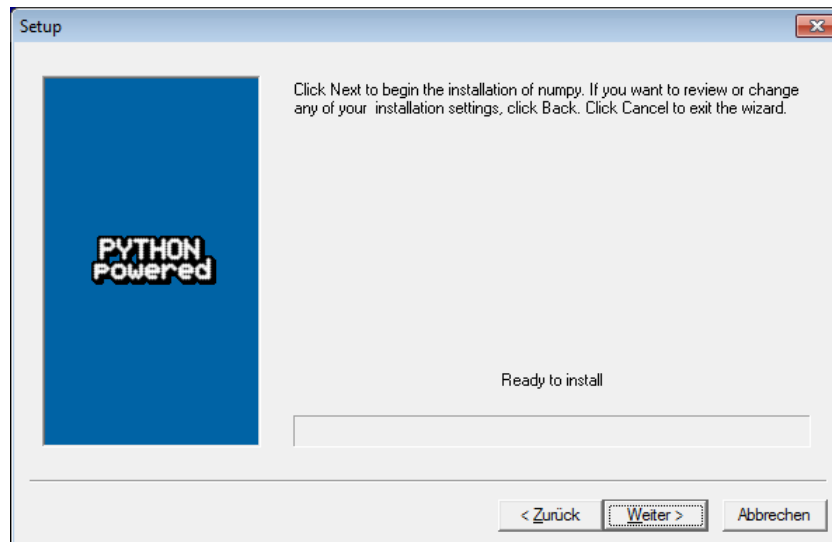


Figure 2.15: Start the NumPy installation.

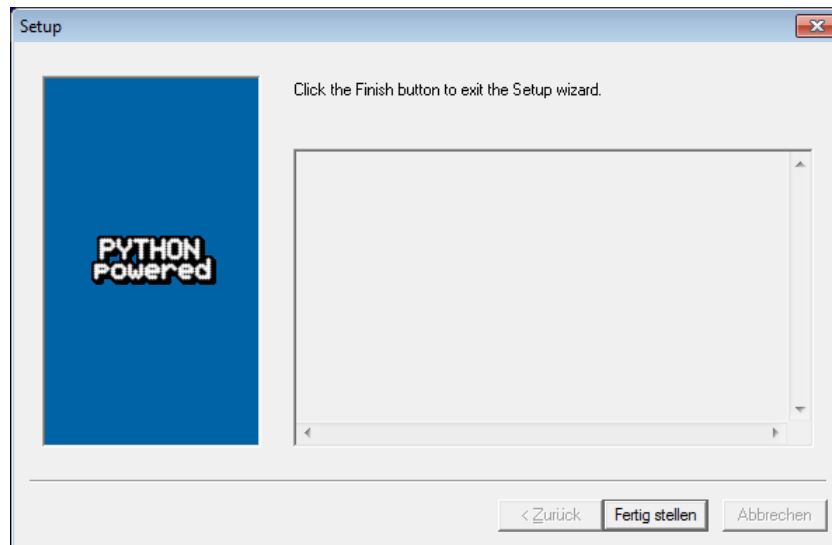


Figure 2.16: Confirm that NumPy was successfully installed.



Figure 2.17: Eventually, we have to install scipy, press “Next” to continue

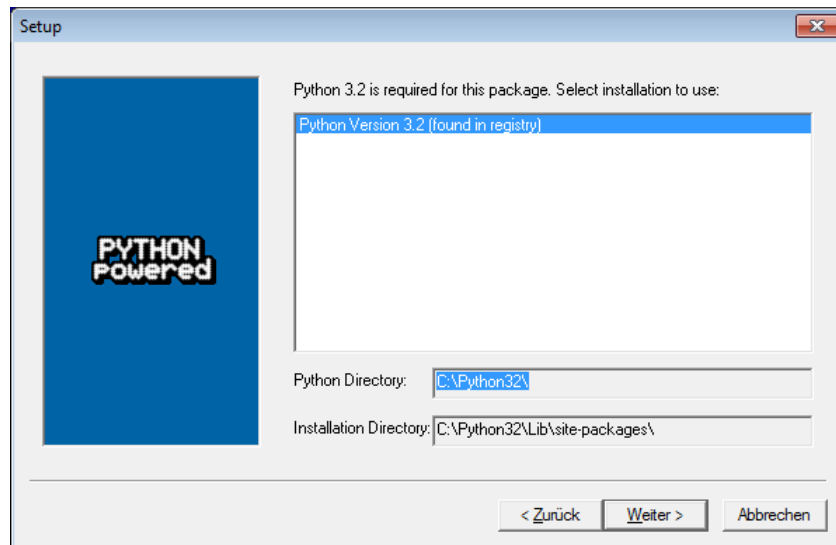


Figure 2.18: Continue.

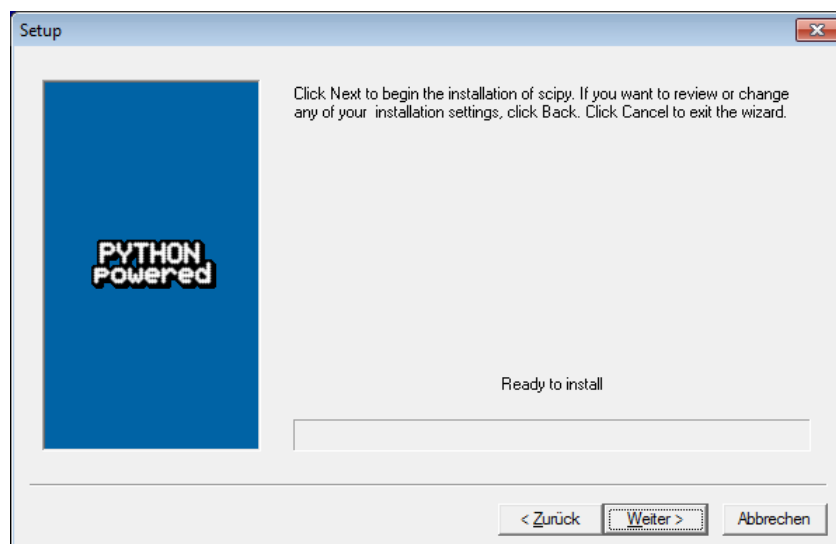


Figure 2.19: Yes, we are ready to install. Proceed please.

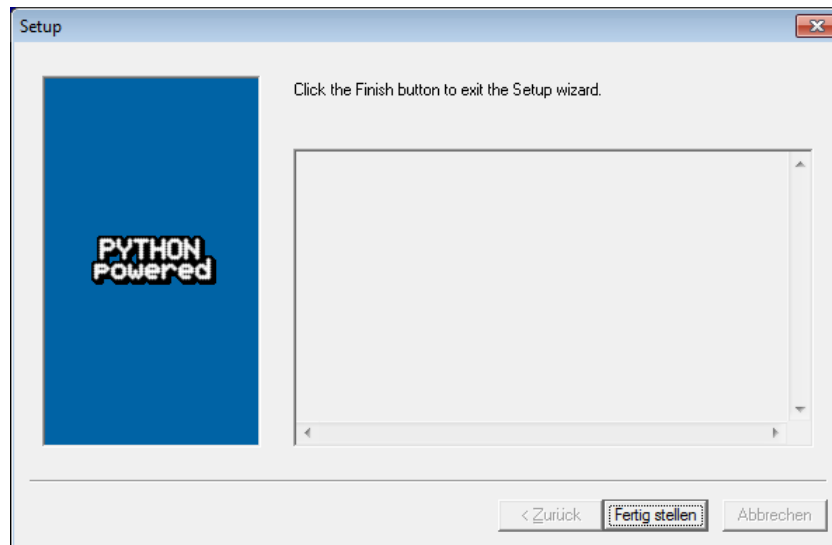
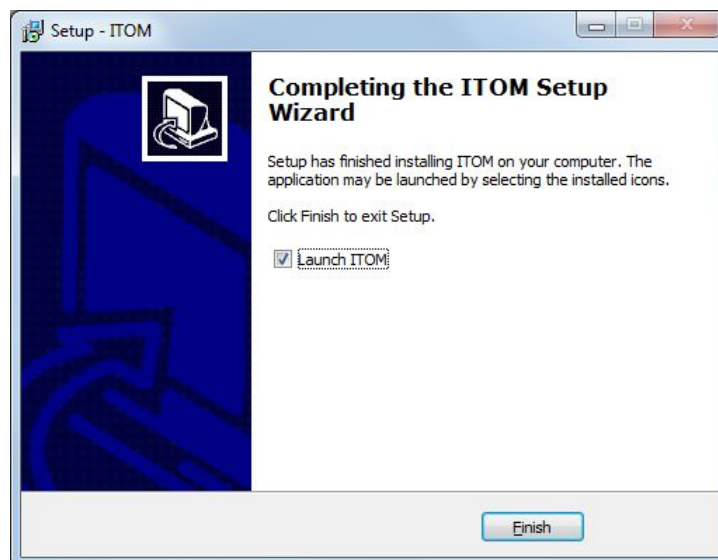


Figure 2.20: Another confirmation, now scipy has been installed successfully.

Finally, the entire setup is finished:



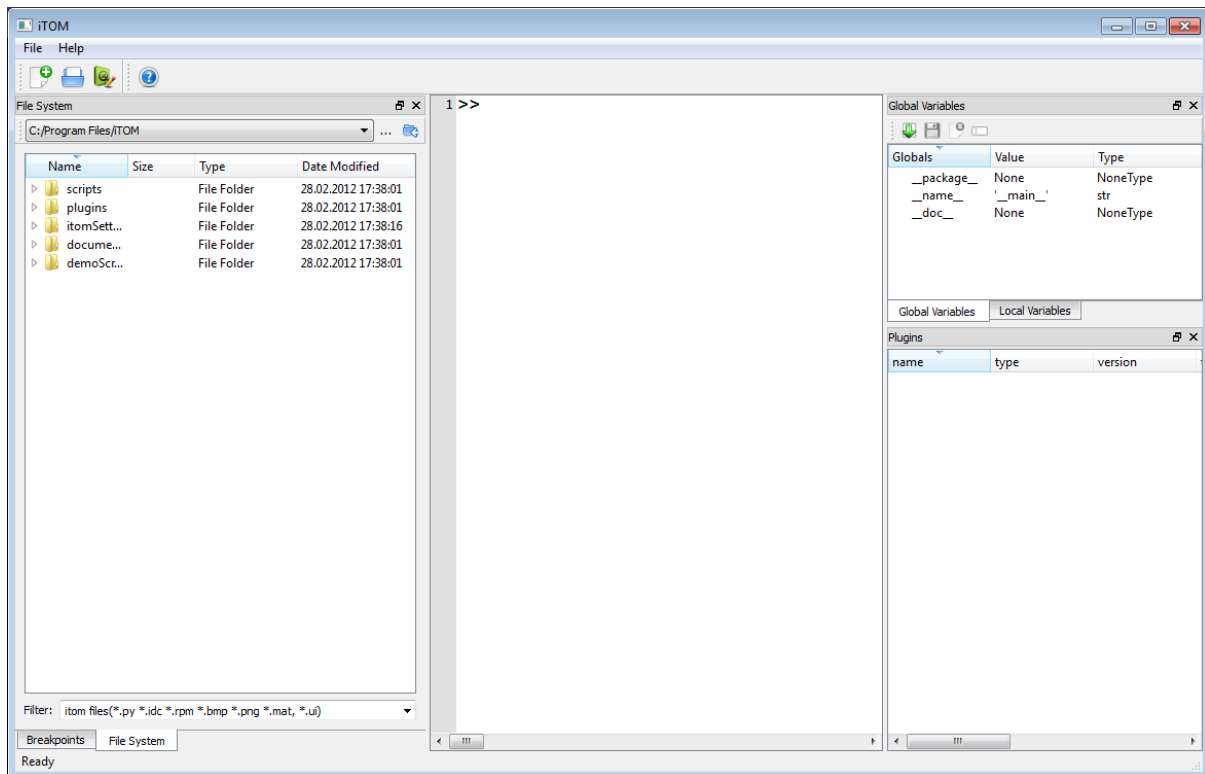
That's it:

2.3 Build from Sources

Alternatively, it is possible to get the sources of **itom** (e.g. clone the latest Git repository from <https://bitbucket.org/itom/itom.git>) and compile an up-to-date version of **itom**. This is recommended for developers (e.g. plugin developers) and required of linux users. Before getting the source files, check the build dependencies below which contain software packages and libraries necessary to build and **itom**.

2.3.1 Build dependencies

This setup lists third-party software packages and libraries that are required if you want to build **itom** from sources. If you run a setup-release of **itom** none of these dependencies (besides a python 3 installation) are required. Most of the following hints address the build on a Windows operating system. However the required packages are

Figure 2.21: This is **itom**!

mainly the same for Linux and most components can directly be obtained by the specific package manager of your Linux distribution.

Software packages

Required Software-Packages

- IDE, Compiler (e.g. Visual Studio 2010 Professional, QtCreator...)
- CMake (recommended 2.8.9 or higher)
- Qt-framework (4.7 or higher, 4.8 recommended, no 5.x)
- QScintilla2
- OpenCV 2.3 or higher (2.4 recommended)
- PointCloudLibrary 1.6 or higher
- Python 3.2 or higher
- Git (git-scm.com) + GUI (e.g. TortoiseGit or GitExtensions) for accessing the remote repository
- Python-Package: NumPy

Optional Software-Packages

- Qt-AddOn for Visual Studio (requires .NET 2.0 framework with SP 1.0)
- Doxygen (for creating the source code documentation)
- Python-Packages: SciPy, Distribute, Sphinx (user documentation generation), scikit-image, matplotlib...

Detailed information

Compiler, IDE (mandatory)

You can use any compiler and integrated development environment (IDE) which is supported by **CMake** (http://www.cmake.org/cmake/help/v2.8.10/cmake.html#section_Generators). On Windows systems, we develop with **Visual Studio 2010 Professional**, whereas we use **QtCreator** for the development under Linux. QtCreator is no generator of CMake, however QtCreator directly supports CMakeLists.txt-files. It is also possible to use the free express edition of Visual Studio.

Note: Please consider that you need to install the Service Pack 1 for Visual Studio 2010 Professional when compiling a 64bit version of **itom**.

CMake (mandatory)

Download **CMake** from <http://www.cmake.org/cmake/resources/software.html> and install it. If possible use any version higher than 2.8.8. CMake reads the platform-independent project files of **itom** (CMakeList.txt) and generates the corresponding project files for your compiler, IDE and platform.

Qt-framework (mandatory)

Download the **Qt-framework** (version 4.7 or higher, branch 5 is not supported yet) from <http://qt-project.org/downloads>. If you find a setup version for your IDE and compiler, you can directly install it. Otherwise, you need to configure and build **Qt** on your computer. This is also required if you want to compile **itom** with Visual Studio for 64bit (see box below).

Create the following environment variables (Windows only - you need to log-off from your computer in order to activate changes to environment variables):

- create an entry **QTDIR** and set it to the *Qt*-base directory (e.g. **C:\Qt\4.8.0**)
- create an entry **QMAKESPEC** and set it to the string **win32-msvc2010** (even if you are compiling for 64bit) or similar (see <http://qt-project.org/doc/qt-4.8/qmake-environment-reference.html#qmakespec>)
- add the following text to the Path variable: **;%QTDIR%\bin** (please only **add** this string, **do not replace** the existing path-entry)

Note: Compiling Qt for 64bit, Visual Studio

This side-note explains how to configure and build Qt for a 64bit build using Visual Studio 2010. The general approach for other configurations is similar.

- Delete files beginning with *sync* from the **%QTDIR%\bin** directory (in order to avoid the requirement of Perl during compilation, which is not necessary in our case).
- 64bit: Open **Visual Studio Commandline x64 Win64 (2010)** in your Start-Menu under *Microsoft Visual Studio >> Visual Studio Tools*.
- (for 32bit always use the **Visual Studio Commandline (2010)**)
- change to Qt-Dir by typing:

```
cd %QTDIR%
```

into the command line.

- configure Qt-compilation by executing the command:

```
configure -platform win32-msvc2010 -debug-and-release -opensource -no-qt3support -qt-sql-odbc
```

- choose the option **open source version** and accept the license information during the configuration process. The configuration may take between 5 and 20 minutes.
- now start the time-intense compilation process (1 to 5 hours) by executing the command:

```
nmake
```

If you want to restart the entire compilation you need to completely remove any possible older configuration. Then open the appropriate Visual Studio command line and execute:

```
nmake distclean
```

Qt-Visual Studio-AddIn (optional, only for Visual Studio, not necessary for QtCreator)

If you want to have a better integration of **Qt** into **Visual Studio** (e.g. better debugging information for Qt-types like lists or vectors), you should download the **Qt-Visual Studio-AddIn** (1.1.11 for Qt 4.8.x, 1.1.10 for Qt 4.7.x) from <http://qt-project.org/downloads#qt-other> and install it. Since we are using **CMake** it is not mandatory to use this **AddIn** like it is usually the case when developing any Qt-project with Visual Studio. Therefore it is also possible to use the Express edition of Visual Studio, where you cannot install this add-in. The **Qt Visual Studio AddIn** requires that you have the **.NET framework 2.0 SP 1** installed on your PC.

Note: Sometimes, there are problems when starting Visual Studio with an installed Qt-AddIn. In case that any component cannot be registered, as warned by a message-box when starting Visual Studio, you should check the bug and its fix described at <https://bugreports.qt-project.org/browse/QTVSADDINBUG-77>. In most cases it was sufficient to register the library **stdole.dll** using the tool **gacutil.exe** from the **Microsoft SDKs/Windows/v7.0A/bin** subfolder of your standard program folder. Start a windows commandline and move to the directory on your computer where the executable program **gacutil.exe** is located, then type:

```
gacutil.exe -i "C:\Program Files (x86)\Common Files\microsoft shared\MSEnv\PublicAssemblies\stdole.dll"
```

QScintilla2 (mandatory)

Download **QScintilla** (2.6 or higher) from <http://www.riverbankcomputing.com/software/qscintilla/download> and build the debug and release version. The original QtCreator project file of QScintilla finally copies the entire output to the **bin** directory of **Qt** so that no other settings need to be adapted. However, the original project settings are not ready for a multi-configuration build in **Visual Studio**. As a result, you need to adapt the Qt-project file. To do this follow these steps:

- Copy the downloaded files to a directory of your choice (preferably **NOT** the windows program directory, we are assuming in the following that you placed them in **C:\QScintilla2**)
- Open the Visual-Studio 2010 32-bit commandline
- **Open the file C:\QScintilla2\Qt4\QScintilla.pro in a text editor and replace the line CONFIG with::**
CONFIG += qt warn_off debug_and_release build_all dll thread
and add the lines:: CONFIG(debug, debug|release){ TARGET = \$\$join(TARGET,,d) }
(see also: http://www.mantidproject.org/Debugging_MantidPlot)
- If you had a previous installation of QScintilla, delete the directory **%QTDIR%\include\Qsci** as well as the files called **qscintilla2.dll** and **qscintilla2d.dll** in the directory **%QTDIR%\bin**
- **Execute the following commands from the commandline console**
 - cd C:\QScintilla2\Qt4
 - nmake distclean
 - QTDIR%\bin\qmake qscintilla.pro spec=win32-msvc2010
 - nmake
 - nmake install
- copy the library files **qscintilla2.dll** and **qscintilla2d.dll** from **%QTDIR%\lib** to **%QTDIR%\bin**

An easier approach is to get the sources from **\Obelix\software\m\ITOM\Installationen\4. QScintilla2** (ITO only) and copy the folder **QScintilla2.6** to a directory on your hard drive (e.g. **C:\QScintilla2.6**, avoid Windows program directory due to restrictions in write access). Open your Visual Studio Command Line and change to the

directory of **QScintilla** on your hard drive. Just execute the batch file **qscintilla_install.bat** and answer the given questions.

Note: If the batch file breaks with some strang error messages, please make sure, that the location where **QScintilla** is installed is writable by the batch file (e.g. the Windows program directory under Windows Vista or higher). Therefore it is recommended to locate **QScintilla** in another directory than the program-directory.

OpenCV (mandatory, 2.3 or higher)

You have different possibilities in order to get the binaries from OpenCV:

1. Download the OpenCV-Superpack (version 2.3) from <http://sourceforge.net/projects/opencvlibrary/files/opencv-win/2.3/>. This superpack is a self-extracting archive. Unpack it. The superpack contains pre-compiled binaries for VS2008, VS2010, MinGW in 32bit and 64bit. (Later map the CMake variable **OpenCV_DIR** to the **build** subdirectory of the extracted archive).
2. Download the current setup (version 2.4 or higher) from <http://opencv.org/> and install it. This installation also contains pre-compiled binaries for VS2008, VS2010 and MinGW. In this case map **OpenCV_DIR** to the **opencv/build** subdirectory.
3. Get the sources from OpenCV and use CMake to generate project files and build the binaries by yourself. Then map **OpenCV_DIR** to the build-directory, indicated in CMake.

Finally, add the appropriate bin-folder of OpenCV to the windows environment variable: - VS2010, 32bit: Add to the path-variable: **;%C:\OpenCV2.3\buildx86vc10\bin** (or similar) - VS2010, 64bit: Add to the path-variable: **;%C:\OpenCV2.3\buildx64vc10\bin** (or similar)

Changes to the environment variable only become active after a re-login to windows.

PointCloudLibrary (mandatory, 1.6 or higher)

The PointCloud-Library is a sister-project of OpenCV and is able to work with large point clouds.

The binaries can be loaded from the website <http://www.pointclouds.org/downloads/windows.html>. Depending on 32bit or 64bit execute the **AllInOne-Installer for Visual Studio 2010**. The installation directory may for example be **C:\PCL1.6.0**. Information: Please install the PCL base software including all 3rd-party packages, besides OpenNI. You don't have to install OpenNI, since this is only the binaries for the communication with commercial range sensors, like Kinect.

If you want to debug the point cloud library (not necessary, optional) unpack the appropriate zip-archive with the pdb-files into the bin-folder of the point cloud library. This is the folder where the dll's are located as well.

Add the path to the bin-folder of PointCloud-library to the windows environment variable:

- Add to the path-variable: **;%C:\PCL1.6.0\bin** (or similar)

Python (mandatory, 3.2)

Download the installer from <http://www.python.org/download/> and install python in version 3.2. You can simultaneously run different versions of python.

NumPy (mandatory)

Get a version of NumPy that fits to python 3.2 and install it. On Windows, binaries for many python packages can be found under <http://www.lfd.uci.edu/~gohlke/pythonlibs/>.

Sphinx (optional)

The Python package **Sphinx** is used for generating the user documentation of **itom**. You can also download sphinx from <http://www.lfd.uci.edu/~gohlke/pythonlibs/>. However, sphinx is dependent on other packages, such that it is worth to get the package **distribute** (see below) first. Then open a command-line (cmd.exe) and move to the directory **[YourPythonPath]/Scripts**. Type the following command in order to download **sphinx** including dependencies from the internet and install it:

```
easy_install -U sphinx
```


Distribute (optional)

Distribute is a python package, that can be used for easily downloading and installing other python packages, like *Sphinx*. Download the latest version (file ending with tar.gz) of **Distribute** from <https://pypi.python.org/pypi/distribute> and unpack it to any temporary directory on your hard drive. Open a command line and switch to the directory of **Distribute**.

Assuming that Python is located under **C:\Python32**, execute the following command:

```
C:\python32\python.exe setup.py install
```

The **Distribute** is installed and you can use the **easy_install** tool (see **Sphinx** installation above).

Other python packages (optional)

You can always check the website <http://www.lfd.uci.edu/~gohlke/pythonlibs/> for appropriate binaries of your desired python package.

Note: If you use any python packages depending on NumPy (e.g. SciPy, scikit-image...) try to have corresponding versions. If your SciPy installation is younger than NumPy, some methods can not be executed and a python error message is raised, saying that you should update your NumPy installation.

2.3.2 Get sources from Git

The core of **itom**, some plugins as well as designer-plugins are hosted in different Git-repositories on *bitbucket.org*.

GIT is a distributed version and source code management system, that allows several developers to work on the same source code simultaneously. **itom** is hosted under GIT at bitbucket.org.

The different parts can be found at the following resources:

Project	Website	Git-Repository (https)	Git-Repository (ssh)
itom (core)	https://bitbucket.org/itom/itom	https://bitbucket.org/itom/itom.git	git@bitbucket.org:itom/itom.git
plugins	https://bitbucket.org/itom/plugins	https://bitbucket.org/itom/plugins.git	git@bitbucket.org:itom/plugins.git
designer- Plugins	https://bitbucket.org/itom/designerplugins	https://bitbucket.org/itom/designerplugins.git	git@bitbucket.org:itom/designerplugins.git

In order to get the sources you want, clone the specific repository to a source-folder on your computer. If you want to receive data by *ssh*, you need your own account on *bitbucket.org*, where you public *ssh* has been set under your personal settings.

2.3.3 Build with CMake

In this chapter, you will learn how to create the project files from the sources, contained in the specific Git-repositories (*Get sources from Git*), using the open source generator **CMake** (<http://www.cmake.org>). The main concept of CMake will be introduced in the following section, followed by a short explanation of process of building the core of **itom**.

Concept of CMake

CMake is a platform independent make system for C or C++ based software projects. Therefore the sources only contain the code base and some simple textfiles (e.g. CMakeLists.txt) that describe how a project must be configured such that the code base can correctly be compiled.

When calling the CMake GUI, the configuration files (CMakeLists.txt) will be analyzed. After indicating your desired compiler or generator of your preferred development system (see http://cmake.org/cmake/help/v2.8.8/cmake.html#section_Generators), CMake will generate the specific makefiles or project files for your generator. Then you can call your development environment or compiler like Visual Studio in order to open the recently created project files and finally build your application.

Experienced Visual Studio user are used to locating the source files (.h, .cpp...), the visual studio project files, the intermediate object files and the final compiled application in the same folder structure. This is different for projects that are initiated using CMake, since this process is closer to the base philosophy of Linux.

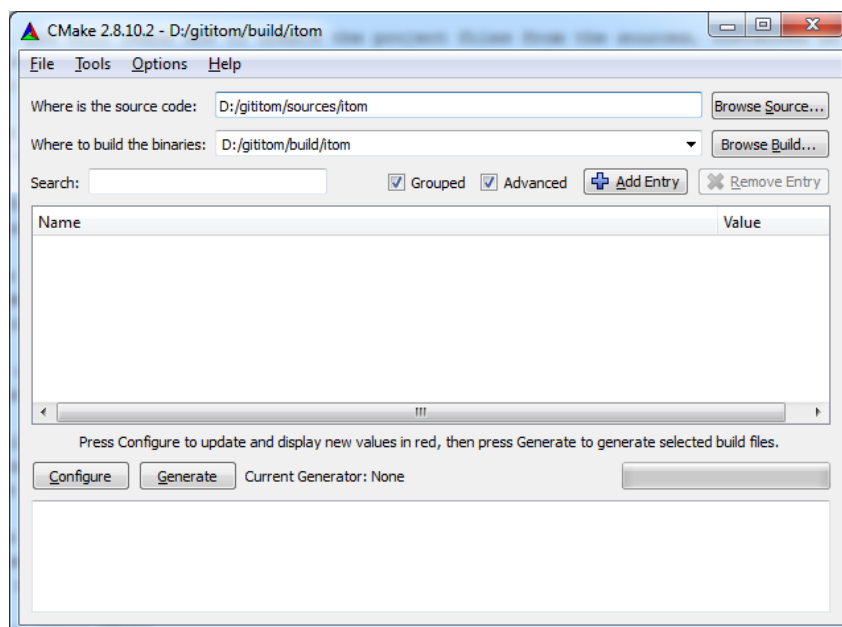
In CMake you will have one folder that only contains the code base, some further resource files and the general CMakeLists.txt-files. CMake is called to analyze this source-folder and to generate the project files in another, independent folder, that is denoted build-directory in the following text. You will only work in this build-directory, that is also the directory, where the intermediate files as well as the compiled executable are located.

All itom plugins can have one or more own build-directories, such that the code base for some or all plugins can also be totally delocated from itom itself. When generating the project files for any plugin using CMake as well, you are asked where the build-directory of itom is located on your harddrive. This is finally used in order to copy the relevant output of every plugin to this itom application directory in a post-build step of every plugin or designer-plugin.

Generating itom using CMake

These are the steps you need to execute if you want to generate and build **itom** from sources using CMake:

1. Get the sources of **itom** core from the Git repository (see [Get sources from Git](#)).
2. Start **CMake GUI**.

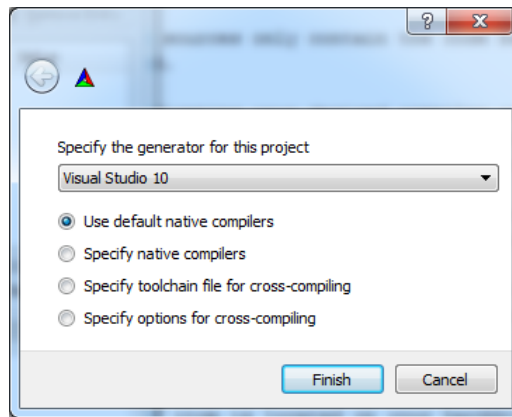


Use the **browse source...** button in order to select the location of the itom source code, e.g. *D:/gititom/sources/itom*. Use **browse build...** to indicate the build directory for itom, e.g. *D:/gititom/build/itom*. If this directory does not exist, it will be created during the configuration.

3. Now click the **Configure** button in CMake. Then, the file *CMakeLists.txt* in your source folder is analyzed. If you create this project for the first time, you are now asked for your desired generator.

If you compile using Visual Studio, choose **Visual Studio 10** or **Visual Studio 10 Win64**. For the **QtCreator** choose **MinGW Makefiles** on Windows or **Unix Makefiles** on Linux machines. Accept your choice by clicking **Finish**.

4. Since **itom** is dependent on other libraries (like OpenCV), CMake is able to automatically guess the location for many common libraries. The resulting variables including many other settings for configuring the compiler will then be shown in the huge listbox in CMake. If CMake added a variable for the first time, this entry is displayed using a red background. Clicking **Configure** again, the red background becomes white.
5. Usually, the configuration process will fail when clicking the button for the first time, since you need to help CMake with some manual changes. The detailed problem is always written in the log-text at the bottom



part of the CMake GUI. Help CMake with some further hints (see below) and press **Configure** again. The current configuration is finally cached in a specific cache-file, that also is located in the build directory such that any reconfiguration will firstly read the content of this cache file.

6. Click the **Grouped** checkbox, in order to get a better overview of the current configurations. Further variables are shown when clicking the **Advanced** checkbox. Usually you need to manually set all or some of the following variables (in this order):
7. relick **Configure** until no red-backgrounded values are visible and no errors appear. Then click **Generate** to create the project files.
8. If you are using Visual Studio, open it now and load the solution **itom.sln**, contained in the build directory. This solution now contains several sub-projects. All projects besides **ALL_BUILD** and **ZERO_CHECK** are itom specific sub-projects. The easiest way to compile itom is to mark qitom as start-project (right click on qitom and select as start project) and build **qitom**. The dependent projects **dataobject**, **pointcloud** and **qpropertyeditor** are automatically build before building qitom. The project **ALL_BUILD** will firstly start **ZERO_CHECK** and then build all projects contained in the solution in the dependent-optimal order. For more information about **ZERO_CHECK** see the Update project settings section below.
9. If you are using **QtCreator**, open it and call the file *CMakeLists.txt* from the itom source folder. Additionally chose the same output folder defined in the CMake GUI. QtCreator can now read the cached configurations from CMake. The **QtCreator** will also run CMake, a process that should be safe and should not fail, as everything should have been properly configured in the previous steps using CMake GUI. Usually you don't need the CMake GUI with **QtCreator**, however you can not easily change any settings directly in **QtCreator**.

Generating any plugin or designer-plugin using CMake

In principle, the build of plugins or designer-plugins follows the same basic steps than the build of **itom** itself. However, both plugins and designer-plugins need some files and libraries, that are contained in the **SDK** of **itom**. If you installed **itom** from a setup, the **SDK** is already available, if you built **itom** from sources, you first need to compile **itom** (at least the libraries *dataObject*, *pointCloud* and *qpropertyeditor*) in **Debug** and **Release** before continuing to configure and to generate any plugin project.

In some cases, multiple plugins or designer-plugins are bundled together in one main directory. You have the choice to either build one project with multiple sub-projects (which is a convenient way) or to build one project for each plugin. If you want to generate all plugins, directly use the main directory as source directory in CMake. If you only want to generate single plugins, directly use the corresponding subfolder as source directory. Any source directory must always contain a file *CMakeLists.txt*.

When configuring any plugin or designer-plugin for the first time, you will probably see that either the variable **ITOM_SDK_DIR** or **OPENCV_DIR** is missing (or both), meaning that their values are set to **ITOM_SDK_DIR-NOTFOUND** or **OPENCV_DIR-NOTFOUND**. First set the variable **ITOM_SDK_DIR** to the **SDK**-directory of **itom**. This is either contained in the installation path of **itom** created by the setup or in the build-directory of **itom** if built from sources. Afterwards set **OPENCV_DIR** to the build-directory of **OpenCV**

(see *install-depend-opencv* for details where the build-directory of **OpenCV** is). If one of these variables is not correctly set, other depending errors may be displayed in **CMAKE**, like:

```
CMake Error at AMMS/CMakeLists.txt:25 (find_package):
By not providing "FindOpenCV.cmake" in CMAKE_MODULE_PATH this project has
asked CMake to find a package configuration file provided by "OpenCV", but
CMake did not find one.
```

If the path to **ITOM_SDK_DIR** is set correctly and you pressed **Configure** again, the variable **ITOM_DIR** should also contain a valid folder (it should map to the build directory of **itom**, containing folders *plugins* and *designer*).

Update CMake

If you want to change any project settings or want to add or delete some source files from any project, then change the corresponding *CMakeLists.txt* files. If you need to change further settings, you should open the CMake GUI and re-configure your project. When done, press the **Generate** button. If Visual Studio is currently opened with the same project, the change in the project files is automatically recognized and you will be asked if you want to reload the project. Accept this. If you only changed some *CMakeLists.txt* files, you can also run the project **ZERO_CHECK** in Visual Studio. Visual Studio will call CMake in the command line in order to check for changes. The rest is the same as calling the CMake GUI. **QtCreator** is also able to directly run CMake in order to check the project for changes.

Additional third-party libraries

Some plugins are dependent on additional third-party libraries. The **itom** SDK provides some find-methods for several commonly used libraries. You will be informed by CMake if any library could not be found by this automatic find-method, if a variable of the following style and content is available:

```
LIBRARYNAME_DIR = LIBRARYNAME_DIR_NOTFOUND
```

LIBRARYNAME is then replaced by the real name of any missing library. See the following table for some common names **LIBRARYNAME**:

Library-name	What to do?
FFTW	Set FFTW_DIR to the directory of your fftw installation. This directory must contain some library files (e.g. libfftw3-3, libfftw3f-3...)
GLEW	Set GLEW_DIR to the directory where the binaries of glew are located on your computer.

FFTW

Some plugins of **itom** are using the library **fftw** in version 3. This library is always used as shared library. Do the following steps in order to get **FFTW 3** (on Windows):

1. Download the windows binaries (32bit or 64bit, depending on itom) from <http://www.fftw.org/install/windows.html>.
2. Unpack the binaries (e.g. at **C:fftw3.3**)
3. Now open the Visual Studio Command Line:
 - If you are compiling with 32bit, go to **Microsoft Visual Studio 2010 >> Visual Studio Tools >> Visual Studio-Command Line (2010)** in the Windows start-menu and open it.
 - If you are compiling with 64bit, go to **Microsoft Visual Studio 2010 >> Visual Studio Tools >> Eingabeaufforderung von Visual Studio x64 Win64 (2010)** in the Windows start-menu and open it.
 - In the opened command line, change to the FFTW directory, e.g. by typing **cd C:fftw3.3** and execute the following commands:

```
lib /def:libfftw3-3.def  
lib /def:libfftw3f-3.def  
lib /def:libfftw3l-3.def
```

4. Close the command line

If you are generating project files for itom plugins or designer plugins, you probably will sometimes get a FFTW-group. Set FFTW_DIR to **C:fftw3.3** and press configure again. All dependent files should be found.

GLEW

For Windows users:

Download the glew binaries from <http://glew.sourceforge.net/index.html> and unzip them to any folder. The CMake variable **GLEW_DIR** should then point to that folder.

2.4 Plugins, Designer-Plugins

Each plugin or designer plugin enhances the core-functionality of **itom** and is compiled in its own project. Therefore the installer or sources of **itom** do not contain any plugin. Every plugin is distributed as a library file (*dll*, *so*,...) and - if necessary - other files.

2.5 Get this help

The user documentation of **itom** can be distributed in various formats. The main format is called **qthelp**, such that the documentation is displayed in the **Qt**-Assistant that is directly accessible from the **itom** GUI. On windows PCs it is possible to compile the help as a Windows Help Document (*chm*) or to create latex-files from the help, in order to create a pdf-document. The base format of all these formats is a collection of *html*-pages.

If you compiled **itom** from sources, no compiled documentation file is provided. Therefore, you need to compile the help by yourself:

2.5.1 Build documentation

Necessary tools

In order to be able to build the documentation, you need to have some tools installed on your computer:

1. Doxygen

Doxygen is a source code documentation tool, that parses your C++-source code and extracts the documentation strings for all methods. Additionally, it displays the entire class- and file-structure of your project. **itom**'s user documentation sometimes uses results from Doxygen in order to show relevant C++-structures for programming plugins.

You will find **Doxygen** under <http://www.stack.nl/~dimitri/doxygen/>. Goto the download section and get the latest release. Windows users can directly download the binaries provided as setup version. Install **Doxygen** on your computer.

2. Python-Package Sphinx

The real user documentation is created in **itom** using a python script that needs the **Python** package **Sphinx** (<http://sphinx-doc.org/>). **Sphinx** itself requires other python packages installed on your computer. Therefore we suggest to get **Sphinx** by the python package **distribute** that is able to download the latest version of **Sphinx** including all depending packages. For the Windows operating systems, **distribute** also installs an application **easy_install** in the **PythonScripts** directory. **Easy_install** can be called from any command line in order to get **Sphinx**.

In order to get **distribute** and **easy_install**, execute the following steps:

- Go to <http://pypi.python.org/pypi/distribute> and download the file *distribute-x.x.xx.tar.gz* and unzip to any folder on your computer.
- Open a command line (cmd) and change to the folder (command **cd**) where you unpacked **distribute**.
- Execute the following command (Replace with your correct python-directory):

```
C:/python32/python.exe distribute_setup.py
```

- Now distribute is installed. When done, you can delete the folder the files were extracted to.
- Verify that the application **easy_install.exe** is now available in your *python/scripts* directory (Windows only)

Now you can use **easy_install** in order to get **Sphinx**:

- Open a command line (cmd) and change to the subfolder **Scripts** of your **Python** installation.
- Execute the following command:

```
easy_install -U Sphinx
```

- Now Sphinx is downloaded and Sphinx and its depending packages are installed.

You can also manually download and install Sphinx and its depending packages. Setup-versions of **Sphinx**, **Pygments**, **Jinja2**, **docutils**... are also available from <http://www.lfd.uci.edu/~gohlke/pythonlibs/>. However, you then need to separately install all depending packages of **Sphinx**.

Run doxygen

In your build-directory of **itom**, you will find a folder **docs**. Open its subfolder **doxygen**. There you will find a document **itom_doxygen.dox**. This document contains absolute paths to the source directories of **itom**'s sources. Run doxygen with this document in order to create the source code documentation.

On Windows computers, the easiest way to do this is open **itom_doxygen.dox** with the tool **doxywizard** that lies in the **bin**-folder of you **doxygen** installation. In **doxywizard** go to the *run*-tab and click on the *run*-button.

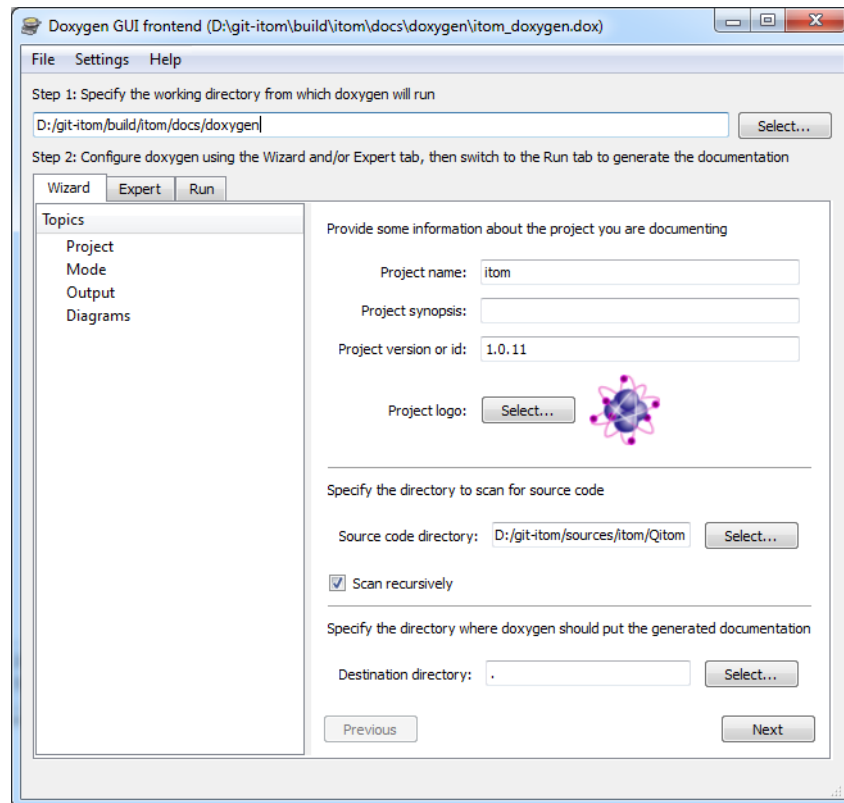
After the build process, a folder **xml** is created in the **doxygen** subfolder of the **docs** folder. This **xml** folder is required afterwards.

Run Sphinx

Now open **itom** and execute the script **create_doc.py** in the folder **docs/userDoc** of the build-directory. The default-builder of the documentation is **qthelp**. If you also want to build the documentation for other builders, you can change the list *buildernames*. The following values are possible:

```
qthelp -> default qthelp format for opening the documentation within itom
htmlhelp -> creates a chm-help format on Windows only
latex -> creates a pdf-document using latex. You need to have latex installed on your computer
```

The output of all build processes are located in the folder **docs/userDoc/build/<buildername>**. The locations of the Windows html-help generator or the latex interpreter are detected when running **CMake** for the **itom**-project. The absolute paths to these tools are automatically inserted into the script **create_doc.py**.



Show documentation in itom

When clicking the *help*-button in **itom** or pressing **F1**, **Qt**'s assistant is opened with a set of documentation files. At first, **itom** checks your **itom** installation for various documentation files. Their latest version is copied into the **help** folder of the build-directory. The search is executed for all **.qch**-files that are located in the **docs/userDoc**-directory and in the folder **plugins** or any subfolder.

After having copied the files, a collection-file is generated (containing all qch-files) and displayed in the assistant. If you have a setup version of **itom**, the help-folder already contains a compiled documentation file, that is displayed in this case.

GETTING STARTED

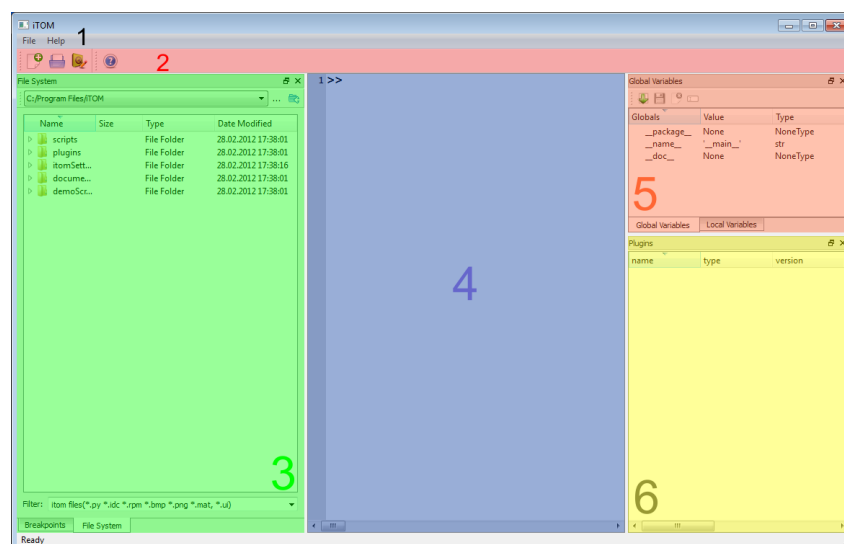
The following chapter is an introduction into the functionalities of **itom** with a number of hands-on examples.

3.1 Quick Start

3.1.1 The GUI

In this section will help you with your first steps in using **itom**. We will start with the main GUI, the standard starting point of your interaction with the software. After startup the **itom** GUI is by default divided into the following parts:

1. Menu bar
2. Quickstart bar
3. File System / Breakpoints
4. Console
5. Global Variables / Local Variables
6. Plugins-Browser



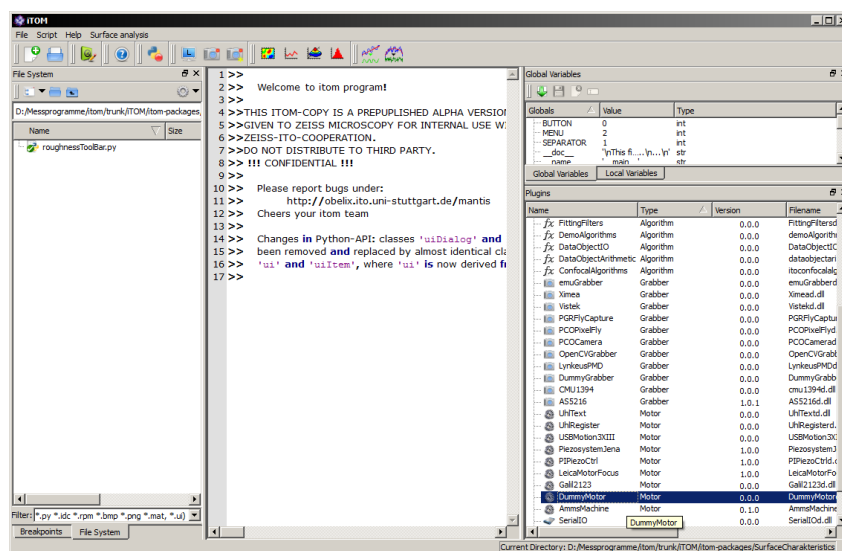
3.1.2 GUI-Operations

The first examples will show you the basic GUI operations together with plugin-handling, data acquisition and saving this data to the harddrive.

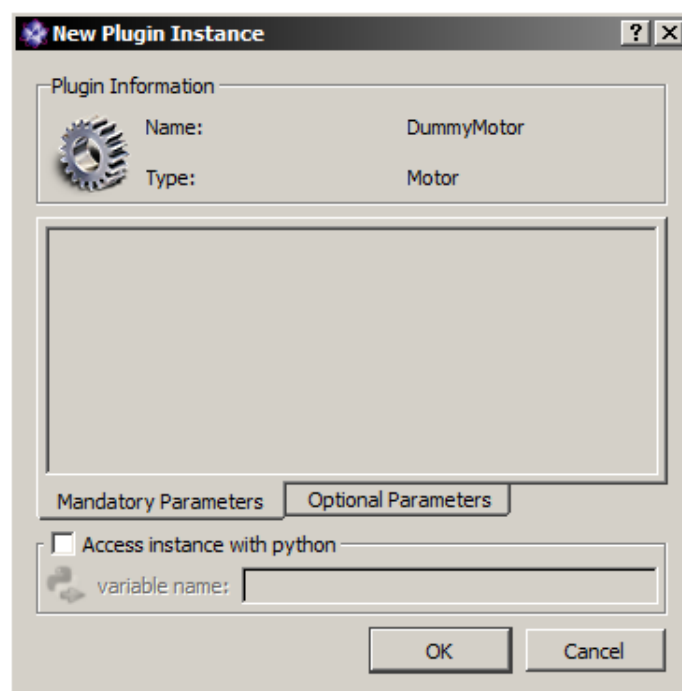
Open and move a virtual stage

Let us learn more about the **itom** -GUI-PlugIn-Interaction using the example of the **DummyMotor**. You will require the **DummyMotor** plug-in.

At first, we have to initialize the stage. The **Plugins-Browser** (No. 6) displays a list of all available Plug-ins, sorted by type. These DLL's are already loaded but not instantiated. To access the functionality of a plug-in we need to create an instance of it. To this end, we select the **DummyMotor** within the **Plugins-Browser** (No. 6). If the **DummyMotor** is not displayed, you need to get or build the plug-in first (refer to *Build with CMake*). We right-click and select 'newInstance'.



A window with initialisation parameters of the plugIn opens.

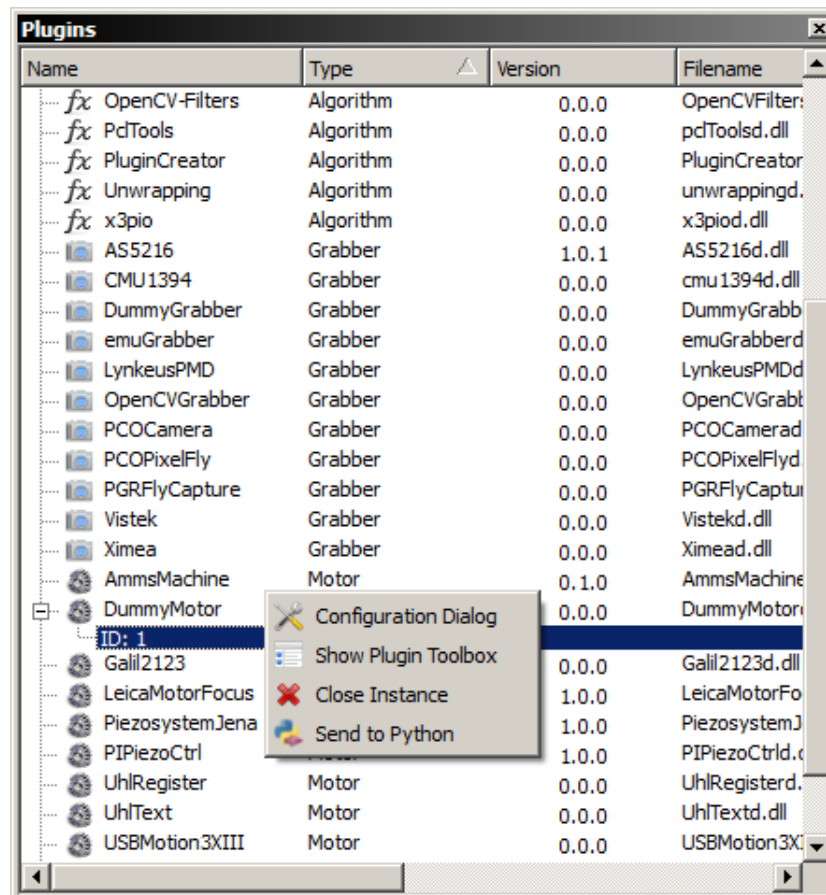


This PlugIn has requires no parameters, so we press OK. More complex plug-ins will require any number of mandatory and optional parameters for correct instantiation at this stage.

If no error message occurs during initialisation we will see a '+' before the **DummyMotor** in the **Plugins-Browser**. This indicates that an instance of the plug-in has been created and is ready for use. We expand the

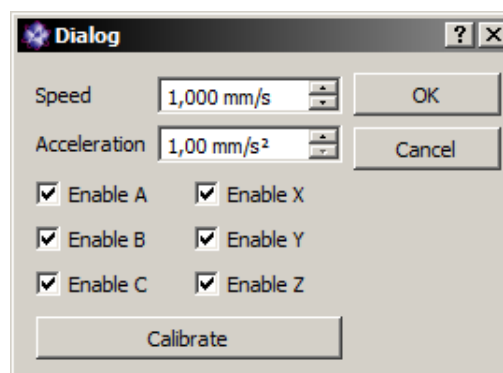
entry **DummyMotor**. We can see the index of the initialised plugin (first time == ID: 1).

If we right-click again on **DummyMotor ID 1**, a context menu opens with the entries:



- Configuration dialog
- Show plugIn Toolbox
- Close instance
- Sent to python

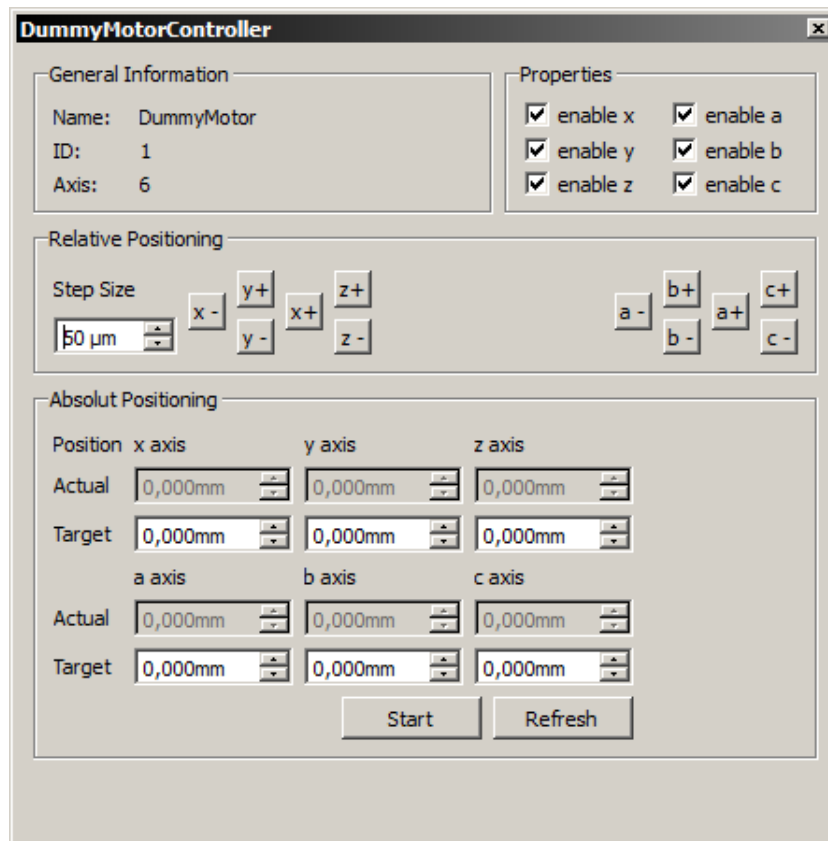
By left-clicking **Configuration dialog**, a window with basic parameter setup functions for this special plugin opens.



As long as this window is open, iTOM will be blocked. So we close this window with the 'ok'-Button.

To move this virtual stage, this plugIn comes with a **Toolbox**. We open this by right-clicking again on **Dummy-Motor ID 1** and then left-click on **Show plugIn Toolbox**.

This opens a non blocking widget usually docked to the GUI-Window with basic control-buttons for moving the motor to the desired position.



At the end we close the plugIn by right-clicking again on **DummyMotor ID 1** and then left-click on **Close instance**.

3.1.3 Scripting

In the second part of this **getting started** we will focus on the script language based on python. The instructions are without lengthy explanations. Use the **tutorial** for more detailed information.

Hello World!

You can get the famous “hello world!” in a very simple way. Just write in the console:

```
1 print("hello world!")
```

The command is executed immediately. Let us proceed to a more advanced example involving a camera.

Getting a snapshot

In this example we will explain how you can open a camera and make a snapshot step by step. Python code can be written and executed in two different ways, either directly, line by line, using the console, or in the form of complete scripts and functions, executed as small programs. In this example we will use the console and will type in our commands directly. Later you can write your code in executable scripts or functions and you can control these functions with your own GUI elements.

You will need the **DummyGrabber-PlugIn** for this script to work.

Step 1: Open the camera

First you have to start your camera device using the respective dataIO plugin. In our example we use the Dummy-Grabber, but in principle you can use any camera connected to your computer for which you have an **itom** plugin. Similarly to the previous example of the **DummyMotor** we need to create an instance of the **DummyGrabber**, only this time, we will instantiate from python instead of from the GUI.

```
1 camera = dataIO("DummyGrabber", 1024, 800)
```

Step 2: Get the list of paramters

With the following command you get a list with all available paramters for your device.

```
1 print(camera.getParamList())
```

Step 3: Setting Parameters

You can set the parameters of the camera as you need it. For example, we might wish to set the upper left corner of the ROI, the integration time (in seconds) of the capturing process or the bit-depth of the captured signal:

```
1 camera.setParam("x0",0)
2 camera.setParam("integration_time",0.05)
3 camera.setParam("bpp",8)
```

Step 4: Getting Parameters

If you are interested in the current value of a paramter you can use the following code to retrieve it.

```
1 sizeX = camera.getParam("sizeX")
2 print("DummyGrabber width: " + str(sizeX))
```

Step 5: Taking a snapshot

Before you can use the camera you have to start it.

```
1 camera.startDevice()
```

Following that you can acquire an image.

```
1 camera.acquire()
```

If you then want to store the image you have to create a new empty dataObject (the dataObject will resize to the correct dimensions later) and save the values of the image in this dataObject.

```
1 data = dataObject()
2 camera.getVal(data)
```

Here you have to be careful, because data is just a reference (**shallow copy!**, no actual data is copied from the camera's memory) to the internal camera memory. If you want to have a deep copy of the image you should use the copy command.

```
1 dataCopy = data.copy()
```

Step 6: Displaying the image

Up to now you have not seen any image. You can plot your acquired image using

```
1 plot(data)
```

or you can watch the live image of the camera using

```
1 liveImage(camera)
```

Step 7: Stop/delete the camera

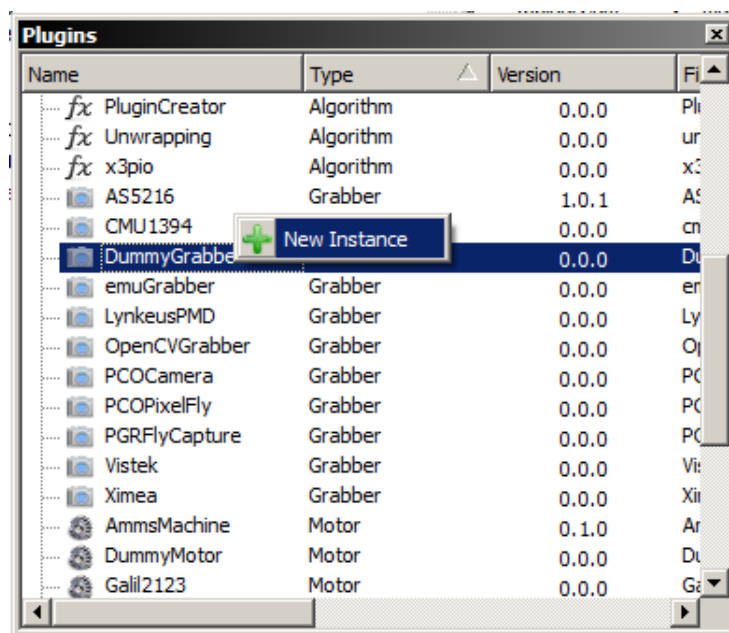
At the end you should stop the camera device. And if you don't need it anymore you can delete the instance of the camera plug-in.

```
1 camera.stopDevice()
2 del camera
```

Alternative Step 1: Open the camera

We can also open the camera via the GUI and make the new instance available for access from python.

Therefore we select the **DummyGrabber** within the **Plugins-Browser** (GUI No. 6). We right-click and select 'newInstance'.

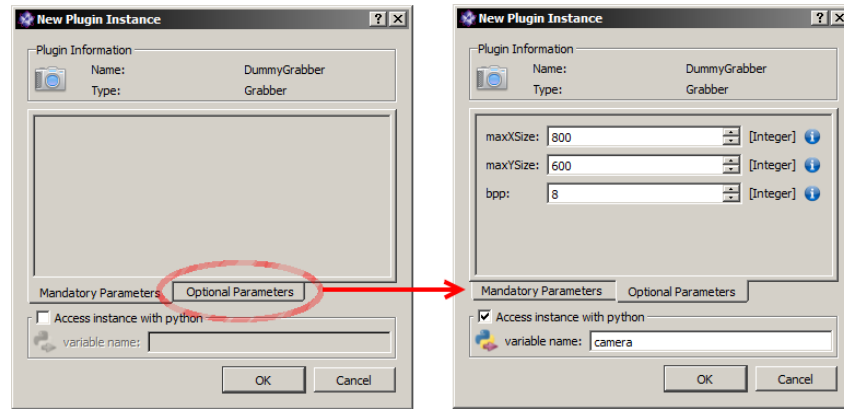


The window with initialisation parameters of the plugIn opens. Select the tab **optional parameters** and insert `maxXSize = 800, maxYSize = 600, bpp == 8`.

Then check **Access instance with python** and type 'camera' into the field **variable name**. Press ok. Now you can proceed with step 2 but since you already set the grabber parameters you can also proceed with step 4.

Apply a median-filter to a snap shot

In the next step, we want to explain how to use filters provided as itom-plugins on the dataObject using the example of a median filter. Instead of executing single python commands from the console, we will now utilize the itom script editor for the first time.



For this example, you will need the **OpenCV-filters-Plugin** and the **DummyGrabber-Plugin**.

Step 1: Open a script editor window

First we open the script editor by left-clicking the **new Script**-button in the button-bar or in the menufilenew Script.



Step 2: Write a simple snap routine

We insert the following code into the editor window. You will recognize the python commands from the previous section.

```
1 camera = dataIO("DummyGrabber")
2 camera.startDevice()
3 camera.acquire()
4
5 dataSnap = dataObject()
6
7 camera.getVal(dataSnap)
8 camera.stopDevice()
9 ...
10 ...
```

Step 3: Retrieve the call for the median filter

What we need in the next step is the correct call for the median-filter, which is defined within the **m++Filter-Plugin**. Each plug-in provides information concerning its usage that is accessible from python. Therefore we switch to the itom-console and type in

```
1 filterHelp("cvMedianBlur")
```

You can see a detailed description of the filter “cvMedianBlur”.

Step 4: Insert the filter-call into the script

We use this information and append our script in the script editor with


```

1 dataFiltered = dataObject() #Destination image
2
3 filter("cvMedianBlur", dataSnap, dataFiltered, 5)

```

to filter the content dataSnap into the new object dataFiltered with a 5 x 5 filter kernel.

The filter also works inplace. That means we can use the input object as the output object, overwriting the input data. To show how this works we add

```

1 filter("cvMedianBlur", dataSnap, dataSnap, 5)

```

Step 5: Add plot commands

To view the results we add

```

1 plot(dataFiltered)
2 plot(dataSnap)

```

to the script.

Step 5: Run the script

To run the script, we press the run button or “F5” on the keyboard. If we have unsaved changes in our script, we are asked to save it.



You should see two 2D-Plots.

Getting online help

The python script language has a lot of methods and classes and **itom** expands these functionalities even further. To retain an overview, python provides a built-in online help. You can type `help(class or method)`. For example:

```

1 help(dataIO)

```

gives a complete help for the class **itom.dataIO**, which is the python-representation for various data communication plugIns (e.g. the dummyGrabber).

By typing

```

1 help(dataIO.getParamListInfo)

```

you get the help for this subfunction of dataIO.

To get an overview over the different itom-plugin functions you can use one of several options.

For Plugin-Initialisation, e.g. the DummyGrabber from the last example, use **pluginHelp(“pluginName”)**.

```

1 pluginHelp("DummyGrabber")

```

For information concerning an already initialized plugIns, e.g. the camera / DummyGrabber from the last example use the member function **.getParamListInfo()**

```

1 camera = dataIO("DummyGrabber")
2 camera.getParamListInfo()

```

To get a list of available (installed and loaded but not necessary instantiated) itom plug-ins of the type filter, use the method **filterHelp()**

```
1 filterHelp()
```

If you want detailed information about one **itomfilter** use `filterHelp("filterName")`

```
1 filterHelp("lowFilter")
```

For user-defined GUI-Elements from plugIns use the function `widgetHelp()`.

3.2 Further Information

For additional information on the features and usage of the **itom** GUI, see GUI *The itom User Interface* For a short introduction to the Python scripting language and use of the **itom** internal `dataObject`, refer to: *Python scripting language* with more information about the `dataObject` *DataObject*

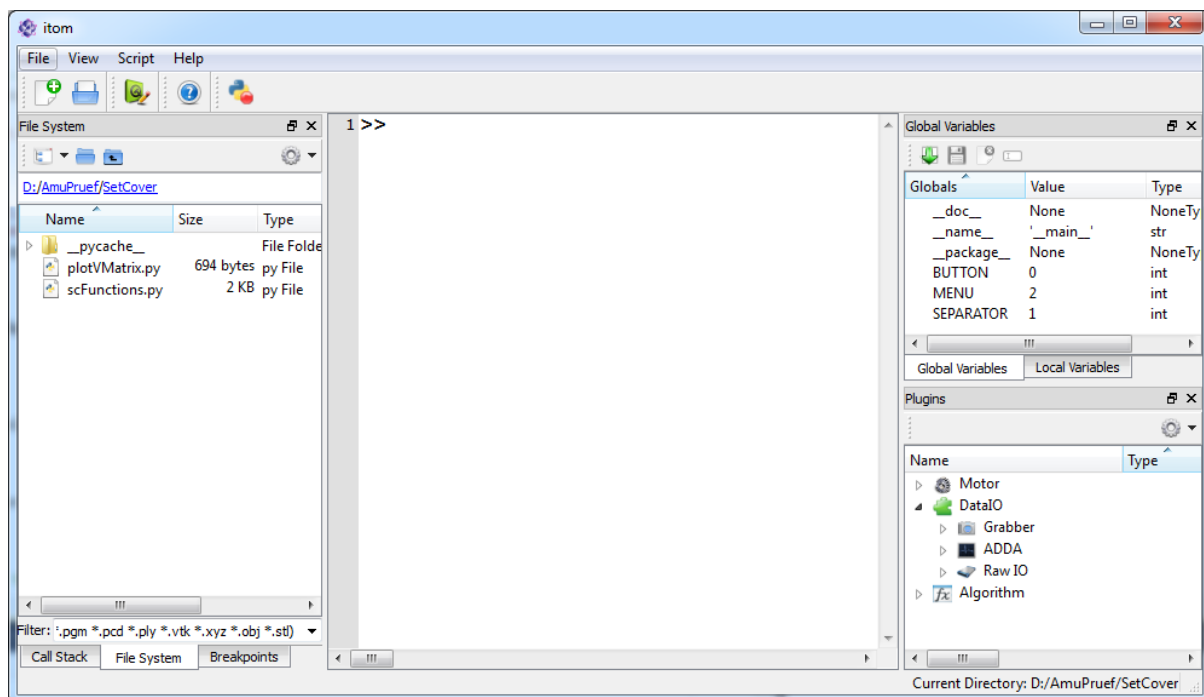
3.3 Tutorials

You can find all scripts used here in the **demo** folder. In the *Demo scripts* section of this manual you can find the overview of all demoscripts which are located in this folder and a short summary what is covered in these scripts.

THE ITOM USER INTERFACE

4.1 Main Window

When starting **itom** a splash screen is shown during the loading process of many components and external plugins. Then, the main window appears like in the following figure:



The GUI consists of a command line widget (**console**) in the center and several **toolboxes**. **itom** contains the embedded scripting language Python 3. Simple python commands can therefore be typed in the console. The toolboxes provide many further functionalities of **itom**. Additionally, every opened actuator or dataIO plugin may provide its own toolbox, that can also be docked into **itom**'s main window.

Besides the console and toolboxes, further important functionalities are provided by the menu or the main toolbars.

4.1.1 Console

The command line in the center of the main window allows to execute single or multi-line python commands. Additionally all messages, warnings and errors coming from python methods or **itom** itself are printed in that console widget, where error messages are highlighted with a red background.

Usually, the last line of the command line shows the **">>"** sign, that indicates that the console is ready for a new input. You can either write a single python command and press the return key in order to execute it or you can

```
Traceback (most recent call last):  
  File "<string>", line 1, in <module>  
RuntimeError
```

write multiline commands. In order to create a new line for this, press the Shift+Return keys, which is a smooth line break (known from other software). After the final command simply press the return key such that the whole command block is executed.

```
>>import time  
time.sleep(2)  
print("I'm awake")
```

The current line or code-block that is executed is highlighted with a yellow background. For multi-line commands, **itom** parses the whole command block and divides it into logical blocks, such that the highlighted background switches from one block to the other one.

In the command line you can use every python command however the command line is not able to handle inputs. Additionally you can use one of the following key-words in order to clear the command line:

```
clc  
clear
```

Instead of typing all commands in the console, write your entire python scripts in the *Editor*.

4.1.2 Main menus and toolbars

This is an overview about the menu structure of itom

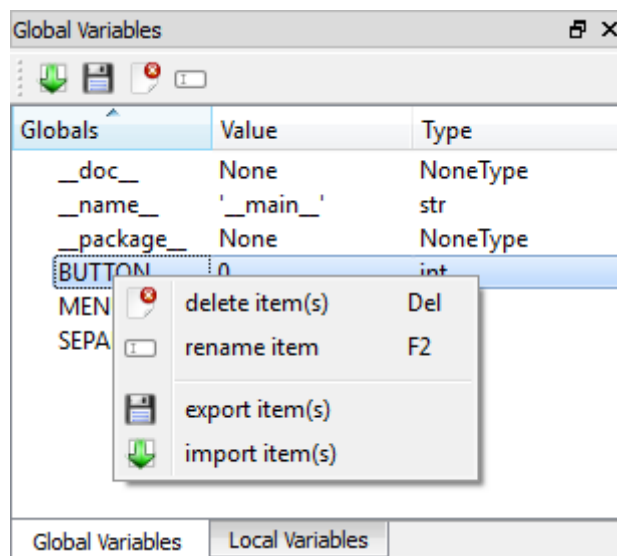
Struc- ture			
File	New Script	 Ctrl+N	Opens a new Python script in the <i>Editor</i>
	Open File	 Ctrl+O	Opens a saved Python script in the <i>Editor</i>
	Properties...		property dialog for important settings of itom
	User Management...		organizing the user dependent appearance of itom
	Loaded plugins...		opens dialog to see the load status of all plugins
	Exit		terminates and quits itom
View	Toolboxes		In the submenu you can toggle the visibility of all toolbars and -boxes
Script	stop	Shift+F10	Stop currently running python code
	continue	F6	Run till end or next break point
	Step	F11	Step into call or if not possible make a stepover
	Step over	F10	Step to next call in the current function
	Step out	Shift+F11	Go on with code till next method call outside current function
	Run ... debug		If toggled python methods triggered by user interfaces is executed in debug.
	Reload modules		Reload imported python packages (e.g. import numpy)
Help	Assistant	 F1	Opens this documentation
	About Qt		Information about Qt
	About Itom		Information about your current itom-Version
Tools	UI Designer		Opens the Qt Designer (see <i>Creating advanced dialogs and windows</i>)

All the actions mentioned in the table above are accessible by either the menu of **itom** or some of them are also available in some main toolbars that come with **itom**.

4.1.3 Toolboxes

Content:

Global and Local Workspace



In Python, every loaded package has a global workspace, where all global variables are saved. The main workspace of **itom** is a huge dictionary, containing all variables as well as method and function pointers and loaded packages and modules. The content of this global workspace is listed in the toolbox **Global Variables**. There is a filter, such that all variables of type **method**, **function** and **type** (classes...) are not listed in that overview. If you want to access items of this global workspace you can directly access them by their variable name, e.g.:

```
>>print(__name__)
'__main__' #answer
```

However, this is only possible if you work in the command line or a script that is not part of a separate module. In this case, you can access items of **itom**'s global workspace by importing the module `__main__`:

```
import __main__
print(__main__.__dict__["__name__"])
```

Note: In Python, every module or package has its own global workspace. The main global workspace depicted in the toolbox is related to the global workspace of **itom** that is always accessible by other modules using the dictionary `__dict__` of the `__main__`-module.

The toolbox **local variables** is only enabled if you are currently debugging a python script and the execution is stopped at a breakpoint or in a certain line. Then, the current workspace (local scope) of the method, where the debugger has been stopped, is shown. If the debugger currently processes any code not contained in a method, no local workspace is available.

Double-clicking on an item opens a small dialog where the content of the variable is printed. This is the same result as:

```
print(VARIABLENAME)
```

The context menu, depicted in the figure above shows some important methods of this toolbox. Most of them are also accessible by the toolboxes toolbar. You can always delete  one or multiple selected variables or you can rename  one selected variable. Additionally, there is the possibility to import  or export  variables to or from this workspace.

A click on **import**  opens a file dialog where you can choose a specific file to import. The following file formats are loadable:

- itom data collection (.idc). This is a pickled file, that may contain several python variables and is loaded using the module **pickle** of python.

- Matlab file (.mat). You can load matlab files in **itom** if you have the Python package **scipy** installed.
- Several itom algorithm-plugins contain methods that implement one of the following interfaces: **iReadDataObject**, **iReadPointCloud**, **iReadPolygonMesh**. Then the data formats that these methods can load are accessible in the dialog as well for loading.

Note: Existing variable names will be replaced unless the variable points to a class, method or function.

If you selected one or multiple variables and click on **export** , another file save dialog is shown where you can choose a specific filename in order to export the variables. You can export into the following file formats:

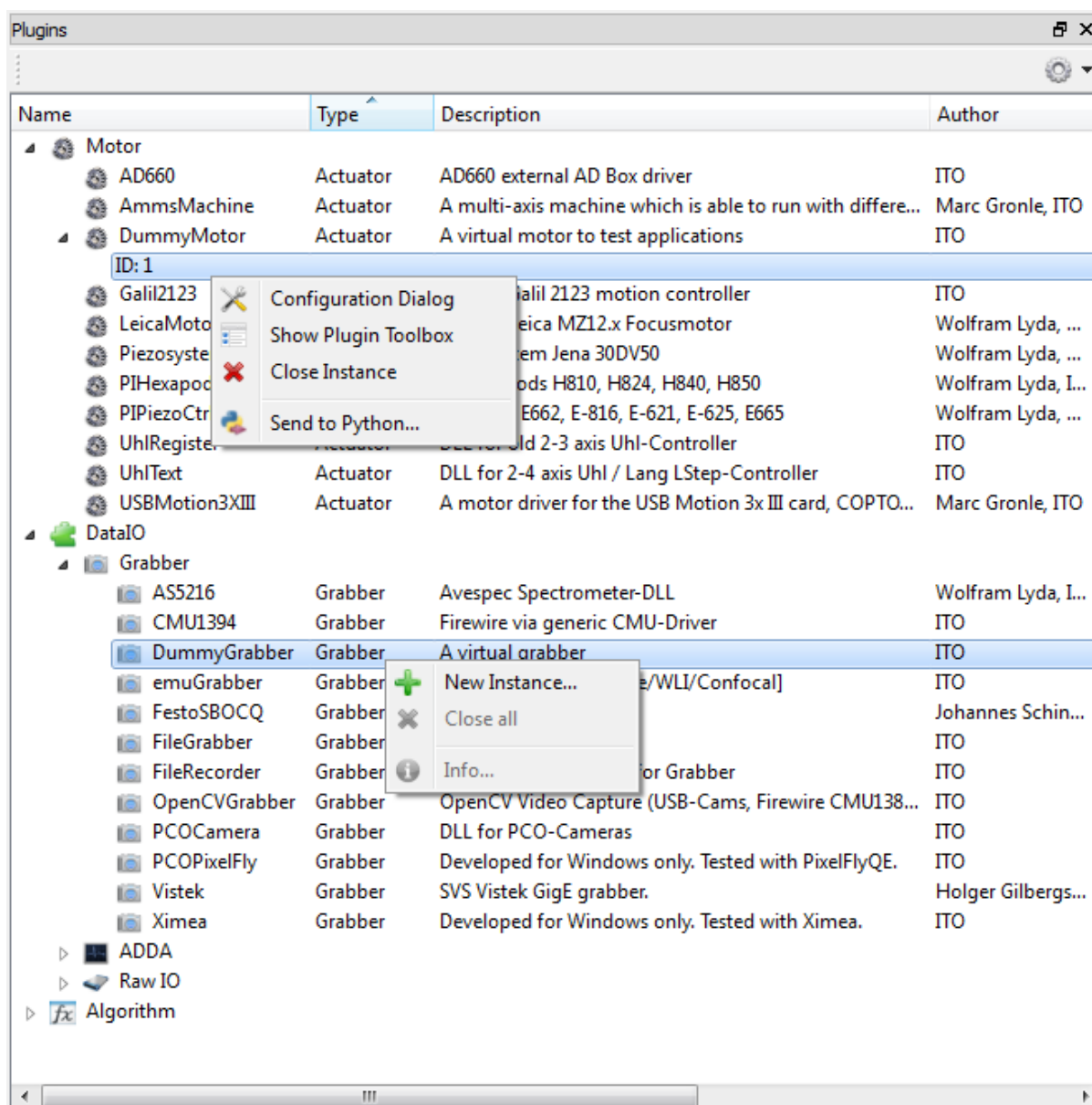
- itom data collection (.idc) [multiple files]. The variables are pickled (python module **pickle**) to the file.
- Matlab file (.mat). [multiple files, Python package **scipy** required].
- If you have itom algorithm-plugins installed that implement one of the following interfaces: **iWriteDataObject**, **iWritePointCloud**, **iWritePolygonMesh**, then you can also export dataObjects, pointClouds or polygonMeshes

into the specific file formats. [single variable only].

In both the file open and save dialogs, the filter list always considers all file formats that are available for import and export on your computer.

All variables which are defined in python console or in any python script are stored as global variables. These can be seen in the Global Variables Toolbox.

Plugins



The plugin toolbox shows all loaded plugins including any opened instances. The plugins are divided into the following types:

- Actuator (Motor) -> any actuator plugins.
- DataIO >> Grabber -> any cameras.
- DataIO >> ADDA -> any analog-to-digital or digital-to-analog converters.
- DataIO >> RawIO -> further input/output plugins.
- Algorithm -> all algorithm and extended-widget plugins.

The plugins of type **actuator** or **dataIO** behave in the way, that it is possible to open one or multiple instances of every plugin, while all algorithm plugins always provide a set of multiple algorithms (filters) or user interfaces.

At startup of **itom** all plugin libraries, located somewhere in the subfolder **plugins** of **itom** are loaded. If the load fails (for instance due to missing 3rd party libraries), their load is aborted and the reason can be seen in the dialog **loaded plugins...** (menu **file** >> **loaded plugins**). A right click to any hardware plugin opens the second context menu, depicted in the figure above. There you have the possibility to open a **new instance** of this plugin. After having created this instance, the plugin instance is unfolded and you see the instance by its identifier or ID like in it

is the case for the plugin **DummyMotor** in the figure above. The context menu of an instance is also depicted in the figure.

Depending on the implementation of the plugin you have the following possibilities:

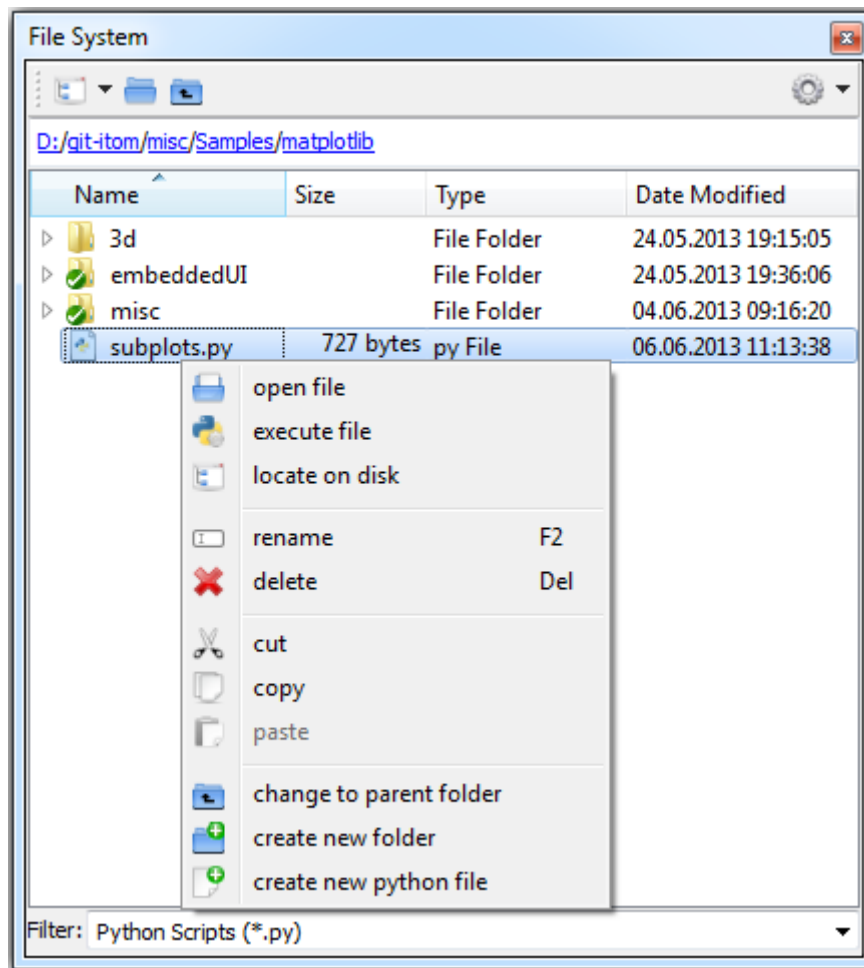
Name	Description
Configuration Dialog	If the plugin provides this a configuration dialog is started (else: disabled)
Show Plugin Toolbox	If provided, the instance's toolbox is shown in the main window (else: disabled)
Close Instance	closes this instance (see note below)
Send to Python...	You can access this plugin's instance by a global variable in Python under the name that you give in the input box that appears after clicking on this action.

Note: If you want to close an instance of a plugin by this toolbox, you need to consider the following remarks:

- If you opened the instance by the toolbox and no python variable is created for this plugin, you can simply close it by the GUI
 - If you created the instance by the GUI, however created a python variable, a click on **close instance** deletes the reference, the GUI is holding to the plugin's instance. However, the instance is only closed if all Python references are deleted as well.
 - If you created the instance by Python, you cannot delete it by the GUI.
-

File System

The file system toolbox gives you access to the file system of your harddrive.

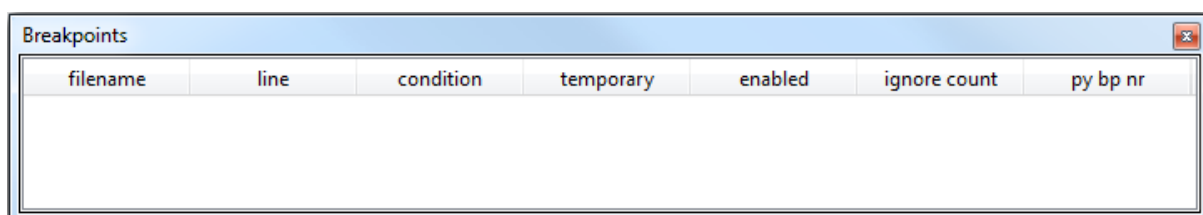


Below the toolbar, a breadcrumb menu allows to see the current path, where a click to any parent folder changes the current path to that folder. The current path is also printed at the right side of **itom**'s statusbar and also corresponds to the current, active directory of python. The filter below the file system, contains file filters to all file formats that are recognized by **itom**, either by a direct way (Python files, itom data collection, ...) or by algorithms implemented in any algorithm plugins. However you can also type your own filters in order to filter the file system tree.

The first combo box in the toolbar gives you access to the last ten current directories, that are also saved at termination of **itom** and are available at the next startup. The context menu of every item depends on the type of the item (file, folder, empty space in the tree). A right click on a python file for instance allows to open that file or to directly execute the script. The command **locate on disk** opens an external explorer with the current directory as start directory. Additionally, you can always create a new directory or python script in the current folder.

The file system toolbox also provides the common drag&drop functionalities allowing to move or copy files from and to another application, like the explorer. You can even drag a data file, that is known by **itom** onto a workspace toolbox in order to import that file in the workspace.

Breakpoints

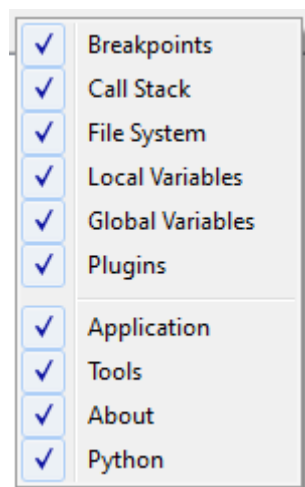


In the breakpoints Toolbox all breakpoints of current functions are listed. It gives an overview of all breakpoints even of functions which are not currently opened in the editor.

For a normal run of the iTOM software the breakpoint toolbox is not necessary, its only useful during debugging of some scripts or functions.

The core application of **itom** already comes with a set of different toolboxes. Additionally, many plugins provide the possibility to open a toolbox for every opened hardware instance, like actuators or cameras. These toolboxes are then also inserted into the main window of **itom**. Usually toolboxes can be docked into the main window or be in a floating state, such they appear like an unresizable window. If docked, they can be positioned at the left or right side or the top or bottom of the main window. Some of them however are limited with respect to the dockable positions.

All available toolboxes are listed in the menu **View >> Toolboxes**, where hidden toolboxes can be shown again. Additionally, a right click to any place in the toolbar opens the following context menu where the first items also access the loaded toolboxes. The items after the separator correspond to the toolbars, such that they can be hidden or shown:



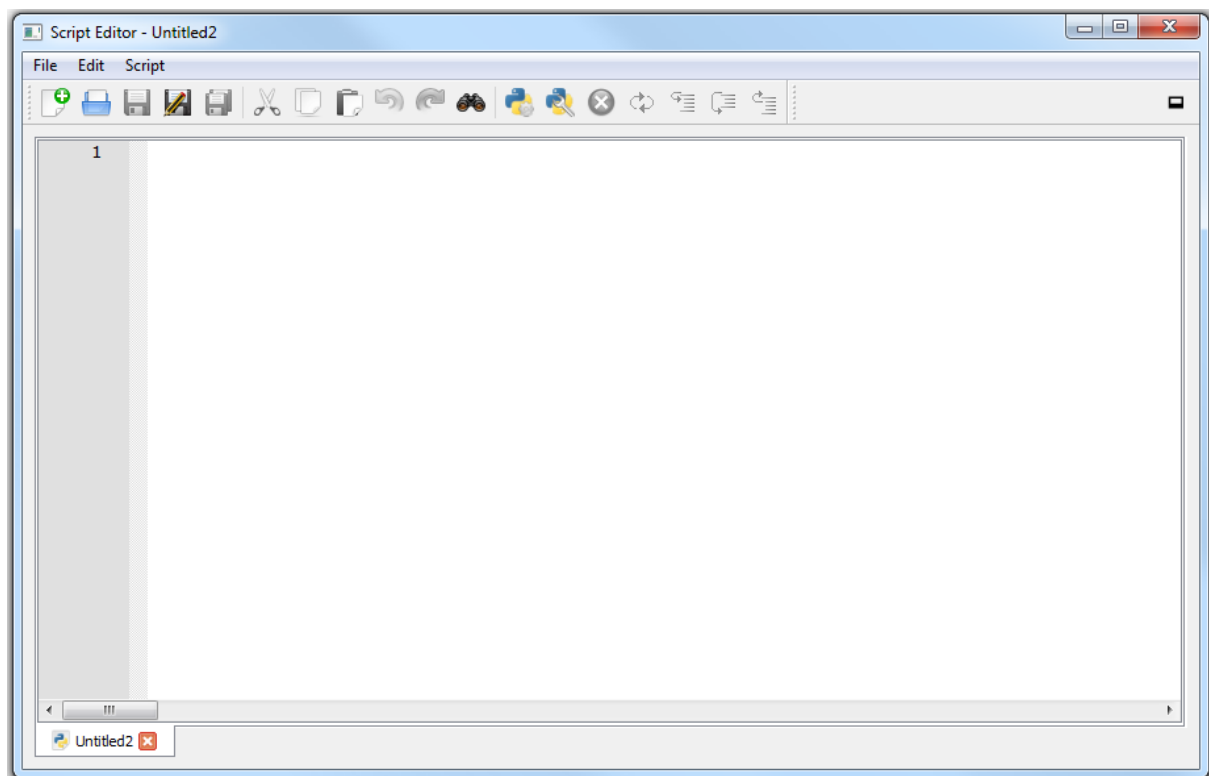
It is possible to (un)dock the Toolboxes to the main frame at different positions. This is done by simple drag and drop of the titel bar of the toolboxes. Another way of (un)docking can be realized by double-clicking on the title bar.

At the startup of the iTOM software all 5 Toolboxes are activated, which are:

The following main toolbars are available:

- *Global and Local Workspace* shows the global and local workspaces of Python.
- *Plugins* shows all loaded plugins including opened instances.
- *File System* gives you access to the file system of your harddrive.
- *Breakpoints* shows all breakpoints added to Python scripts.
- **callstack** shows the callstack when the Python script execution stops at a breakpoint.

4.2 Editor

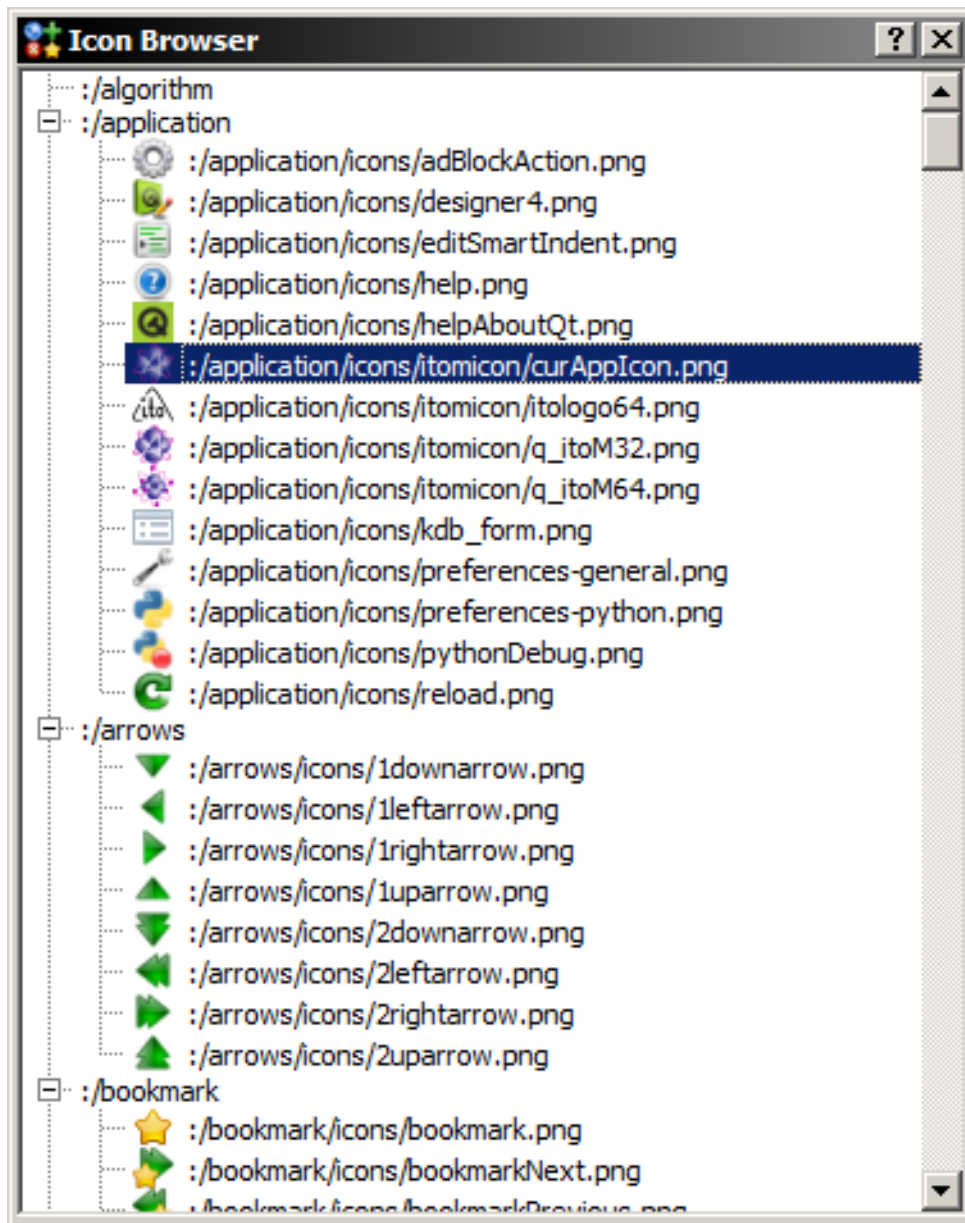


In the editor python scripts and methods can be written and executed. It offers debug-functions, syntax-highlighting and auto-completion.

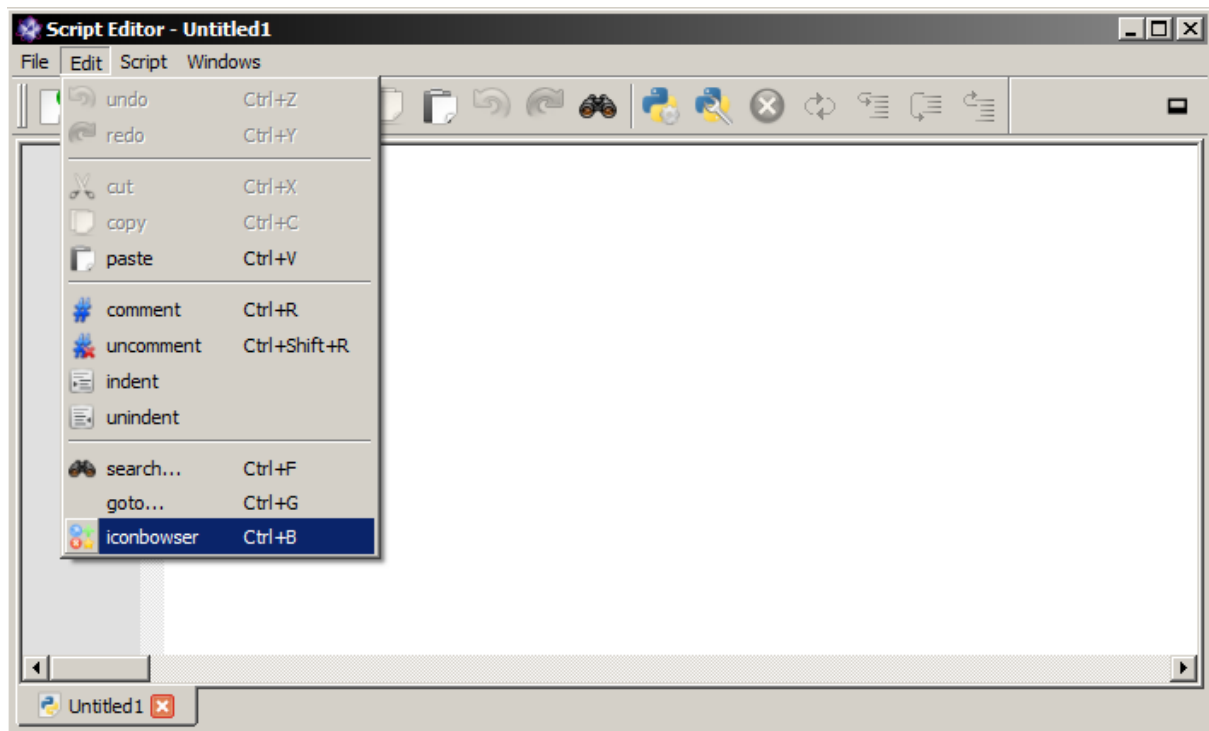
For debugging see `../script-language/debugging`

4.2.1 Icon-Browser

To help adding buttons and menu entries with icons a browser for internal icon-files is implemented.



By 'doubleClicking' a certain icon, the browser is automatically closed and the path of the selected icon is copied to the clipboard. The icon browser can be found under 'Edit' or open with the shortcut 'Ctrl+b'.



PLOTS AND FIGURES

itom contains functionalities to plot the mainly supported data structures like `dataObjects` or compatible numpy arrays. The visualization of more complex polygon mesh or point cloud structures is not part of the core application but can be provided by further plugins. Plots can be directly plot into their own figure window, they can be nested as subplots into one common figure and it is also possible to integrate plots into user-defined widgets and windows (see *Creating advanced dialogs and windows*).

Such that the integration into user interfaces, created at runtime by the external Qt Designer application, is feasible, every different type of plot is a plugin that implements the interface given by Qt Designer plugins. Therefore you can handle every itom plot as any other widget in Qt Designer, if this tool is called by itom.

Usually itom comes with a set of default plots, that can be used for the following plotting tasks:

- Static plot of a 1D data object (line plot).
- Static plot of a 2D data object, where the matrix values can be colorized by different color maps (default: gray map). For data objects with higher dimensions it is usually possible to change the currently depicted plane.
- Live plot of a line camera device (line plot).
- Live viewer for a camera providing two dimensional data objects (image plot).
- Three-dimensional plotting of 2D data objects where the data is interpreted as 2.5D, such that each value is considered to be a height value (isometric plot).

See the following files in order to learn more about plots and figures in itom:

Content:

5.1 Quick tutorial to plots and figures

5.1.1 Plots of data objects

The most common way to plot an arbitrary data object is the `plot()` command contained in the module `itom`.

In the first example, we create an one-dimensional data object with random values (16bit, signed, fixed point precision) and simple want to visualize this data object in a line plot. itom is able to recognize the type of plot you desire and uses the plot plugin which is set to be default for this type of plot (static, line plot). The defaults can be set in the property dialog of itom.

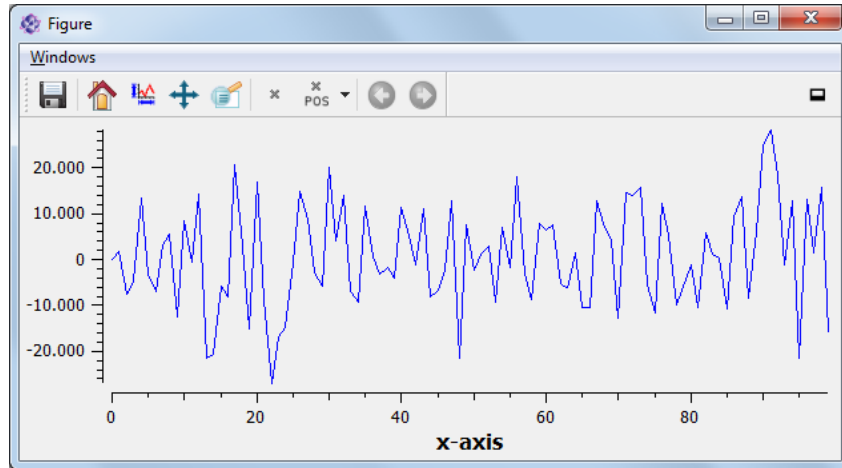
```
data1d = dataObject.randN([1,100], 'int16')
plot(data1d)
```

Note: Please consider that any one-dimensional data object is always exposed as two-dimensional data object, where the first (y) dimension is set to 1.

If you have various plot plugins available that can handle that type of data object, you can also force the plot command to use your specific plugin, which is defined by its class name (see itom's property dialog for the class name). If the class name cannot be found or is not able to plot the type of data object, itom falls back to the default plot plugin:

```
plot(dataId, "Itom1DQwtFigure")
```

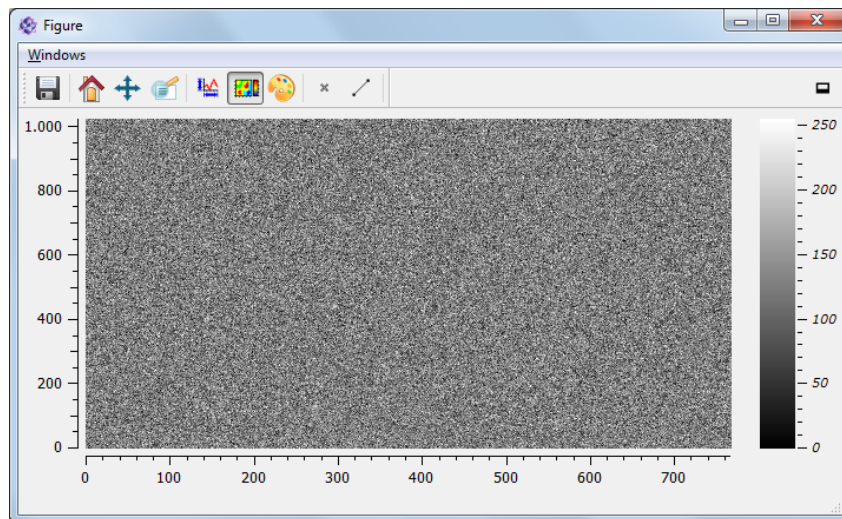
The result of both examples can be like this:



Equivalent to the one-dimensional case, the following example shows how to simply plot a two-dimensional data object also using the command `plot()`.

```
data2d = dataObject.randn([1024,768],'uint8')
plot(data2d)
```

Then, you obtain a figure that looks like this:



If you not only work with data objects but also with numpy

If the plot is opened in its own figure window, you have a dock-button in the toolbar on the right side. Click on this button in order to dock the plot into the main window of itom.

5.1.2 Live images of cameras and grabbers

itom is not only able to plot data objects but can also show live streams of connected and opened cameras. Cameras can be all plugins of type `dataIO` that also have the `grabber-type` flag defined (see the section grabbers of your

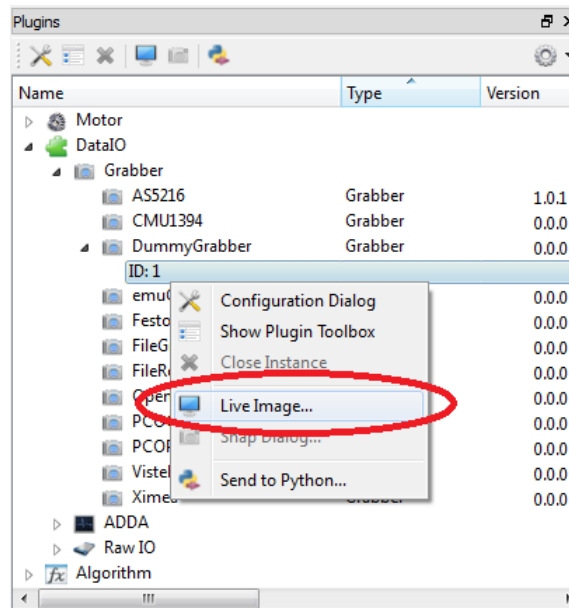
plugin toolbox in itom). If a live image of a specific camera should be created, the following process is started:

1. The camera is asked for its parameters `sizeX` and `sizeY`. If one of these dimensions is equal to one, a live line image is opened, else a two-dimensional live image is opened.
2. The command `startDevice()` of the camera is called (idle command if the camera is already started)
3. A timer continuously triggers the image acquisition of the camera and sends the result to all currently connected live images. However the timer is not started or stopped whenever the auto-grabbing property of the camera is disabled. This is useful, if you are in the middle of measurement process. Then you don't want the timer to force the image acquisition but your process. Therefore, disable to auto-grabbing property before starting your measurement and reset it to its previous status afterwards. In any case, whenever any process triggers an image acquisition, all results will always be sent to connected live images.
4. When the live plot is closed or disconnected, the command `:py:meth:itom.dataIO.stopDevice` is called (this is again an idle command if the camera is still used by other live images or has been started by any python script and not stopped yet).

In the following example, the dummy grabber camera is started and the live image is opened using the command `liveImage()`. The auto-grabbing property is set to `True` (which is also the default case):

```
cam = dataIO("DummyGrabber")
cam.setAutoGrabbing(True)
liveImage(cam)
```

You can also show the live image of any camera using the GUI. Right-click on the opened camera instance in the plugin toolbox and choose **live image**:



5.1.3 Changing properties of plots

Any plots have properties defined, which indicate the appearance or currently depicted data object or camera. To access these cameras you need to get the instance of the plot or live image item. This is always an instance of the class `plotItem`. This class is inherited by `uiItem` which finally provides the access to the properties by the functionalities described in [Creating advanced dialogs and windows](#).

In order to access the necessary instance of `plotItem`, you will see that the return value of the commands `plot()` or `liveImage()` is a tuple consisting of a number of the overall figure (window), where the plot is print and of the requested instance as second value.

In the next example, the title of a two-dimensional data object plot is changed:

```
data2d = dataObject.randN([100,100])
[idx,h] = plot(data2d)
h["title"] = "new title"
```

Note: Not all plot plugins have the same properties defined, since this also depends on their type and special features. However it is intended to use the same property names for the same meaning in the different plugins.

Note: If the figure closed while you still have a reference to its instance, any method of this instance will raise an error saying that the plot does not exist any more.

In order to get a list of all properties of a plot, call the method `info()` of the plot instance. This method prints a list of available properties as well as slots and signals.

```
h.info()
```

There are two other important properties that let you change the displayed data object or camera:

```
#set new data object
h["source"] = dataObject.randN([100,100])

#assign new camera
h["camera"] = dataIO("DummyGrabber")
```

These properties are also the way to set the content of plot widgets, that are integrated in your user-defined GUIs.

5.2 Figure Management

Technically, every plot widget, hence the plot containing the axes and the content, is represented as an instance of `plotItem`. If you show the plot in its own window, this window is called figure and represented as an instance of `figure`. Every figure is also able to show various sub-plots arranged in a regular grid of m rows and n columns. Plots, integrated in an user-defined interface, are directly integrated there without being part of a special figure.

If you want to continuously use the same figure window for sequentially plotting different data or if you want to use the sub-plot mechanism, it is useful to firstly create or access a specific figure and then plot data objects or camera live images using the specific commands of this figure.

Every figure has a specific handle (integer value). Therefore you can have access to any figure by its figure instance or via the handle value.

In the following example, a new figure is created and opened. The instance is saved in the variable *fig*:

```
fig = figure()
```

Assuming, that you want to access an already existing figure or create a new figure with your desired handle number, give the handle as first argument (keyword: *handle*):

```
fig = figure(1)
#or
fig = figure(handle = 1)
```

fig is then an instance of `figure`. This class is derived from `uiItem` and therefore has the same functionality like any other widget that is represented by `uiItem` (see *Creating advanced dialogs and windows*). If you provide no further parameter (besides *handle*) to the figure constructor, a figure with one plot area is created. If you want to create a grid of more plot areas, you need to give the number of rows and columns as argument to the constructor:

```
fig = figure(rows = 2, cols = 3)
```

Not only the module `itom` has the methods `liveImage()` in order to plot data objects or camera streams. In the same way, the class `figure` has these methods, too:

With method signature:

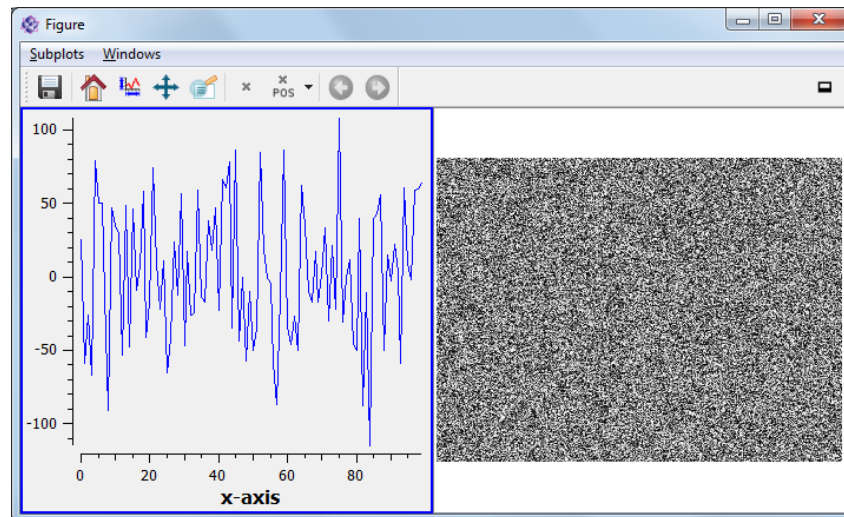
<code>itom.figure.plot((data, [areaIndex, ...])</code>	Plot an existing dataObject in not dockable, not blocking window.
<code>itom.figure.liveImage((cam, [areaIndex, ...])</code>	Creates a plot-image (2D) and automatically grabs images into this window.

Both have the optional keyword parameter *areaIndex*, such that you define in which sub-plotting area the requested plot or live image should be depicted. If you don't provide the *areaIndex* parameter, the first area (top, left) is assumed (this is also the default in case that you don't use subplots). The *areaIndex* value is a fixed-point number beginning with 0 for the top-left area of the figure. Then, the index iterates row-wise through the entire grid of sub-plots.

In the following example, a 2x1 grid is created where the left plot shows a 1D data object and the right plot contains the live stream of the dummy grabber camera instance:

```
fig = figure(rows = 1, cols = 2)
dataId = dataObject.randN([1,100])
fig.plot(dataId, 0)
cam = dataIO("DummyGrabber")
fig.liveImage(cam, 1)
```

The result looks like this:



The currently depicted toolbar is provided by the active subplot, marked by a border. You can switch the active subplot by choosing your desired one in the figure's menu *subplot*. In order to get the instance of `itom.plotItem` of any subplot, use the following command:

```
plotLeft = fig.subplot(0)
plotRight = fig.subplot(1)
```

Note: If you have an instance to a `plotItem` only in python and the window is closed, the window is finally deleted and the `plotItem` becomes invalid. In difference, a handle to a figure keeps the figure alive until all references to the figure's instance are deleted. Therefore you can always show or hide a figure using its instance and the methods `show()` or `hide()`.

The `figure` also provides a static method `close()` to close and delete a figure defined by its specific integer handle or to close all figures using the string 'all' as parameter. Please note, that only these figures are finally deleted where no other python references exist to them.

```
#close figure with handle 7
figure.close(7)
```

```
#close all figures  
figure.close('all')
```

5.3 line plots (1D)

“itom1DQWTFigure” is the basic line plot for plotting DataObjects with 1xN or Mx1 data-Slices based on QWT. All DataTypes are accepted. To plot complex objects, it is possible to select between the following modes: “absolut”, “phase”, “real” and “imaginary”.

5.4 2D image plots

“itom2DQWTFigure” and “itom2DGVFigure” are the basic plots for visualization of image like DataObjects. Both plots has line-picker and point pickers included. By pressing “Ctrl” during picker movement, the picker can only be moved horizontal or vertical according to the mouse movement.

“itom2DQWTFigure” is designs as a visualization of metrical data, false color or topography measurements. It supports the axis-scaling / axis offset of DataObjects, offers axis-tags and meta-data handling. It does not offer plotting of real color images. All DataTypes are accepted. To plot complex objects, it is possible to select between the following modes: “absolut”, “phase”, “real” and “imaginary”. The data is plotted mathematically correct. This means the value at [0,0] is in the lower left position.

“itom2DQWTFigure” is designs as a fast visualization of images, e.g. direct grabber output or colored images. It allows the plotting of real colors (at the moment only 24-bit or 32-bit). It does not handle meta-data. All DataTypes are accepted. To plot complex objects, it is possible to select between the following modes: “absolut”, “phase”, “real” and “imaginary”. The data is plotted image orientated. This means the value at [0,0] is in the upper left position.

5.5 isometric Plot

“itom1DQWTFigure” is a plot for pseudo 3D visualization of image like DataObjects. It is based on OpenGL and renders the objects either to triangles (“triangle mode”) or points (“Joe-Mode”). All DataTypes are accepted. To plot complex objects, it is possible to select between the following modes: “absolut”, “phase”, “real” and “imaginary”.

The line plot offers two point pickers, addressed by “Key 1” and “Key 2”. Picker_2 is allways on the right side of Picker_1. The are moved by “Key Left” and “Key Right” or by pressing the left mouse key.

EXTENDING THE USER INTERFACE OF ITOM

6.1 Customize the menu and toolbars of itom

In this section, it is shown how you can add your user-defined toolbars and menus to the main window of **itom**. Clicks to these execute arbitrary python code or methods. The creation of the toolbars, buttons and menus is done using python code, too.

6.1.1 Add toolbars and buttons

Using the embedded scripting language in **itom**, you can add your own toolbars and buttons in order to automatically execute specific **Python**-commands or -methods. Every button that is created is related to a toolbar defined by its toolbar-name. If a toolbar-name does not already exist, a new toolbar with this name is created and the button is added to this toolbar. Every toolbar can be arbitrarily positioned in **itom** or undocked to any floating position on your screen.

A single button in a toolbar is created using the command `addButton()`, vice-versa the button is removed with `removeButton()`. The syntax for both is:

```
addButton(toolbarName, buttonName, code [, icon, argtuple])
removeButton(toolbarName, buttonName)
```

Your button is accessed by its name **buttonName**, which also is printed on the button if no icon is defined. The toolbar, where the button is displayed, is defined by its name **toolbarName**. You can either provide a python code snippet as string or a reference to any method or function. If this method requires parameters, you can add these parameters as tuple to the keyword-parameter *argtuple*. Additionally, it is possible to assign an icon to the button. Therefore you give the absolute path of the icon to the parameter *icon*. See the [icon section](#) below about the different possibilities how to assign a valid filename.

In the following example, three different buttons linking to both python code and some functions are created:

```
def test1():
    print("The button test1 has been clicked")

def test2(a,b):
    print("The result of a+b is",a+b)

addButton("myToolbar","HelloWorld","print('Hello World')")
addButton("myToolbar","BtnTest1",test1, ":\application\icons\itomicon\curAppIcon.png")
addButton("myToolbar","BtnTest2",test2, argtuple=(4,7))
```

Note: In this example, it is assumed that the module **itom** has been globally and totally imported by:

```
from itom import *
```

If this is not the case, you can also import `itom` and call `itom.addButton(...)`. Additionally, if you create a button in a toolbar, whose name already exists, the previous button will be deleted first. If you assign a code-snippet to the `code`-parameter that includes quotation marks, make sure that the quotation mark around the code-snippet is different to the quotation marks within the code snippet or escape your inner quotation marks by a backslash.

```
addButton("myToolbar", "QuotExample", "print('different quotation mark')")
addButton("myToolbar", "QuotExample", "print(\"escaped quotation mark\")")
```

All these buttons are removed by the following lines of code:

```
removeButton("myToolbar", "HelloWorld")
removeButton("myToolbar", "BtnTest1")
removeButton("myToolbar", "BtnTest2")
```

Note: If the last button of a toolbar has been removed, the toolbar is removed as well.

6.1.2 Create menus

You can not only add buttons to the toolbar of **itom**'s main window but also create your menu and sub-menu structure. Therefore the commands `removeMenu()` are available.

There are three types of menu items, that can be created:

- **MENU** (`itom.MENU` [2]) creates a menu-item, having any possible sub-item. This item cannot be connected to any code. Every menu always starts with an item of this type.
- **BUTTON** (`itom.BUTTON` [0]) creates a real menu-item, that is the mode of the menu-tree. Only a click to these items can execute code.
- **SEPARATOR** (`itom.SEPARATOR` [1]) creates a separator-item in the menu or submenu. It is also not connected to any code.

Any menu-item is defined by its key. The key is a slash-separated list, where the single items stand for the path one has to walk through the menu-tree in order to access the desired item. If an item with a complex-tree structure is created where some of the parent-nodes do not already exist, they are iteratively created (type **MENU**) in order to finally create the desired node-element.

The call to `addMenu()` is as follows:

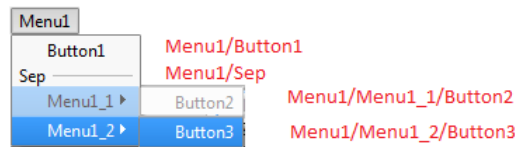
```
addMenu(type, key [,name, code, icon, argtuple])
```

Hereby *type* is the item's type like stated above and *key* denotes the absolute key to the item, which also indicates the tree-structure, where the item should be added. The name of your item is given by *name*, while *code*, *icon* and *argtuple* have the same meaning like in the case of adding a button to the toolbar (*see above*).

Here are some examples for creating a menu:

```
1 def btn2(arg=None):
2     print("button 2 clicked:",arg)
3
4 addMenu(itom.BUTTON, "Menu1/Button1", "Button1", "print('button1 pressed')")
5 addMenu(itom.SEPARATOR, "Menu1/Sep")
6 addMenu(itom.MENU, "Menu1/Menu1_1")
7 addMenu(itom.BUTTON, "Menu1/Menu1_1/Button2", "Button2", btn2)
8 addMenu(itom.BUTTON, "Menu1/Menu1_2/Button3", "Button3", btn2, argtuple=[2])
```

In line 4, the *Button1* is created. Since its parent node *Menu1* does not exist, it will be created (type **MENU**). Next, in line 5 a separator is added as subitem of *Menu1*. Hence, it is appended to *Button1*. Afterwards two sub-nodes *Menu1_1* and *Menu1_2* are added which both have a children respectively, called *Button2* and *Button3*.



Note: Please consider, that in line 8 an *argtuple* is appended to the function-call to *btn2*. Although only one argument is passed it must be included in a tuple. Usually a tuple is created by the bace operator (2). However, since only one argument is given, python is interpreting this brace-operator as mathematical expression and it is reduced to 2 only. Therefore, we use the square bracket in order to create a list, that is implicitly converted to a tuple.

In order to remove any menu item including its subitem, call `removeMenu()` with the key-word of the specific item. For instance, the removal of all menus created above, is done by:

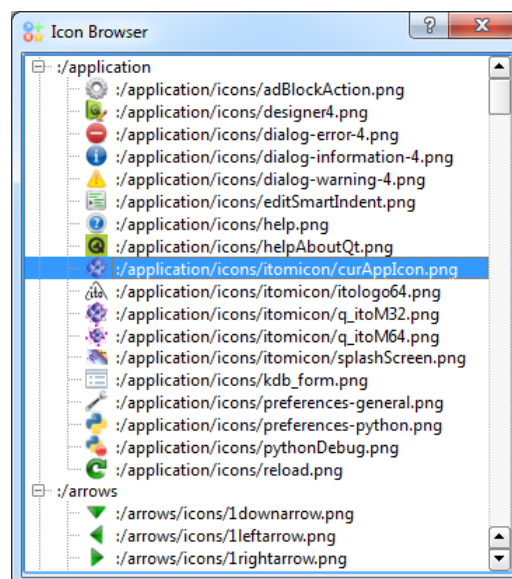
```
removeMenu("Menu1")
```

The only argument of the command `removeMenu()` is only the key of the menu-item to delete.

6.1.3 Icons in user-defined toolbars and menus

Both for toolbar-buttons as well for menu-entries you can assign an arbitrary icon. Usually it is recommended to have an icon file with a size of 24x24 Px or below (will be automatically resized) in any image format (*png* recommended, available is *bmp*, *gif*, *jpg*, *tiff*...). The argument **icon** of the commands `addButton()` and `addMenu()` must be a string with the absolute or relative filename of the icon-file. The relative filename is always considered to be relative with respect to the current working directory, as it is printed at the right bottom side of **itom**'s main window (or use the command `getCurrentPath()`). Besides that, you can also pass an absolute path to your icon-file. The command `getAppPath()` returns the absolute path of the **itom**-application. Additionally you can use methods from the **Python**-module `os.path` in order to create valid absolute paths.

Besides assigning an external icon-file to the **icon** parameter, **itom** also gives you access to any icons that are compiled as resources within the **itom** application. All icons, that are included in these resources are listed in the **icon browser**:



The icon browser is accessible in any script window by its menu **edit >> icon browser** (or Ctrl+B). If you found your desired icon, double click on the entry in order to copy the appropriate string to the clipboard and paste it afterwards into your script. Resource locations always start with a colon (:) sign.

6.2 Show messages, input boxes and default dialogs

itom gives you the possibility to show the standard message and input boxes as well as standard dialogs per **Python** script command.

6.2.1 Message boxes

There are three types of message boxes that you can show via a script command. These are divided into the four types **Information**, **Warning**, **Question** and **Critical**.

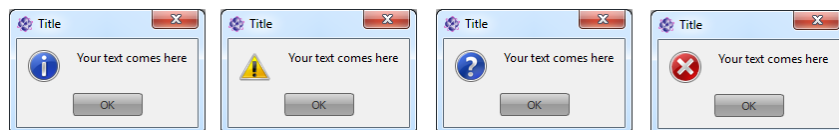


Figure 6.1: Appearance of the different message boxes on a Windows 7 operating system. From left to right: *Information*, *Warning*, *Question*, *Critical*

The corresponding commands are the following static methods in the class `ui`:

- `msgInformation()`
- `msgWarning()`
- `msgQuestion()`
- `msgCritical()`

All these methods have the same syntax:

```
(button,buttonText) = msgInformation(title, text [,buttons, defaultButton, parent])
```

The parameters *title* and *text* are the strings for the titlebar and the content area of the message box respectively. The other three parameters are optional: *buttons* is an or-combination of button-IDs, denoting the different buttons you want to display at the bottom of the message box. The possible values correspond to the enumeration **QMessageBox::StandardButton** of the *Qt*-framework and are:

Enumeration	Value
<code>ui.MsgBoxOk</code>	1024
<code>ui.MsgBoxOpen</code>	8192
<code>ui.MsgBoxSave</code>	2048
<code>ui.MsgBoxCancel</code>	4194304
<code>ui.MsgBoxClose</code>	2097152
<code>ui.MsgBoxDiscard</code>	8388608
<code>ui.MsgBoxApply</code>	33554432
<code>ui.MsgBoxReset</code>	67108864
<code>ui.MsgBoxRestoreDefaults</code>	134217728
<code>ui.MsgBoxHelp</code>	16777216
<code>ui.MsgBoxSaveAll</code>	4096
<code>ui.MsgBoxYes</code>	16384
<code>ui.MsgBoxYesToAll</code>	32768
<code>ui.MsgBoxNo</code>	65536
<code>ui.MsgBoxNoToAll</code>	131072
<code>ui.MsgBoxAbort</code>	262144
<code>ui.MsgBoxRetry</code>	524288
<code>ui.MsgBoxIgnore</code>	1048576
<code>ui.MsgBoxNoButton</code>	0

For an **or**-combination, use the bar-operator (`|`). The parameter *defaultButton* only accepts one button value of buttons, listed in the parameter *button*. The specific button then becomes the default button that is selectable by

simply pressing the return-key. Additionally, you can let the message box be a child of any other instance of class `:py:class`~itom.ui``. This is useful, if you want to make the message box modal with respect to any *user defined dialog* and not with respect to the main window of **itom**.

The return value is a tuple containing two values. The first is the value of the button that has been pressed to close the message box with respect to the enumeration values stated above. The second value is the text of the pressed button (e.g. "Ok").

An example for a question message box is:

```
(btn, btnText) = ui.msgQuestion("Silly question", "Do you like itom?", \
    ui.MsgBoxYes | ui.MsgBoxNo, ui.MsgBoxYes)
print("The user pressed the button", btnText, "with ID", btn)
if(btn == ui.MsgBoxNo):
    print("what can we do in order to change your opinion?")
```

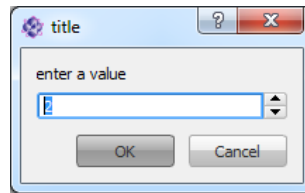
6.2.2 Input boxes

The class `:py:class`~itom.ui`` also contains another group of static methods, that can be used in order to show modal input boxes. These are:

Integer input box

Use `getInt()` in order to ask the user for a fixed-point number:

```
(value, accepted) = ui.getInt(title, label, defaultValue, min, max, step=1)
```

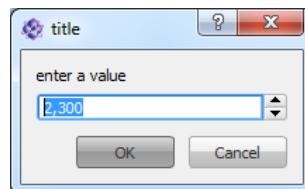


The minimum and maximum value of the input box is set to the values *min* and *max* or to the full integer range, if not otherwise stated. The step-size *step* defines the iteration value when the arrows are clicked. The return value is a tuple that contains the current value of the input box and **True** if the dialog has been accepted, otherwise **False**, the value still contains the last value of the spinbox.

Double input box

Use `getDouble()` in order to ask the user for a double-precision floating-point number:

```
(value, accepted) = ui.getDouble(title, label, defaultValue, min, max, decimals=1)
```

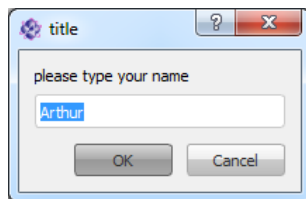


The minimum and maximum value of the input box is set to the values *min* and *max* or to the full double range, if not otherwise stated. In this case you can assign the number of decimals that are shown (default: 1). The step size is always set to 1. The return tuple has the same form than for the integer input box.

Text input box

Use `getText()` in order to ask the user for a string:

```
(value, accepted) = ui.getText(title, label, defaultValue)
```

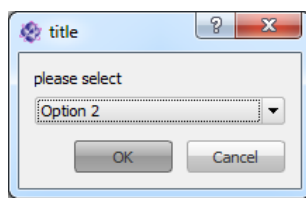


The default text can be passed as parameter *defaultValue*. The return tuple also returns the current value of the textbox and the boolean variable **True** if the dialog has been accepted, else **False**.

Option selection dialog

Use `getItem()` in order to let the user select an item from a given list of items:

```
(value, accepted) = ui.getItem(title, label, stringList [,currentIndex=0, editable=False])
```



Use the parameter *stringList* in order to pass any sequence (tuple or list) of strings that are displayed as options, the user can choose. You can preselect any item by setting the parameter *currentIndex* to the item's index (zero-based, default:0). Additionally it is possible to keep the selection dialog editable, hence, the user can also type its own string (if *True*, default: *False*).

The returning tuple again contains the current active or typed string value and the boolean variable, that is only **True** if the dialog has been accepted.

6.2.3 Standard dialogs

itom provides access to a standard dialog to indicate an existing directory, to choose an existing file or to indicate a path for saving a file.

Indicate existing directory

The dialog is wrapped by the static method `getExistingDirectory()`:

```
ret = ui.getExistingDirectory(caption, startDirectory [, options = 1, parent])
```

The parameters are as follows:

- **caption** is the title of the dialog
- **startDirectory** is a string containing an absolute path to the default directory
- **options** is an optional or-combination of one of the following options:
 1. ShowDirsOnly [default], only directories are shown in the dialog
 2. DontResolveSymlinks, if indicated symbolic links are not shown
 other values can be taken from the Qt-enumeration **QFileDialog::Option**

- **parent** can be an instance of `ui`, such that the dialog becomes modal and on top of this user defined dialog

The return value **ret** of this dialog is either **None** if the dialog has been rejected or else the absolute path to the chosen directory.

OpenFileName-Dialog

Use the static method `getOpenFileName()` to show this dialog. By this dialog, the user can select one specific existing file in any directory. This is usually taken for opening files.

```
ret = ui.getOpenFileName([caption="", startDirectory="", filters="", \
    selectedFilterIndex=0, options=0, parent])
```

The title of the dialog can be assigned by the parameter *caption*. You can choose a default directory by the string *startDirectory*. If this is not given or empty, the current working directory is taken. The possible file filters are set by the string **filters**. This is a double-semicolon separated list of entries. An example is:

```
Images (*.png *jpg)
Text Files (*.txt);;Itom Data Collection (*.idc);;Scripts (*.py *pyc)
```

The selected, default index is given by *selectedFilterIndex*, where 0 selects the first item in *filters*. You can also pass some additional options to the dialog by the parameter *options*. This basically is an or-combination of the Qt-enumeration `QFileDialog::Option`. Reasonable values are:

- 0x02, `DontResolveSymlinks`. If given, symbolic links are not shown in the dialog
- 0x40, `HideNameFilterDetails`. This only shows the names of the filters, but not the pattern mask.

The return value *ret* is *None*, if the dialog has been rejected, else it is the absolute filename of the selected file.

SaveFileName-Dialog

Use the static method `getSaveFileName()` to show a dialog, where the user can select a new filename or an existing file in any directory. This is usually taken for saving files.

```
ret = ui.getSaveFileName([caption, startDirectory="", filters="", \
    selectedFilterIndex=0, options=0, parent])
```

The parameters are the same than for the open-file-dialog above. However for options, reasonable values are now:

- 0x02, `DontResolveSymlinks`. If given, symbolic links are not shown in the dialog
- 0x04, `DontConfirmOverwrite`. If given, the user don't has to confirm if an existing file has been choosen.
- 0x40, `HideNameFilterDetails`. This only shows the names of the filters, but not the pattern mask.

For combining different options, use the or-operator `|`.

6.3 Creating advanced dialogs and windows

With **itom** it is not only possible to add menus and toolbar elements to the main GUI of **itom** or to use the default set of input and message boxes, but it is also possible to create own user interfaces. These interfaces are designed by help of a WYSIWYG ("what you see is what you get") design tool (Qt Designer). The logic behind the surfaces is then scripted using **Python**. Therefore it is possible to change the appearance of control elements at runtime or to connect a signal, emitted when for instance clicking on a button, with a user-defined python method.

In this chapter, the creation of such user interfaces is explained.

6.3.1 Qt Designer

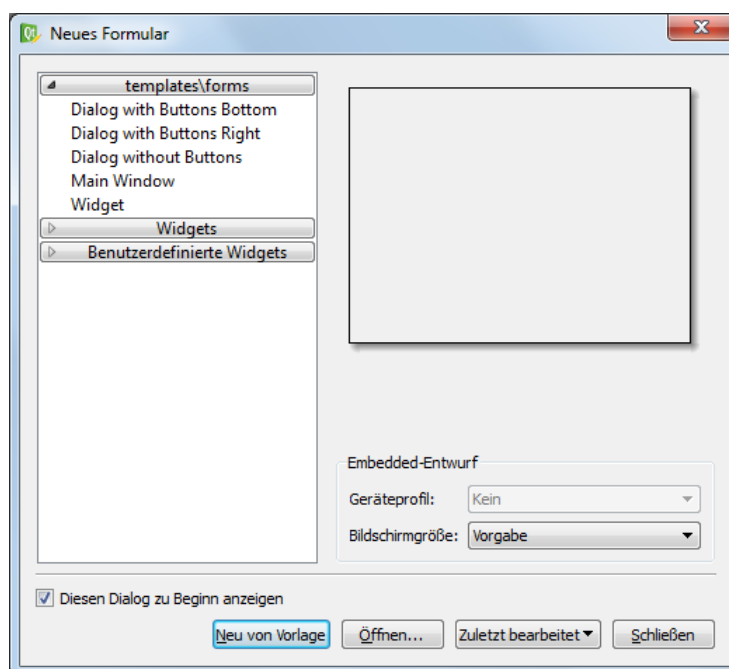
The Qt Designer can be used to create a GUI for interaction with the **itom** software.

For details see the Qt Designer documentation under <http://qt-project.org/doc/qt-4.8/designer-manual.html>

In order to start the **Qt Designer**, click on the corresponding icon in the toolbar of **itom**:



or double-click on a corresponding **ui**-file in the file system widget of **itom**. In the first case, **Qt Designer** shows an initialization dialog, where you can choose the base type of the user interface you want to create.

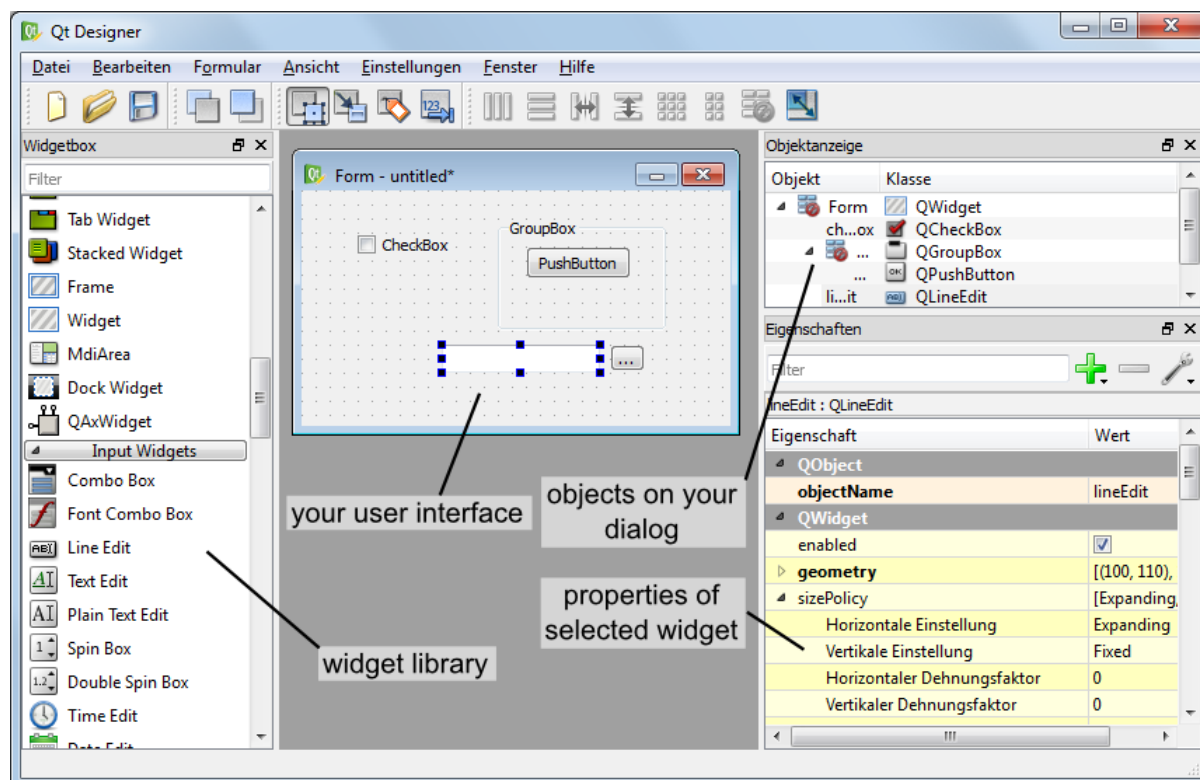


In principle you have the possibility to choose between three different base layouts:

1. **Dialog**. A Dialog is usually displayed on top of the main window and only has got one close-button in its title bar. Often, dialogs are used for configuration dialogs where the user finally closes the dialog using one of the standard buttons (OK, Cancel, Apply...) in order to confirm or reject the current changes in the dialog. A dialog cannot have its own toolbar, menu or status bar.
2. **Main Window**. A main window is a fully equipped main window, which can be minimized, maximized, can have toolbars, menus and a status bar. Therefore it is recommended to use this type of user interface for the main window of your measurement system. Like a dialog, it is possible to show the main window on top of **itom** (as sub-window of **itom**) or as independent window, which has its own icon in the windows tray.
3. **Widget**. A widget is the base class for all control elements provided by **Qt**. Therefore a widget does not have any title bar or windows frame. Nevertheless you can choose a widget for your user interface, since **itom** provides the possibility to stack this widget into a default dialog which can optionally show some default buttons on the right side or at the bottom of the dialog. This is the easiest way the generate a configuration dialog in **itom**, since you do not need to script the necessary methods handling clicks on one of these buttons. In this case, **itom** automatically gets full information about the close status and type of closing of the dialog (accepted, rejected...).

After having chosen one of these base layouts (types), your surface is displayed in the middle of the **Qt Designer** and you can start to drag elements from the widget library on your surface. If the **Qt Designer** is started from **itom** you will even find a section **ITOM Plugins** in the library list, which contains all loadable designer plugins that

are provided by **itom** and can also be placed on your surface. The choice of these plugins depend on the designer plugins that are currently available in your installation of **itom**.



After having placed one widget on the canvas, you will see its properties in the property toolbox of **Qt Designer**. Every widget has the common property **objectName**. If you assign a unique object name to any of your control elements, it is possible to access and manipulate this widget from a **Python** script in **itom** using this name, too. In general many of the properties that are visible in the property toolbox can afterwards be read or changed by an appropriate script (depending on the data type of the property).

The alignment of control elements on the surface is mainly controlled by so-called layout elements. These layouts together with size policies that can be assigned to every widget control the appearance of the entire user interface and provide the feature that the dialog can be changed in size whereas all widgets are dynamically repositioned. For more information about laying out your user interface, see <http://qt-project.org/doc/qt-4.8/designer-layouts.html>.

Finally, save your user interface under a convenient filename with the suffix **.ui**.

Widget Library

In principle, you are allowed to place every widget on your user interface that is available in the widget library (widget box) of **Qt Designer**. Later, you will learn how you can access properties of any widget (read and/or write) and how you can call specific functions provided by any widget. However, you will also learn that you do not have access using **Python** to all functions a widget has and you are not able to sub-class any widget, like you can it using a native **C++** program. Therefore, it is not recommended to place any widget from the group **Item Views (Model-based)** on your user interface since only few functions of these widgets are accessible by a **Python** script. If you need a list box, use the item-based list widget. **itom** also provides some widgets (section **ITOM widgets**) that can be placed on your user interfaces, for instance some plot widgets or the widget for plotting the result of the python module *matplotlib* (see *Python-Module matplotlib*).

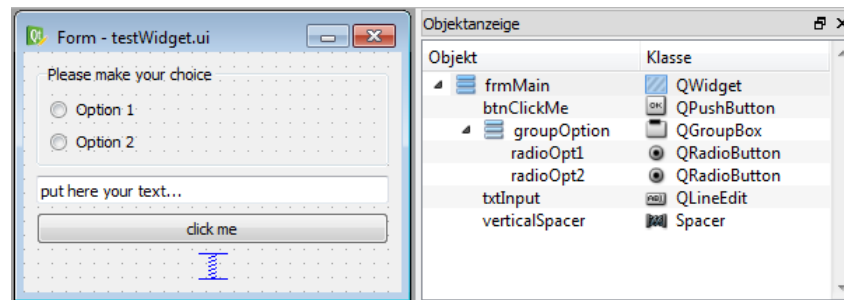
6.3.2 Loading user interface in itom

In this section, an introduction is given how to create and load user interfaces in **itom** depending on different type-attributes.

Widget embedded in itom-dialog (TYPEDIALOG)

Like described above, the easiest and most comfortable way to load user interfaces in **itom** is to use the type **TYPEDIALOG**. In **Qt Creator** you design a widget with your individual content and then when loading this GUI in **itom**, the widget is embedded in a dialog provided by **itom**, which optionally adds a horizontal or vertical button bar at the right side or at the bottom of the dialog.

Let us create an exemplary user interface. In **Qt Creator** the following widget has been created:



On the right side of the widget *testWidget* you see the hierarchical organization of objects that are put on the widget. At first, a group box has been placed on the widget. Inside of this group box two radio buttons have been placed using a simple drag&drop from the widget library. Both radio buttons are aligned inside of the group box with a vertical layout. This is reached by a right-click on the group box and choosing *vertical layout* from the *layout* menu. Below the group box, a widget of type *lineEdit* and a push button (type *pushbutton*) have been placed. Finally the three main elements are also aligned in a vertical layout with respect to the overall widget. This can be achieved by a right click on an empty space of the widget or directly in the *object inspector*. If you increase now the size of the overall widget, you will see that all sub-elements are resized according to their layout. Since we don't want sub-widgets to be vertically stretched and distributed, a vertical spacer element has been placed at the bottom of the vertical layout stack.

The following properties have been directly set in **Qt Creator**:

- group box: *objectName*: groupOption, *title*: 'Please make your choice'
- push button: *objectName*: btnClickMe, *text*: 'click me'
- line edit: *objectName*: txtInput, *text*: 'put here your text...'
- radio buttons: *objectName*: radioOpt1 and radioOpt2, *text*: 'Option 1' and 'Option 2'

The entire widget is saved under the filename *testWidget.ui* in an arbitrary directory.

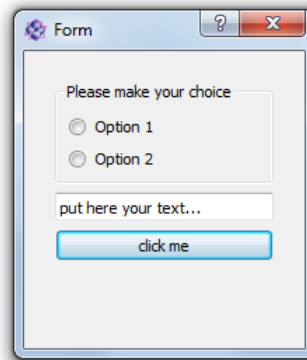
Then you can load and show the widget in **itom** by creating a python-script in the same directory with the following content. You can also directly type these lines into the command line of **itom**, however, you should then assure that the current directory is equal to the directory where the user interface has been stored.

```
dialog = ui("testWidget.ui", ui.TYPEDIALOG) #loading dialog
result = dialog.show(1) #modally show, wait until the dialog has been closed
print("The dialog has been closed with code", result)
```

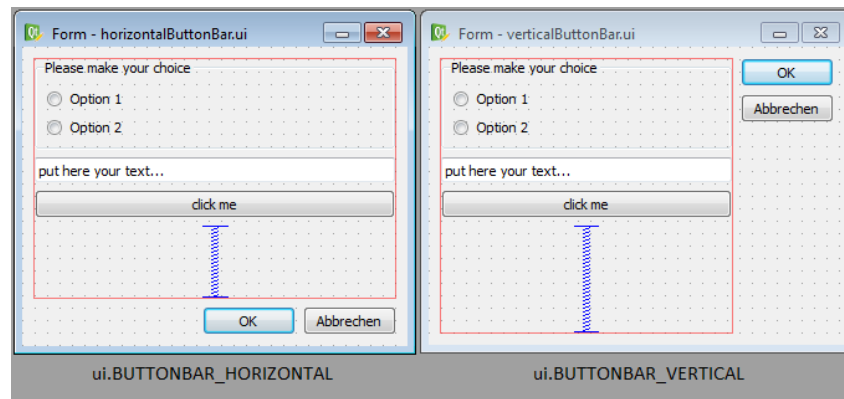
At first, an instance of class `itom.ui` is created that is given the name of the user interface file. This instance can then be accessed by the name *dialog*. By calling the method `show()`, the dialog is shown. Since the parameter has been set to **1**, the dialog is shown in a modal style, such that python waits until the dialog has been closed again and **itom** is entirely blocked during that time. However, then it is possible to get informed about the way the dialog is closed, such that the variable *result* will be set to **0** if the user closed the dialog using a cancel button (not available here) or the close button in the title bar or **1** if the user clicked an **OK**-button.

It is also possible to open the dialog in a non-modal version or to open it in a modal style however to immediately force python to continue the script execution. This depends on the parameters of `show()`. However only in the modal case above, the closing result can be tracked by **Python**. Additionally, this is also only possible if a widget is embedded in a dialog, given by **itom**, like it is always the case if you create an instance of `itom.ui` with the second parameter set to `ui.TYPEDIALOG`.

```
>>dialog = ui("testWidget.ui", ui.TYPEDIALOG)  
result=dialog.show(1)
```



Right now, you don't have the possibility to quit the dialog using any button (**OK**, **Cancel**...). In order to obtain a button bar with these buttons, the call to the class `itom.ui` needs to be changed. There is the choice between two different appearances of a button bar, which can be automatically added to your widget:



Next, you need to select which buttons should be included in the button bar. This is done by creating a python dictionary, where each elements corresponds to one button. The key-word of the item corresponds to the role of the button (see enumeration `QDialogButtonBox::ButtonRole`* of the **Qt**-library documentation) and the value is the text of the button. Common roles are:

- “AcceptRole”: Use this role for an **OK**-button. The dialog is closed and the return value in modal style is 1.
- “CancelRole”: Use this role for a **Cancel**-button. The dialog is also closed but the return value is 0.

Finally, the call to `itom.ui` must be in the following way, in order to get an auto-generated button bar:

```
dialog = ui("testWidget.ui", ui.TYPEDIALOG, ui.BUTTONBAR_VERTICAL, \
    {"AcceptRole": "OK", "CancelRole": "Cancel"})
#or
dialog = ui("testWidget.ui", ui.TYPEDIALOG, ui.BUTTONBAR_HORIZONTAL, \
    {"AcceptRole": "Yes", "CancelRole": "No"})
```

Note: You can also use a keyword-based call to `ui` since every parameter has its default value such that you can omit parameters beside the first one. For more details about all parameters, keywords and its default values see `itom.ui`.

The dialog is closed and deleted if the variable **dialog** is deleted using the command `del`.

Main window or dialog (TYPEWINDOW)

If you are not interested in the exact return value of the dialog but you want to have full control and all available functionalities of any dialog or main window, create an user interface based on a **dialog** or **main window** in **Qt Designer**.

The figure shows an exemplary user interface (**testWindow.ui**) that is based on a main window. On the right side, there have been added three buttons, nested in a vertical layout. On the left side, there is a list widget (objectName: **listWidget**, type: **List Widget**). Additionally a menu has been added that consists of three items.

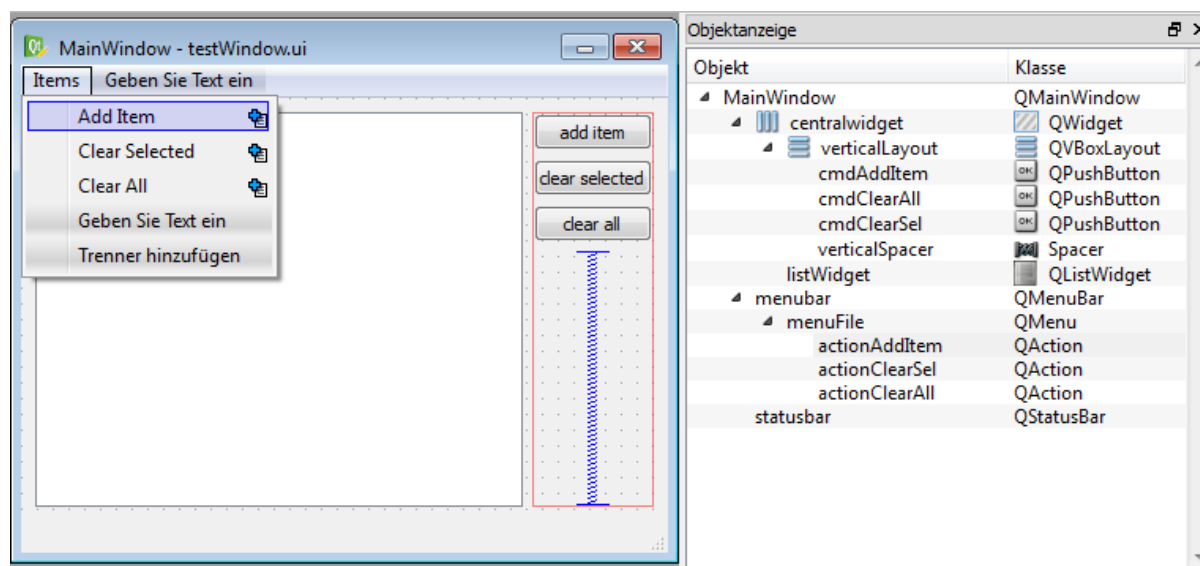
This main window can now be shown using the following code snippet:

```
win = ui("testWindow.ui", ui.TYPEWINDOW)
win.show() #this is equal to win.show(0) -> non-modal
```

Then, the window is shown on top of the main window of **itom**, since it is considered to be a child of **itom**. If you don't want this, you can also add the keyword-parameter `childOfMainWindow=False` to the call of `itom.ui`:

```
win = ui("testWindow.ui", ui.TYPEWINDOW, childOfMainWindow=False)
```

Here, you need to use the keyword, since the parameters `dialogButtonBar` and `dialogButtons` (used for `TYPEDIALOG`) are not given in this case, since they are useless in case of `TYPEWINDOW`. If your window is no child of **itom**, it gets its own icon in the Windows tray bar and does not stay on top of **itom**.



6.3.3 Accessing control elements

Until now, you know how to design an interface and how to show it using **itom**. This and the following sections explain how you dynamically interact with the user interface and its elements. One elementary tool for this is to access any desired element of the GUI. For instance, if you want to change properties of a button or the text of a linedit-widget, you first need to access these elements.

The accessing is simply done by the unique and specific **objectName** of each element and the dot-operator (.). Let's take the first example **testWidget.ui** again. The dialog has been assigned the variable **dialog**. Then have the following possibilities to access its elements:

```
elemGroup = dialog.groupOption # access the group box by its objectName

elemRadioOpt1 = dialog.radioOpt1 # OR
elemRadioOpt1 = elemGroup.radioOpt1

textfield = dialog.txtInput
```

Each variable created by the code block above is an instance of `itom.uiItem`. It is this class that defines the dot-operator. Looking at the example of accessing the first radio button, which is a child of the group-box, it is both possible to access the group button by its `objectName` as child of the entire dialog or as child of the groupbox. This is feasible since the class `ui` is derived from `uiItem`, such that the dot-operator not only works for entire dialog references but also for accessing sub-elements of other widgets. However, since each `objectName` is unique among all elements of the entire dialog, it doesn't matter how to access any element.

But why do we need to access these elements? Why do they return its own instance of class `uiItem`. These questions are answered in the following sections...

6.3.4 Getting and setting properties

As already mentioned, you can read or write most properties of any element that are also listed in the property toolbox of **Qt Designer**. Properties are also separately listed in the corresponding **Qt** documentation. In general it makes sense to set properties - when offline possible - in the **Qt Designer**. This is a little bit more efficient and keeps your script tiny. Getting and setting properties is possible if you have an object of type `uiItem`. Therefore you need to get this object like described in the section above.

Getting the property value can either be done by using the mapping-operator `[]` or by using the method `getProperty()`. For instance, if you want to get the current text and the enabled status of the textfield in dialog **testWidget.ui** from the first example, you can use one of the following possibilities:

```
#1. possibility
text = dialog.txtInput["text"]
enabled = dialog.txtInput["enabled"]

#2. possibility
[text,enabled] = dialog.txtInput.getProperty(["text", "enabled"])
```

In order to set one or multiple properties, you can use similar methods. Simply assign a value to the mapping-operator [] or use the method `setProperty()`.

```
#1. possibility
dialog.txtInput["text"] = "new text for this textfield"
dialog.txtInput["enabled"] = False

#2. possibility
dialog.txtInput.setProperty( {"text":"new text for this textfield", \
                             "enabled":False} )
```

If you use `setProperty()`, you always need to pass a dictionary as argument. This dictionary can contain one or multiple properties, where the keyword always is the property-name (string) and the value is the corresponding new value (type depends on corresponding C++ type). For more information about supported datatypes, that can be accessed by python in **itom** see *Supported datatypes*.

6.3.5 Supported datatypes

The classes `itom.ui` and `itom.uiItem` are the connection between any python-script in **itom** and GUI-elements, written in C++ and provided by **Qt**. Therefore, it is necessary to transform types from python to corresponding C++-structures and vice-versa. The following table lists some convenient type casts. In general, it is always tried to convert the input type to the desired destination type, such that a number can also be transformed to a string, if it is always known, that the destination requires a string.

C++/Qt-Type	Python-Type
QString	str or any type, that has a string representation
QByteArray	unicode or byte type
QUrl	any string that can be interpreted as Url
bool	any type that can be casted to a boolean value (1,0,True,False...)
QStringList	any sequence that only contains values castable to QString
int, short, long	integer, floats are rounded to integer, True=1, False=0
unsigned int ...	integer, floats are rounded to integer, True=1, False=0
float, double	integer, floats, True=1.0, False=0.0
QVector<int>	any sequence whose values are castable to int
QVector<double>	any sequence whose values are castable to double
PCLPointCloud	<code>pointCloud</code>
PCLPoint	<code>point</code>
PCLPolygonMesh	<code>polygonMesh</code>
DataObject*	<code>dataObject</code> or any type convertible to an array (see numpy)
AddInDataIO*	<code>dataIO</code>
AddInActuator*	<code>actuator</code>
QVariant	any of the types above can be transformed to QVariant
QVariantMap	a dictionary where keys are strings and values are generally convertible.
QVariantList	any sequence whose items can be convertible.
QRegion	<code>region</code>

If a property or other arguments in **Qt** require other datatypes, it is possible to implement a converted for them. It only becomes a little bit more difficult for pointers to extended C++ or **Qt** classes. The conversion is mainly done in the **itom** class `PythonQtConversion`.

6.3.6 Connecting signals

Now, you know how to change properties of dialogs at runtime of **itom** using a small python script snippet. In this section, you will learn how you can let **itom** a specific python-method for instance if a button on the user interface is clicked. Whenever something is changed in a user interface or the user starts to interact with the interface, any type of event is emitted. In **Qt** many of these events are specially handled and called signals. For instance, if an user clicks a button, toggles a checkbox, triggers an item in a menu or selects an item in a list widget, a signal is emitted or sent.

The counterpart to a signal is called slot. **Qt** provides the possibility to **connect** a signal with a slot, under the only condition, that both have exactly the same order and type of arguments. It is even possible to connect the same signal to various slots. Whenever a signal is emitted, all connected slots are executed. **itom** provides you the possibility to define slots in form of ordinary python methods or functions and to also connect them to signals of widgets on your user interface.

For establishing the connection, you need again a reference to the specific widget on the user interface. This reference is any variable of type `uiItem`. Next, you need the name and the arguments of the **Qt** signal, you want to connect to. This information can be obtained by the **Qt** documentation. For instance, if you need any signal that a widget of type **QPushButton** (the type of our push button, placed in the user interface in file `testWidget.ui`), go to <http://qt-project.org/doc/qt-4.8/qpushbutton.html>. Unfortunately, you won't find a headline called **Signals** at this page, since **QPushButton** does not directly declare any signal. However, you can see under **Additional Inherited Members**, that **QPushButton** inherits signals from its base classes. The most important signals are inherited from **QAbstractButton**. Click on its link and you will see the available signals for a push button:

```
void clicked ( bool checked = false )
void pressed ()
void released ()
void toggled ( bool checked )
```

If any argument provides a default value, you can also omit the specific argument. Select the signal that is convenient for you and create its string-signature. The signature always contains the following structure:

```
signature = "signalName(typeName1, typeName2, ...)"
```

For instance, the signatures for the signals above are:

```
"clicked()" or "clicked(bool)"
"pressed()"
"released()"
"toggled(bool)"
```

Then, create a python method in your script, which you want to consider to be a slot and that should be connected with the signal. This method always requires the same number of arguments than given in the signature. If you want to connect a signal to a method that is a bounded method of a class in python, the first argument **self** does not count to the number of total arguments, hence, you always need to define the first parameter **self**, like it is the case for bounded methods.

Finally, use the method `connect()` in order to establish the connection. For instance let us create a method, that should show a message when the push button “click me” on the first exemplary dialog (`testWidget.ui`) has been clicked:

```
dialog = ui("testWidget.ui", ui.TYPEDIALOG, ui.BUTTONBAR_VERTICAL, \
    {"AcceptRole": "OK", "CancelRole": "Cancel"})

def showMsg():
    #slot executed in button 'click me' is clicked
    ui.msgInformation("itom", "you pressed the button click me")

#connect(signature, method)
dialog.btnClickMe.connect( "clicked()", showMsg )

#show dialog
dialog.show()
```

You have seen that the method `connect()` of the element `dialog.btnClickMe` (the push button) has been called. Its first argument is the signature of the signal, as second argument the reference to the slot-methods is given. If you integrate the dialog within a class and the slot is a member of this class, too, the exemplary code can look as follows:

```
class MyDialog():

    def __init__(self):
        self.dialog = ui("testWidget.ui", ui.TYPEDIALOG, ui.BUTTONBAR_VERTICAL, \
            {"AcceptRole": "OK", "CancelRole": "Cancel"})
        self.dialog.btnClickMe.connect("clicked()", self.showMsg)
        self.dialog.show()

    def showMsg(self):
        ui.msgInformation("itom", "you pressed the button click me")

#instance of class MyDialog
test = MyDialog()
```

Let us use the second example `testWindow.ui`. If you want a python method to be executed if the user clicks an action in the menu of the main window, you should connect the signal `triggered()` of every item in the menu with your method. In **Qt** such an item is an instance of *QAction* and is also accessed by its *objectName*.

```
win = ui("testWindow.ui", ui.TYPEWINDOW)

def addItem():
    print("action addItem clicked")

win.actionAddItem.connect("triggered()", addItem)
#actionAddItem is the objectName of the action
win.show()
```

6.3.7 Calling slots

Widgets on user interfaces not only emit signals but they also have slots defined, such that you can connect other signals (e.g. from other widgets) to these slots. Using a python script in **itom** you can also call (or: invoke) these slots.

In order to invoke a slot, call the method `call()` of any element on your user interface. For instance, in order to clear the list widget (*objectName: listWidget*) of `uiWindow.ui`, you can invoke its public slot `clear()`:

```
win = ui("testWindow.ui", ui.TYPEWINDOW)
listWidget = win.listWidget
listWidget.call("clear")
```

Here, the method `call()` is only called with one argument, the name of the slot in **Qt**. If this slot would have any arguments that can be converted from **Python** (see *Supported datatypes*), add these arguments as further parameters to the call.

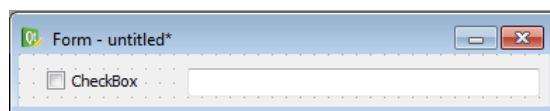
Unfortunately, there are some methods of important widgets in **Qt**, which are not defined to be a *public slot*. For instance, the methods to add item(s) to a list widget are no slots. However, there are some exceptions defined in **itom** such that some *public methods* of widgets can also be called with the method `call()`. These exceptions are contained in the following table:

Widget / ClassName	Public Method
QWidget	void resize(int,int)
QWidget	void setGeometry(int,int,int,int)
QListWidget	void addItem(QString)
QListWidget	void addItems(QStringList)
QComboBox	void addItem(QString)
QComboBox	void addItems(QStringList)
QTabWidget	int isTabEnabled(int)
QTabWidget	void setTabEnabled(int,bool)
QMainWindow	uiItem statusBar() <i>returns a reference to the statusbar widget</i>
QMainWindow	uiItem centralWidget() <i>returns a reference to the central widget of the mainWindow</i>
QTableWidget	void setHorizontalHeaderLabels(QStringList)
QTableWidget	void setVerticalHeaderLabels(QStringList)
QTableWidget	QVariant getItem(int,int)
QTableWidget	void setItem(int,int,QVariant)
QTableView	uiItem horizontalHeader()
QTableView	uiItem verticalHeader()

Please notice, that every method listed above is also valid for a widget, that is derived from the specific class (derived in C++). Therefore the additional slots of *QWidget* hold for every other widget, since every widget is derived from *QWidget*.

6.3.8 Connecting internal signals and slots in Qt Designer

If you want to connect the signal, emitted by any widget, with a slot from another widget, you will learn in this section how to do this. This type of connection can for instance be useful if you want to enable or disable certain widgets depending on the status of other ones, like the check-status of a checkbox. The following figure shows an user interface with a checkbox and a textfield. Let us define a signal-slot-connection, such that the textfield gets disabled if the checkbox is unchecked.



This type of gui-internal connections are completely done in **Qt Creator**. Therefore chose the “Signal and Slots”



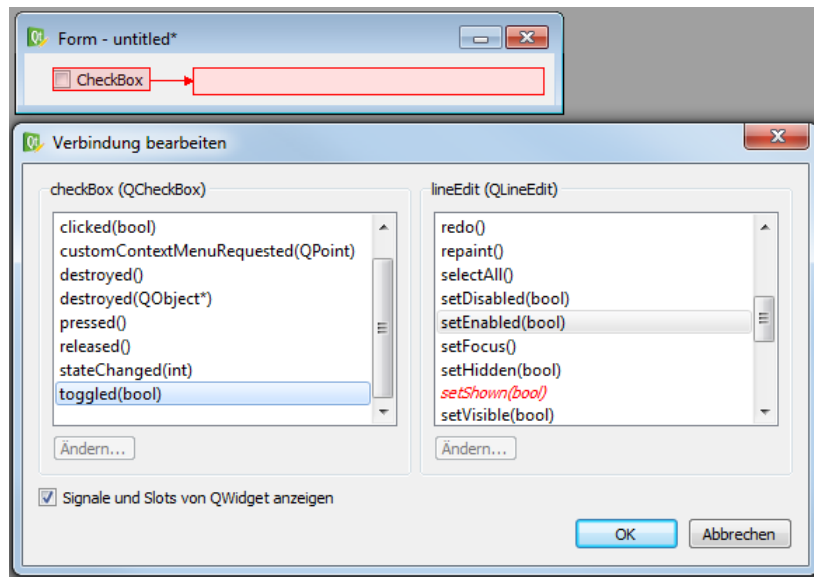
editing mode, that is obtained by clicking the symbol in the toolbar or by pressing *F4*. Then you can make a drag&drop connection between the emitting widget and the receiver-widget. After releasing the mouse button, the connection dialog, depicted in the following figure becomes visible:

Here you can choose which signal of the emitting widget should be connected with which slot of the destination. At the beginning, only slots and signals of the specific widget classes are visible. However, you can check the checkbox below, in order to also show the signals and slots of the inherited classes. Please make sure, that you only choose pairs of signals and slots which have the same parameter types. In our case, we connect the signal *toggled(bool)* with the slot *setEnabled(bool)*, which is the setter-method of the property *enabled*.

This example has also shown, that not only slots defined in the slot-section can be called as slots, but also every setter-method of any property can be called like every slot. However, in this case it is more convenient the property like described in section [Getting and setting properties](#).

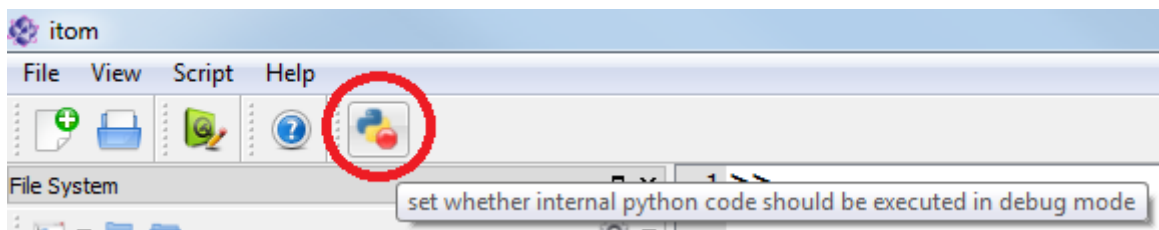
6.3.9 Auto-connecting signals using python decorators

TODO



6.3.10 Debugging user interfaces and slot-methods

If you established a signal-slot-connection between an element of the GUI and a **Python**-method, you probably want to debug this method once the signal has been emitted. This is obtained by setting any breakpoint into the specific line and toggling the button *Run python code in debug mode* in the menu **Script** of **itom**.



6.3.11 Hints and limitations

All methods described in this chapter explain how to create and use user-defined dialogs and windows using **Python** scripts in **itom**. Finally, all dialogs are created using the **Qt**-framework. The classes `itom.ui` and `itom.uiItem` finally are wrappers for the underlying **Qt**-system. Using pure python, similar things can also be obtained with the famous packages **PyQt** or **PySide**. However, in **itom** you must not use these packages. The reason is, that create a new main instance of the **Qt**-engine, that needs to be created in the main thread. This is the case, if **PyQt** or **PySide** is executed directly in **Python**. However using **itom Python** is embedded as scripting language, such that **itom** is executed in the main thread while **Python** is moved to its own additional thread. The reason is to enable the execution of longer scripts, while the main application **itom** still keeps reactive. Therefore, **Python** does not have access to the real main thread and it is forbidden to explicitly execute some GUI-related stuff in secondary threads. Therefore all methods in `itom.ui` and `itom.uiItem` have thread-safe implementations and communicate with an organization structure, that runs in the main thread of **itom**, in order to interact with all dialogs.

6.4 Implement a more complex GUI in a plugin (C/C++)

PLUGINS

Stuff about plugins

7.1 Basic concept

The basic **itom** -plugin concept

7.2 How to start a plugin

GUI and script

7.3 Development under C++

The functionality of *itom* can be enlarged by programming plugins (sometimes also called add-ins). Such a plugin is written in C++ and can be integrated into *itom* using the possibilities which are given by *Qt* (plugin management of *Qt*). Additionally each plugin must fit to one of three possible interface classes, such that *itom* is able to communicate with this plugin. Finally, the resulting plugin is compiled as a *dll*-file (on Windows) or as an appropriate *a*-file on Linux-based machines and must be located in the *plugin*-folder of *itom* or any subfolder. Then, the plugin is automatically recognized at startup of *itom*.

For programming plugins, you can and should use basic structures, offered by **itom**:

7.3.1 RetVal - The return type of itom methods

The class **RetVal** is used for creating, transmitting and handling return values of methods in **itom** and plugins. This class is defined in the file *sharedStructures.h* in the folder *common* of the **itom** base directory.

Any return value consists of the following main components:

- **error state (enumeration `ito::tRetValue`)**
 - `retOk`
 - `retWarning`
 - `retError`
- error number (user-defined error number; this number has no further functionality yet)
- error message (NULL or the warning or error message of the return value)

You can create a variable of type **RetVal** using different constructors or the static method **RetVal::format**.

`ito::RetVal::RetVal (tRetVal retValue=retOk)`

creates a return value with the default state *retOk*, if not otherwise stated. (constructor)

`ito::RetVal::RetVal (int retValue)`

creates a return value with the given state. (constructor)

`ito::RetVal::RetVal (ito::tRetVal retValue, int retCode, const char* pRetMessage)`

creates a return value with a given state, error number and error message. If you don't want to indicate an error message, set that value to NULL. (constructor)

static `ito::RetVal ito::RetVal::format (ito::tRetVal retValue, int retCode, const char* pRetMessage, ...)`

Use this static method to create a new return value where the message string can contain placeholders, known by the ordinary methods **sprintf**, ... The values filled into these placeholders are then appended as additional parameters to this method.

`RetVal& operator= (const RetVal rhs)`

By this operator you can assign a new return value to this return value.

Additionally you can use the mathematical operators **+** and **+=** to add a return value to an existing instance of class **RetVal**. The resulting return value keeps unchanged if the state of the added return value is less critical than the internal state, e.g. a state *retOk* is less critical than *retError* as well as *retWarning* is less critical than *retWarning*. If both states are the same, the return value is unchanged, too. In any other cases the state, error number and error message of the added return value is used for the new return value.

By this mechanism you can create chains of return values in the following form:

```
ito::RetVal retValue = ito::retOk
retValue += method1()
retValue += method2()
retValue += method3()

if (retValue.containsError())
{
    print("error while executing some methods");
}
```

The actual status of the return value can be obtained using the following methods:

int containsWarning ()

Returns true if the error state is equal to *retWarning*, that means the worst error state which has been added to this return value has been *retWarning*.

int containsError ()

Returns true if the error state is equal to *retError*, that means the worst error state which has been added to this return value has been *retError*.

int containsWarningOrError ()

Returns true if the worst error state, which has been added to this return value has been unequal to *retOk*.

char* errorMessage ()

Returns a zero-terminated string containing the actual error-message of this return value or NULL.

Additionally you can use the comparison operators **==** or **!=** to compare the error state of two return values or the error state of one return value with a given error state.

Note: For a full reference of the class **RetVal** see [RetVal - Reference](#).

RetVal - Reference

class **ito::RetVal**

Class for error value management.

The *RetVal* class is used for handling return codes. All classes should use this class. In case an error occurs, only the first error is stored and will not be overridden by potentially subsequent occurring errors.

Public Functions

RetVal(tRetVal retValue = retOk)

RetVal(int retValue)

RetVal(ito::tRetVal retValue, int retCode, const char * pRetMessage)

constructor with retValue, retCode and errorMessage

Parameters

- retValue - return Value constant, for possible values see tRetVal
- retCode - user definable return code
- pRetMessage - error message to be passed, string is copied Makes a deep copy of *RetVal*, i.e. a copy of the error message

RetVal(const *RetVal* & copyConstr)

Deep copy constructor Makes a deep copy of *RetVal*, i.e. a copy of the error message

~RetVal()

Destructor, in case frees errorMessage string

RetVal & **operator=**(const *RetVal* rhs)

assignment operator, copies values of rhs to current *RetVal*. Before copying current errorMessage is freed

RetVal & **operator+=**(const *RetVal* rhs)

Concatenation of *RetVal* “Adds” RetVals, i.e. returns the most serious error. In case of equally serious errors the first is retained

RetVal **operator+**(const *RetVal* rhs)

Concatenation of *RetVal* See operator *RetVal::operator+=*

char **operator==(const *RetVal* rhs)**

equality operator compares retValue with with retValue of rhs *RetVal*. For possible constant values see tRetValue

char **operator!=(const *RetVal* rhs)**

unequality operator compares retValue with with retValue of rhs *RetVal*. For possible constant values see tRetValue

char **operator==(const tRetValue rhs)**

equality operator compares retValue with tRetValue constant. For possible constant values see tRetValue

char **operator!=(const tRetValue rhs)**

unequality operator compares retValue with tRetValue constant. For possible constant values see tRetValue

int **containsWarning()**

checks if any warning has occurred in this return value (true), else (false)

int **containsError()**

checks if any error has occurred in this return value (true), else (false)

int **containsWarningOrError()**

checks if any warning or error has occurred in this return value (true), else (false)

char * **errorMessage()**

int **errorCode()**

void **appendRetMessage(const char * addRetMessage)**

Public Static Functions

RetVal **format**(ito::tRetVal retValue, int retCode, const char * pRetMessage, ...)

7.3.2 ItomSharedSemaphore

In **itom** different main components are executed in different threads.

Note: What is a thread

In a modern operating system, different programs can run simultaneously. Then, every program is executed in its own process and the operating system is responsible such that each process continuously gets some processing-time on the processor. If your computer has multiple processors, then the operating system can assign processing time of different processes to different processing units.

Every program itself can now run different components of its own software in different threads. Thread are comparable to processes on operating system level, however threads can share memory, such that every thread can have access to a common set of global variables, instances... The advantage is, that a thread can execute a time-consuming calculation while the other thread is responsible for the graphical user interfaces, which now is not blocked while the calculation thread is working.

For more information about threads see [Wikipedia \(english\)](#) or [Wikipedia \(german\)](#)

In **itom**, the following components run into a different thread:

1. Main programm including all graphical user interface components (this is a restricting of windows operating systems). This thread is called “main thread”.
2. Python scripting engine. All python scripts are executed in a second thread. Therefore you should not directly use any 3rd-party python modules which open any graphical user interface (like *PyQt*, *tkInter* ...).
3. Every *DataIO* or *Actuator*-plugin is executed in its own thread. Every algorithm is executed in the thread of the calling instance (e.g. main thread or python thread). All widgets provided by any *Algo*-plugin are always executed in the *main thread* (see restricting of point 1).

In figure *Figure: Scheme of a communication between itom (main thread) and any plugin-method (plugin is running in another thread)*, a common communication between a method **mainfunc** which is executed in **itom**’s main thread (e.g. in *AddInManager* and another method **func1** of a plugin is depicted. The plugin is of type *actuator* or *dataIO* and therefore runs in its own thread. **mainfunc** is calling the method **func1**. Usually the calling method has to wait until the called method (**func1**) has been finished (or until the most important parts of **func1** have been executed). In order to consider cases where the plugin-method is not answering within a certain timeout, **itom** has integrated a mechanism such that the waiting-process in the main-function can be stopped after that the timeout time expired. This mechanism is realized by the class **ItomSharedSemaphore**, which is defined in the file *sharedStructuresQt.h* (folder *common*).

ItomSharedSemaphore consists of the following elements:

QSemaphore *m_pSemaphore

ItomSharedSemaphore internally contains an instance of *QSemaphore*, the platform independent semaphore provided by **Qt**. A semaphore can be considered as set of *gaming piece*. Any method executed within different threads can take one or more *gaming pieces* from the common semaphore. If there are not enough *gaming piece* in the storage, the method has to wait until the necessary number of *gaming pieces* become available. Any method which exits some “critical sections” in the code release their previously picked *gaming pieces*, such that other instances can repick them. In most cases in the communication between **itom** and a plugin-method, the semaphore contains one *gaming piece*, since only two partners are participating in the communication: the method in the main thread and the called method in the plugin-thread. The method in the plugin-thread is picking the *gaming piece* and the calling method has to wait until the *gaming piece* gets back to the storage of the semaphore. This is achieved by *releasing* the *gaming piece* in the called method.



Figure 7.1: Figure: Scheme of a communication between **itom** (main thread) and any plugin-method (plugin is running in another thread)

int m_numOfListeners

This is the number of participants at a communication process (without the calling method). Usually only one method is called, therefore this value is usually equal to 1. This value usually is automatically set.

int m_instCounter

This value indicates how many *gaming pieces* are out of the storage. If the called method is still executing, this value is equal to one. If the called method already release the semaphore, hence *gaming piece*, the value drops to zero.

bool isCallerStillWaiting

This method returns true if the calling method (*here*: the method in main thread) is still waiting that the called method has released the semaphore, else the method returns false.

ito::RetVal returnValue

The called method in the plugin-thread can set that globally accessible return value and the calling method can read its value. That return value is the single way to return the success or error message to the calling method, since no return value of the called method (slot) can directly be returned.

Scheme of the inter-thread communication

1. The calling method **mainfunc** creates an instance of *ItomSharedSemaphore*:

```
ItomSharedSemaphore *waitCond = new ItomSharedSemaphore();
```

2. Method **func1** is called in the plugin-thread. This can only be done if **func1** is declared as **slot** (*signal-slot-system* of **Qt**). The call is executed using the method **invokeMethod** of *QMetaObject*:

```
QMetaObject::invokeMethod(instance-of-plugin, "func1", Q_ARG(paramType1, paramValue1), ..., Q_ARG(paramTypeN, paramValueN));
```

For the use of the macros **Q_ARG** and the method **invokeMethod**, see also the documentation of **Qt**.

3. Then **func1** is executing and their last parameter *waitCond* contains the pointer to the instance of *ItomSharedSemaphore*.
4. If **func1** has been finished, you should write the return value to the instance of *waitCond* and release the semaphore, hence, return the *gaming piece* to the global storage of the semaphore:

```
if(waitCond) //can also be NULL, therefore check it
{
    waitCond->returnValue = ito::RetVal(ito::retOk)
    waitCond->release()
}
```

5. The caller has to wait until the semaphore has been released or a timeout expired. This is done by the member **wait** of **ItomSharedSemaphore**. The single argument of that method is the timeout in milliseconds. Usually you can use the global variable **PLUGINWAIT**, which is set to **5 seconds**. The *wait*-method returns *true* if the semaphore has been released within the timeout, else it returns false:

```
if(!waitCond->wait(PLUGINWAIT))
{
    //timeout occurred
}
```

5. Finally all participants at the communication process (*here*: caller and called method) have to delete the semaphore. Be careful: This can not be done by simply **deleting the pointer to waitCond**. Instead both the caller and the called method have to execute the following command, hence, call the method **deleteSemaphore** of the semaphore pointer:

```
waitCond->deleteSemaphore()
```

6. After that the last participant at the communication process deleted the semaphore, it is really deleted by **itom**. Then you don't have access to the semaphore any more, hence, you also don't have access to the internal return value.

In order to simplify the process of deleting the semaphore, both the caller and the calling method can also create a variable of type **ItomSharedSemaphoreLocker**, where the pointer to **waitCond** has to be given as constructor-argument. If this variable finally is destroyed, which is automatically done if the method is finished - even after that the *return* command has been executed - it calls the *deleteSemaphore*-method of *ItomSharedSemaphore* with *waitCond* as argument.

Then the scheme of the caller is:

```
ItomSharedSemaphoreLocker locker(new ItomSharedSemaphore())
QMetaObject::invokeMethod(plugin-instance, "func1", Q_ARG(paramType1, paramValue1), ..., Q_ARG(ItomSharedSemaphore, waitCond))

if(!locker.getSemaphore()->wait(PLUGINWAIT))
{
    //timeout
    retvalue += ito::RetVal(ito::retError, 0, "timeout while executing method");
}
else
{
    retvalue += locker.getSemaphore()->returnValue;
}
```

and the scheme of the called method **func1** is:

```
void func1(type1 param1, ..., ItomSharedSemaphore* waitCond)
{
    ItomSharedSemaphoreLocker locker(waitCond)
    ... do some calculation ...

    if(waitCond)
    {
        waitCond->returnValue = ito::retOk;
        waitCond->release();
    }

    return
}
```

Note: For more information about the implementation of class **ItomSharedSemaphore** see [ItomSharedSemaphore - Reference](#) or the doxygen documentation for **itom**.

ItomSharedSemaphore - Reference

class ItomSharedSemaphore

semaphore which can be used for asynchronous thread communication. By using this class it is possible to realize timeouts.

This semaphore is usually applied if any method invokes another method in another thread and should wait for the called method being terminated or the waiting routine drops into a possible timeout. Therefore, the calling method must create an instance of *ItomSharedSemaphore* with a number of listeners equal to one. Then, the pointer to *ItomSharedSemaphore* is transmitted to the called method (usually as last argument). If the called method is done or wants the caller to continue the release-method of *ItomSharedSemaphore* is called. The calling method calls the wait-method of the semaphore which blocks the method until the semaphore is released. Finally, both the caller and the calling method must call *ItomSharedSemaphore::deleteSemaphore* in order to decrease the reference counter from two to zero, which allows the system to delete the semaphore.

Consider to guard an instance of *ItomSharedSemaphore* by the capsule-class *ItomSharedSemaphoreLocker* both in the caller and calling method, such that the decrease of the reference counter is exe-

cuted if the `ItomSharedSemaphoreLocker`-variables are deleted, e.g. if they run out of scope, which is even after the return-command of any method.

Public Functions

ItomSharedSemaphore(int numberOfListeners = 1)

constructor

A new *ItomSharedSemaphore* is created and the underlying semaphore is already locked with a certain number. This number depends on the number of listeners (usually: 1). A listener is a method which is called in another thread. The caller creates the *ItomSharedSemaphore* in order to wait until the listener decreases the lock level of the semaphore by one or a certain timeout time has been reached.

Parameters

- `numberOfListeners` - (default: 1) are the number of different methods in other threads which should release this semaphore before the caller can go on.

~ItomSharedSemaphore()

destructor (do not call directly, instead free the semaphore by *ItomSharedSemaphore::deleteSemaphore*)

bool **wait**(int timeout)

The call of this method returns if a certain timeout has been expired or every listener released the semaphore.

bool **waitAndProcessEvent**(int timeout, QEventLoop::ProcessEventsFlags flags = QEventLoop::AllEvents)

mutex initialization

The call of this method returns if a certain timeout has been expired or every listener released the semaphore.

void **release**()

decreases the number of locks by one

int **available**()

checks whether the semaphore is still locked or not

bool **isCallerStillWaiting**()

indicates whether caller-method is still waiting that the lock is released by the listener(s).

void **deleteSemaphore**(void)

static method to decrease the reference counter of any *ItomSharedSemaphore* or delete it if the reference counter drops to zero

Public Members

ito::RetVal **returnValue**

public returnValue member variable of *ItomSharedSemaphore*. This return value can be used to return the result of any called method (listener) to the caller. Please access this value in caller only if the wait-method returned with true. Write to returnValue in called method (listener) before releasing the semaphore. This is important, since the returnValue is not fully thread safe.

7.3.3 DataObject

The class **DataObject** (part of the library **dataObject**) provides a *n*-dimensional matrix that is used both in the core of **itom** as well as in any plugins. The *n*-dimensional matrix can have different element types. These types and their often used enumeration value are defined in the file *typeDefs.h* and are as follows.

Typedef	Enumeration	Description
ito::int8	ito::tInt8	8bit, signed, fixed point
ito::uint8	ito::tUInt8	8bit, unsigned, fixed point
ito::int16	ito::tInt16	16bit, signed, fixed point
ito::uint16	ito::tUInt16	16bit, unsigned, fixed point
ito::int32	ito::tInt32	32bit, signed, fixed point
ito::uint32	ito::tUInt32	32bit, unsigned, fixed point
ito::float32	ito::tFloat32	32bit, single-precision floating point
ito::float64	ito::tFloat64	64bit, double-precision floating point
ito::complex64	ito::tComplex64	real and imaginary part is float32 each
ito::complex128	ito::tComplex128	real and imaginary part is float64 each

The last two dimensions of each dataObject are denoted *plane* and physically correspond to images. In order to also handle huge matrices in memory, The entire matrix is divided into planes, where each plane can be allocated at arbitrary positions and memory and is of type **cv::Mat_<type>**, derived from **cv::Mat** of the **OpenCV** library. Therefore every plane can be used with every operator given by the **OpenCV**-framework (version 2.3.1 or higher). If available, the first (*n*-2) are allocated as one vector of pointers, each pointing to its corresponding plane. This kind of dataObject and its way of allocating memory is called *unorganized*.

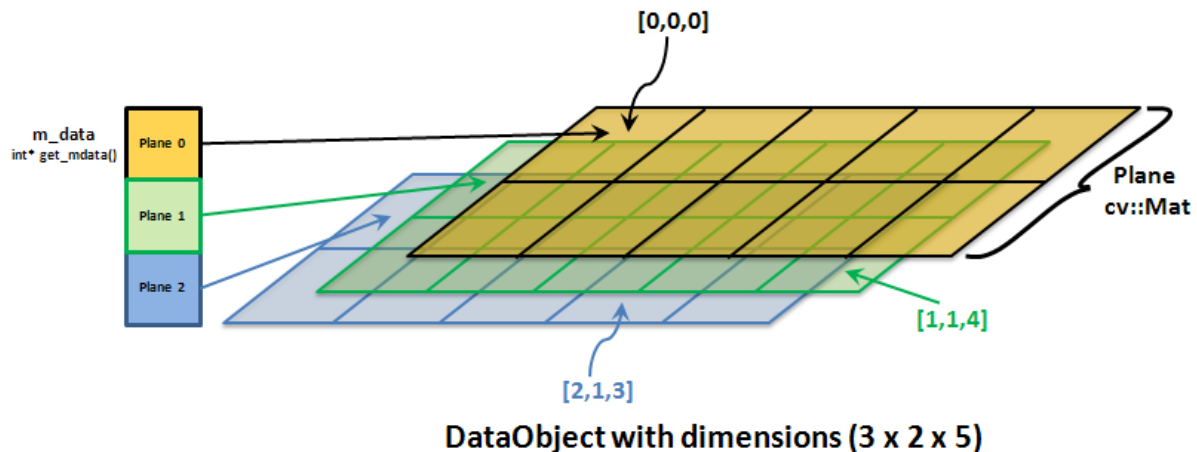
In order to make the *dataObject* compatible to matrices that are allocated in one huge memory block (like Numpy arrays), it is also possible to make any *dataObject* continuous. Then, a huge data block is allocated, such that all planes lie consecutively in memory. Nevertheless, the pointer-tree is still available, pointing to the starting points of all planes. This reallocation is implicitly done, when creating a Numpy-array from a dataObject.

DataObject can be declared in different possible ways with different dimensions and different data types. Various possible implementations of declaring DataObject are listed below.

1. DataObject();
2. DataObject(const size_t size, const int type);
3. DataObject(const size_t sizeY, const size_t sizeX, const int type);
4. DataObject(const size_t sizeZ, const size_t sizeY, const size_t sizeX, const int type ,const unsigned char continuous = (unsigned char)'000');
5. DataObject(const unsigned char diimesions, const size_t *sizes, const int type, const unsigned char continuous = (unsigned char)'000');
6. DataObject(const unsigned char dimensions, const size_t *sizes, const int type, const uchar "continousou-DataPtr, const size_t *steps = (const size_t *)0);

7. `DataObject(const size_t sizeZ, const size_t sizeY, const size_t sizeX, const int type, const uchar*continuousDataPtr, const size_t *steps = (const size_t *)0);`
8. `DataObject(const ito::DataObject &copyConstr);`
9. `DataObject(const unsigned char dimensions, const size_t *sizes, const int type, const cv::Mat *planes, const unsigned int nrOfPlanes);`
10. `DataObject(const ito::DataObject dObj);`

Let's take an example of a data object with a dimensions 3x2x5. It can be imagined as shown in the figure below.



As we can see in this figure, each plane is of a type `cv::Mat` class from `openCV` library which we know. `m_data` corresponds to the zero-based two-dimensional matrix-index for a particular plane. This matrix-index can be retrieved using `seekMat()` method and later on with the use of `get_mdata()` method, we can have the matrix pointer of the whole plane. The return type of `get_mdata()` method is `int*` which can be type-casted by `cv::Mat*` easily. Please read the section [Direct Access to the underlying cv::Mat](#) to understand this concept in detail with a working example.

Following are some sample codes to get one quickly understand *basic programming structure of Data Objects*.

Example Codes

itom

The following code creates empty data object with no dimensions (0) and no type (0).

```
1 ito::DataObject d0;
2 std::cout << "empty data object: \n";
3 std::cout << " dimensions: " << d0.getDims() << "\n";
4 std::cout << " type: " << d0.getType() << "\n" << std::endl;
```

Note: Here `getDims()` method returns the dimensions of data object `d0`, whereas `getType()` method returns the type of `d0`.

Now, let's take an example of creating a two dimensional data object.

The following code creates such 2 dimensional data object of dimensions Y=2, X=5 and of type float32.

```
1 ito::DataObject d1(2,5, ito::tFloat32);
2 std::cout << "2x5 data object, float32: \n";
3 std::cout << " dimensions: " << d1.getDims() << "\n";
4 std::cout << " type: " << d1.getType() << "\n";
5 std::cout << " size: " << d1.getSize(0) << " x " << d1.getSize(1) << "\n";
```

```
6 std::cout << " total: " << d1.getTotal() << "\n";
7 std::cout << d1 << std::endl;
```

Note: Here, `getSize()` method returns the size of each dimension of a data object `d1`. `getDims()` method returns the number of dimensions of `d1`. `getType()` method returns the type of `d1`. `getTotal()` method returns total number of elements in `d1`.

7.3.4 Addressing the elements of a data object

Now, we know how to create a data object, so let's have a look at how can one address the elements of a data object. Sometimes it is necessary to read or set single values in one matrix, sometimes one wants to access all elements in the matrix or a certain subregion. Therefore, the addressing can be done in one of the following ways:

Direct access of one single element of a Data Object using `at()` method

```
1 ito::DataObject d1(2,5, ito::tFloat32);
2 d1.at<ito::float32>(0,1) = 5.2;
3 std::cout << "d1(0,1) = " << d1.at<ito::float32>(0,1) << "\n" << std::endl;
```

Here, the addressing is done by the method `at()` under the class `ito::DataObject`, which is pretty similar to the same method under the **OpenCV** class `cv::Mat`. The `at()` method can either be used to get the value at a certain position or to set value at that position in a data object. There are special implementations of `at()` for addressing values in a two- or three-dimensional data object, where the first argument always is the **z-index** (3D), followed by the **y-index** and the **x-index**. All indices are zero-based, hence the first element can be referred by addressing **0th position** in every dimension.

Note: The `at()` method is templated where the template parameter must correspond to the type of the corresponding data object.

Let's try to summarize some pros and cons of this method.

Advantages

- This method gives flexibility to a developer to access any element of a data object directly.
- A developer can also access a part of a data object as well using `at()` method as described in [Direct Access to the underlying `cv::Mat`](#).

Drawbacks

- Developer has to implement the code under the nest of **if...else** conditions if one needs to access the whole data object.
- It is a slower method to execute performance wise if compared to the other methods of accessing the data object.

Addressing elements of a data object using row pointer

When one needs to iterate through certain region of a data object, then the previous method of accessing a data object using `at` method seems quite insufficient. In such case, one can define a row pointer for each row in matrix and work with row pointer to address elements of a data object in a following way.

```

1  ito::DataObject d1(3,5, ito::tInt16);
2  size_t planeID = d1.seekMat(0);    //get internal plane number for the first plane
3  ito::int16 *rowPtr = (ito::int16*)d1.rowPtr(planeID,0);
4  size_t height = d1.getSize(0);
5  size_t width = d1.getSize(1);
6  for(size_t m = 0 ; m< height ; m++)
7  {
8      rowPtr = (ito::int16*)d1.rowPtr(planeID,m);
9      std::cout << "Row " << m << " ";
10     for(size_t n=0 ; n < width; n++)
11     {
12         rowPtr[n] = m;                //accessing each element of data object w
13     }
14 }
15 std::cout << d1 << std::endl;

```

Here, **seekMat()** method gets the internal plane number of the 1st plane in line #2. In line #3, dynamic array **rowPtr** is defined as row pointer to the 0th plane of the data object **d1**. Now accessing each element of row pointer will access each element of the data object in that row.

To use this row pointer method for data objects more than 2 dimensions, following code can be used.

```

1  ito::int16 *rowPtr1= NULL;
2  int dim1 = d1.getSize(0);
3  int dim2 = d1.getSize(1);
4  int dim3 = d1.getSize(2);
5  int dim4 = d1.getSize(3);
6  int dim5 = d1.getSize(4);
7  size_t dataIdx = 0;
8  for(int i=0; i<dim1; i++)
9  {
10     for(int j=0; j<dim2; j++)
11     {
12         for(int k=0; k<dim3; k++)
13         {
14             dataIdx = i*(dim2*dim3) + j*dim3 + k;
15             for(int l=0; l<dim4; l++)
16             {
17                 rowPtr1= (TypeParam*)d1.rowPtr(dataIdx,l);
18                 for(int m=0; m<dim5;m++)
19                 {
20                     //Assigning unique value to each element of d1.
21                     rowPtr1[m] = cv::saturate_cast<TypeParam>(calcUniqueValue5D(i, j, k, l, m));
22                 }
23             }
24         }
25     }
26 }

```

Note: Here **dataIdx** represents the number of the plane in the matrix. The formula in line #14 assigns a non repeating increasing value to **dataIdx** such that each plane of the data object can be pointed out without any overlapping.

Some advantages and disadvantages of using this method are given in the section below.

Advantages

- This method is the most efficient way to access the data object.
- This method gives flexibility to access some rows or the full data object at once.

Drawbacks

- Complex implementation. One needs deep understanding of pointers to implement this method to access data object.
- This is not an advisable method if one needs to access a few elements of the data object which are not in sequence.

Assigning a single value to all elements of a Data Object

One can assign a single value to all elements of a data object using assignment operator “=” in the following way. Here we will also have a look on how to declare a 5 dimensional data object and assign a single floating point value to each element of the data object.

```
1 size_t *temp_size = new size_t[5];
2 temp_size[0] = 10;
3 temp_size[1] = 12;
4 temp_size[2] = 16;
5 temp_size[3] = 18;
6 temp_size[4] = 10;
7 ito::DataObject d1 = ito::DataObject(5,temp_size,ito::tFloat32);
8 d1 = 3.7;
9 std::cout << d1 << std::endl;
10 delete[] temp_size;
```

Here line #7 uses implementation #4 for declaring 5 dimensional data object d1.

Direct Access to the underlying cv::Mat

In some cases, one needs to assign values of elements of a data objects based on some portion of another data object. This can be done by using this method of accessing the underlying matrix (cv::Mat) of a data object directly. Following example shows the method to access underlying planes in multidimensional matrices.

```
1 // 4 x 5 x 3 DataObject, int16
2 ito::DataObject d4(4,5,3,ito::tInt16);
3 std::cout << "DataObject (4x5x3), int16 \n" << std::endl;
4 d4 = 3; //assign value 3 to all elements
5 //access to the third plane (index 2)
6 planeID = d4.seekMat(2);
7 cv::Mat *plane3 = (cv::Mat*)d4.get_mdata()[planeID];
8 std::cout << "OpenCV plane" << std::endl;
9 std::cout << *plane3 << std::endl;
10 //accessing second line in plane3
11 ito::int16* rowPtr2 = (ito::int16*)plane3->ptr(1);
12 //regions of interest
13 //d5 = d4[1:3,0:2,:];
14 ito::Range ranges[3] = { ito::Range(1,3), ito::Range(0,2), ito::Range::all() };
15 ito::DataObject d5 = d4.at(ranges);
16 d5 = 7;
```

Let's try to analyse the code above. As we can see in line #6, we used **seekMat()** method to retrieve the plane id of 3rd plane in 3 dimensional matrix d4.

line #7 declares a pointer variable plane3 of type cv::Mat to hold the contents of plane 3 of data object d4. Line #11 declares a row pointer to point a particular row in plane 3 of data object d4 as a revision to the previous method of accessing elements of a data object using row pointer.

line #14 defines the exemplary ranges to create a new data object d5 from a part of data object d4, which is done in line #15 with the use of **at()** method.

The other way to perform the same operation of line #14 is shown below.

```

1 ito::Range *ranges = new ito::Range[3];
2 ranges[0] = ito::Range(1,3);
3 ranges[1] = ito::Range(0,2);
4 ranges[2] = ito::Range::all();

```

This code shows the way to modify ranges individually, which can be very useful if one needs to modify this range later in this code to work on other data objects perhaps.

Note: `get_mdata()` is a function declared under *DataObject* class. It returns pointer to vector of *cv::_Matrices*.

Accessing all elements of a Data Object using iterators

There are two classes defined, called **DObjIterator** and **DObjConstIterator** respectively, under the namespace **ITOM**, which support the developer with an easy way to iterate through the whole data object. This method can be used only if one needs to iterate through all elements of a data object at once. Following code snippet shows the example of this method.

```

1 int temp = 0; // Temporary variable for indexing some arrays used in this test.
2 ito::DataObject d6(21,13,ito::tInt16); // Declaring a 21 x 13 data object with data type int16.
3 ito::DObjIterator it; // Declaration of DObjIterator
4 for(it=d6.begin();it!=d6.end();++it)
5 {
6     *((ito::int16*)(*it_2d)) = cv::saturate_cast<TypeParam>(temp++); // Assigning a un
7 }

```

As can be seen in the code above, line #2 declares a 21x31 data object **d6** of type **int16**. Line #3 declares an iterator object **it** of class **DObjIterator**. **DataObject** class contains **begin()** and **end()** methods to work with iterators. A brief description to this methods can be found under *plugin-DataObject-Ref* document. These methods contains pointers to the first and last elements of any data objects respectively. Line #4 makes a meaningful use of these methods in for loop to iterate through the data object **d6**. We first initiate the iterator **it** with the pointer returned by **d6.begin()**, iterate through the whole data object increasing the iterator value by one in each iteration till the pointer value in iterator **it** reaches the pointer value of the last element of the data object checking the condition **it!=d6.end()**.

Advantages

- This method is a compromise between its useability with ease and performance on execution level. Integration of this method in code is fast and easy.
- Developer does not think about **if...else** conditions to decide the boundaries of Region of Interest to access any data object.

Drawbacks

- Performance degrades against the method [Addressing elements of a data object using row pointer](#).
- It is not advisable to use this method if one needs to access some part or a single element of a data object.

7.3.5 Working with Data Objects

Now, lets have a look on various methods to work with data objects.

Creating Eye Matrix

Any square Data Object might need to be converted in eye matrix during many operations in matrix calculations. This can be quickly done using function **eye()** declared under *DataObject* class. Syntax for the function **eye()** is shown below.

dataObjectName.eye(noOfDimensions, dataType);

Return Type: void

To understand the use of this function, following is an exemplary code given. Let's have a look at it.

```
1 ito::DataObject *d2 = new ito::DataObject();
2 d2->eye(3, ito::tInt8);
3 std::cout << "3x3-eye matrix (int8)" << *d2 << std::endl;
4 delete d2;
5 d2 = NULL;
```

Here, the function **eye()** has been used with pointer variable *d2* to convert the data object *d2* into eye matrix.

Note: **eye()** function accepts only square matrices as inputs, otherwise it throws exception.

Creating Ones Matrix

Like Eye Matrix, Ones Matrix is equally important in matrix calculations. So we have developed a function called **ones()** under *DataObject* class to quickly convert any *data object* into *ones matrix*. Syntax for the function **ones()** is shown below.

dataObjectName.ones(dim 1,dim 2,...,dim n, dataType);

Return Type: void

```
1 ito::DataObject *d3 = new ito::DataObject();
2 d3->ones(2,3,4,ito::tFloat64);
3 std::cout << "2x3x4-ones matrix (double)" << *d3 << std::endl;
4 delete d3;
5 d3 = NULL;
```

Here, the function **ones()** has been used in Line #2 with pointer variable *d3* to convert the data object *d3* into ones matrix of dimension 2x3x4.

Creating Zeros Matrix

We have developed a function called **zeros()** under *DataObject* class to quickly convert any *data object* into *zero matrix*. Syntax for the function **zeros()** is shown below.

dataObjectName.zeros(dataType);

dataObjectName.zeros(const size_t size, dataType);

dataObjectName.zeros(const size_t sizeY, const size_t sizeX, dataType);

dataObjectName.zeros(const size_t sizeZ, const size_t sizeY, const size_t sizeX, dataType);

*dataObjectName.zeros(const unsigned char dimensions, const size_t *sizes, dataType);*

Return Type: RetVal

As **zeros()** function is overloaded, there are more than one syntax shown above.

Note: The **RetVal** class is used for handling error management and return relative codes. More description on this class can be seen in [RetVal - Reference](#).

Following code explains the usage of `zeros()` function.

```
1 ito::DataObject *dObjZeros = new ito::DataObject();
2 dObjZeros->zeros(2,3,4,ito::tFloat64);
3 std::cout << "3x4x5-zeros matrix (double)" << *dObjZeros << std::endl;
4 delete dObjZeros;
5 dObjZeros = NULL;
```

Here, line number 2 shows the way to use `zeros()` function to convert data object `dObjZeros` into a zero matrix.

Adjusting ROI of a Data Object

This section will teach you about how to adjust Region of Interest (ROI) in any data object.

The following example code shows the way to adjust ROI with `adjustROI()` method and to locate ROI with `locateROI()` method.

```
1 //adjusting ROI of 6x7 data object.
2 ito::DataObject d6(6,7,ito::tInt16);
3 int roiLocate[] = {0,0,0,0}; //Empty Array to locate ROI of 2 dimensional data object d6.
4 d6.adjustROI(-2,0,-1,-4);
5 d6.locateROI(roiLocate);
6 std::cout << d6 << std::endl;
7 for(int i =0; i<4; i++)
8 {
9     std::cout << roiLocate[i] << std::endl;
10 }
```

Here, line #4 shows the use of `adjustROI()` function where negative parameters indicate that the ROI is shrinking in particular dimension. More detailed description of `adjustROI()` and `locateROI()` methods can be seen under *plugin-DataObject-Ref* document.

One can also pass an array as a parameter to this `adjustROI()` function describing the offset details as shown in the following code.

```
1 int matLimits2d[] = {-2,0,-1,-4};
2 d6.adjustROI(2,matLimits2d);
3 std::cout << d6 << std::endl;
```

Here, `adjustROI()` function is called with 2 parameters and as can be seen in line #1, the array `matLimits2d[]` contains the same offset values as passed in `adjustROI()` method in the previous example.

One can use this example to adjust ROI of data objects more than 2 dimensions as well as shown in later examples below.

The following code shows such an example to modify ROI of a 3 dimensional data object.

```
1 //adjusting ROI of 6x7x8 data object.
2 ito::DataObject d7(6,7,8,ito::tFloat32);
3 int matLimits3d[] = {-1,-2,0,-2,-3,-1};
4 int lims3d[] = {0,0,0,0,0,0}; //Empty Array to locate ROI of 3 dimensional data object d7.
5 d7.adjustROI(3,matLimits3d);
6 d7.locateROI(lims3d);
7 std::cout << d7 << std::endl;
8 for(int i = 0; i<5; i++)
9 {
10     std::cout << lims3d[i] << std::endl;
11 }
```

Here, a 3 dimensional data object `d7` of dimensions 6x7x8 is declared in line #1 using the implementation #4 for data objects.

As can be seen in line #3, array `matLimits3d[]` of return type `int` contains required 6 offset values to adjust ROI of 3 dimensional data object `d7`. As shown in line #4, an empty array `lims3d[]` of `int` as return type is defined to

locate the ROI of data object d7 using **locateROI()** function. Line #7 will print the resultant data object after being adjusted by **adjustROI()** method in line #5 and the for loop in line #8-11 will print these located offset values of the resultant ROI of d7.

Setting and Getting Axis Units

In this section, you will study about assigning and retrieving the axis units at each dimension of a data object. **setAxisUnit()** method is used to assign a unit to a particular axis (dimension) of a data object and is declared under **DataObject** class. Syntax for this method is given below.

```
dataObjectName.setAxisUnit(const unsigned int axisNum, const std::string &unit);
```

Return Type: int

This function returns 1 if the axis does not exist.

getAxisUnit() method is used to retrieve a unit of a particular axis and is also declared under **DataObject** class. The syntax for this method is shown below.

```
dataObjectName.getAxisUnit(const int axisNum, bool &validOperation);
```

Return Type: std::string

This method returns Null if the axisNum is out of the range.

Following example code explains both of these methods.

```
1  ito::DataObject d8(6,7,ito::tFloat32);
2  bool vop1, vop2 = 0;
3  d8.setAxisUnit(0,"µm");
4  d8.setAxisUnit(1,"cm");
5  std::string AxisUnit1 =d8.getAxisUnit(0,vop1); //Getting axis unit of 1st dimension of data object
6  std::string AxisUnit2 =d8.getAxisUnit(1,vop2); //Getting axis unit of 2nd dimension of data object
7  std::cout << "Axis Unit of 1st Dimension:" << AxisUnit1 << std::endl;
8  std::cout << "Axis Unit of 2nd Dimension:" << AxisUnit2 << std::endl;
```

Here, a 2 dimensional data object *d8* of dimensions 6x7 is declared in line #1. Line #2 declares boolean variables *vop1* and *vop2* to pass as parameters in *getAxisUnit()* method later. Line #3 and #4 sets the units for dimensions 1 and 2 of data object *d8* respectively.

These assigned axis units can be retrieved by *getAxisUnit()* method as shown in line #5 and #6. Line #7 and #8 prints these retrieved axis units of data object *d8*.

Setting and Getting Axis Scale

In this section, we will learn about setting and getting the scale values of particular Axis (dimension) of a data object. This can be done using *setAxisScale()* and *getAxisScale()* functions as shown below. These both functions are declared under *DataObject* class. Syntax for the *setAxisScale()* function is shown below:

```
DataObjectName.setAxisScale(const unsigned int axisNum, const double scale);
```

Return Type: int

In the same way, syntax for the **getAxisScale()** function is shown below.

```
DataObjectName.getAxisScale(const int axisNum)
```

Return Type: double

Following exemplary code can explain the usage of these functions in a better way.

```
1  ito::DataObject d9(6,5,3,ito::tInt16);
2  d9.setAxisScale(0,5);
3  d9.setAxisScale(1,-0.5);
4  d9.setAxisScale(2,3.24);
5  double AxisScale1 =d9.getAxisScale(0);
```

```

6  double AxisScale2 =d9.GetAxisScale(1);
7  double AxisScale3 =d9.GetAxisScale(2);
8  std::cout << "Axis 1 Scale:" << AxisScale1 << "Axis 2 Scale:" << AxisScale2 << "Axis 3 Scale:" <<

```

As shown in line #2-4, **setAxisScale()** function is used to assign scales of 5, -0.5 and 3.24 on Axis 0, 1 and 2 respectively of data object d9. Line #5-7 explains how the axis scales can be retrieved using the function **getAxisScale()**. Line #8 prints down these retrieved axis values at the end of this code snippet.

7.3.6 Operations on Data Objects

In this section, we will learn some basic operations which can be performed using data objects.

Adjugate the Data Object

In many matrix calculations, there occurs a need to adjugate a matrix. Here, we also have a function **adj()** declared under *DataObject* class, which returns an adjugated matrix of the original data object. Syntax for this method is shown below.

DataObjectName.adj();

Return Type: ito::DataObject*

Following code snippet explains the use of **adj()** function.

```

1  ito::DataObject d10(6,5,3,ito::tComplex128);
2  d10.at<ito::complex128 >(0,1,2) = cv::saturate_cast<ito::complex128 >(ito::complex128 (23.2,0));
3  d10.at<ito::complex128 >(1,0,1) = cv::saturate_cast<ito::complex128 >(ito::complex128 (0,3));
4  d10.at<ito::complex128 >(2,2,1) = cv::saturate_cast<ito::complex128 >(ito::complex128 (1234,-23.3));
5  ito::DataObject adjugatedDataObj = d10.adj();
6  std::cout << "The Adjugated data object:" << std::endl;

```

In the code above, a 6x5x3 data object d10 of data type *ito::tComplex128* is created. Later on, in line #2-4, some complex values are assigned at data object elements (0,1,2), (1,0,1) and (2,2,1). In line #5, an adjugated matrix of data object d10 is created using **adj()** function and stored in new data object called **adjugatedDataObj**. Line #6 prints out this adjugated matrix.

Transpose the Data Object

Transposing a data object is also one of the very important techniques in matrix calculations. With the use of *trans()* function, we can achieve a transposed matrix of the original data object. This function is also declared under *DataObject* class. Syntax for this function is shown below.

DataObjectName.trans()

Return Type: ito::DataObject

Following code snippet explains the usage of **trans()** function.

```

1  ito::DataObject d11(2,2,ito::tInt16);
2  int temp=0;
3  for(int i=0;i<2;i++)
4  {
5      for(int j=0;j<2;j++)
6      {
7          temp++;
8          d11.at<ito::int16>(i,j)= cv::saturate_cast<ito::int16>(temp);
9      }
10 }
11 ito::DataObject transDObj = d11.trans();
12 std::cout<< "The Transposed data object:" << std::endl;

```

In the code above, a 2x2 data object d11 of type `ito::tInt16` is created. This data object d11 is initiated by assigning different values to each element. In line #11, `trans()` function is used to transpose this data object d11 and resulted transposed data object is stored in new data object transDobj, which gets printed out in line #12.

Note: Transposing the data object also transposes the axis related informations.

Basic Operators with Data Objects

(Same syntax can be used for other operators like '+', '-', '=', '+=', '=-', div, cross multiplication (!=), << (shift left), >> (shift right)) For the sake of simplicity, some arithmetic operators are overloaded to work upon data objects easily. In this section, such operators to work upon data objects are discussed in details with example codes and syntaxes. Let us start with basic Add “+” operator. Following is one example shown to add two data objects with “+” operator.

```
1 ito::DataObject d12(2,2,ito::tInt16);
2 ito::DataObject d13(2,2,ito::tInt16);
3 ito::DataObject d14(2,2,ito::tInt16);
4 d12= cv::saturate_cast<ito::int16>(2);
5 d13= cv::saturate_cast<ito::int16>(2);
6 d14= d12 + d13;
7 std::cout << "Addition of two matrix is:" << d14 << std::endl;
```

Here, two data objects d12 and d13 are added elementwise and the resultant data object is stored in d14. In the same way many other arithmetic, compare, bitwise operators can work with data objects. Following is the explanation of each possible operator to work with data object.

Add Operator ‘+’

If mat1, mat2 and mat3 are data objects of same size and same type, then “+” operator can be used to perform elementwise addition in the following way.

```
mat3 = mat1 + mat2;
```

“+” operator performs elementwise addition on mat1 and mat2 and returns a resultant data object. This operator returns the resultant data object of same size and type which needs to be stored in some other data object of same size and type (mat3) as shown in the statement above.

Add and Assign Operator ‘+=’

If mat1 and mat2 are data objects of same size and same type, then “+=” operator can be used to perform elementwise add and assign operation in the following way.

```
mat2 += mat1;
```

“+=” operator performs elementwise addition operation on mat1 and mat2 and stores the resultant data object leftside of this it.

Subtract Operator ‘-’

If mat1, mat2 and mat3 are data objects of same size and type, then “-” operator can be used to perform elementwise subtraction between these data objects in the following way.

```
mat3 = mat1 - mat2;
```

In the statement above, mat2 is subtracted from mat1 elementwise and the resultant data object is stored in mat3.

Subtract and Assign Operator ‘-=’

If `mat1` and `mat2` are data objects of same size and type, then “-=” operator can be used to perform elementwise subtraction between these data objects in the following way.

```
mat2 -= mat1;
```

In the statement above, `mat1` is subtracted from `mat2` elementwise and the resultant data object is stored back to `mat2`.

Multiplication with Constant Factor ‘*’

Operator “*” is overloaded to perform multiplication of a data object with constant factor in the following way.

```
mat2 = mat1 * x;
```

Here, the data type of `mat2` must proper enough to store the correct resultant data object obtained from multiplication of each element of `mat1` with constant factor “`x`”.

Multiplication with Constant Factor and Assign ‘*=’

One can multiply all elements of a data object with a constant factor and assign the result back to the same data object as shown in the following statement.

```
mat1 *= x;
```

Here each element of the data object **mat1** gets multiplied with a constant factor “`x`” and the result is stored back into data object `mat1`.

Cross Multiplication Operator ‘**’

One can use the operator “**” to perform a cross multiplication between two data objects with adequate dimensions in the following way.

```
crossMul_mat = mat1 ** mat2;
```

In the statement above, operator “**” is used to perform cross multiplication between data objects `mat1` and `mat2`. If sizes of `mat1` and `mat2` are `axb` and `bxc` respectively, then the size of `crossMul_mat` should be declared `axc` according to the principle of cross multiplication of matrices.

Note: This operator is defined only for “float32” and “float64” datatypes.

Dot Multiplication Operator ‘mul’

With the use of “mul” operator, one can perform dot product (elementwise multiplication) between two data objects in the following way.

```
mul_mat=mat1.mul(mat2);
```

Here, “mul” operator takes two data objects `mat1` and `mat2` of same type and same size as arguments and perform dot product between them. The result of this dot product is saved in data object of same size `mul_mat`. Here, the size of `mul_mat` should be big enough to store correct result of dot product.

Division Operator 'div'

This operator is used to perform elementwise division between two data objects in the following way.

```
div_mat = mat1.div(mat2);
```

Here, data object **mat1** is divided from data object **mat2** elementwise using “**div**” operator and the result is stored in data object **div_mat**. Sizes and data types of mat1, mat2 and mat2 should be same.

Note: “**div**” operator can not be used to calculate inverse matrix.

Compare Operators Compare operations are frequently needed while working with data objects. Keeping this in mind, we have developed some overloaded compare operators as well which we will discuss in this section.

== (Equal to) (Elementwise operator)

This operator compares the element values of two different data objects on respective positions and returns a resultant data object with **0s** and **1s** according to the comparison result. If two values on the respective positions match in two data objects, then a **1** is returned, otherwise a **0** is returned for that position in the resultant data object. The way to use this **== (Equal to)** operator is shown as follows.

```
mat3 = mat1 == mat2;
```

Here, a comparison between data objects mat1 and mat2 is done with **==** operator and the result is stored in the data object mat3. As a result, mat3 contains ‘1’ at the positions where the values in mat1 and mat2 are equal, otherwise ‘0’ where the values are unequal. Sizes of mat1, mat2 and mat3 should be equal.

!= (unequal to) (Elementwise operator)

This operator compares the element values of two different data objects on respective positions and returns a resultant data object with **0s** and **1s** according to the comparison result. If two values on the respective positions differ in two data objects, then a **1** is returned, otherwise a **0** is returned for that position in the resultant data object. The way to use this **!= (unequal to)** operator is shown as follows.

```
mat3 = mat1 != mat2;
```

Here, a comparison between data objects mat1 and mat2 is done with **!=** operator and the result is stored in the data object mat3. As a result, mat3 contains ‘1’ at the positions where the values in mat1 and mat2 are unequal, otherwise ‘0’ where the values are equal. Sizes of mat1, mat2 and mat3 should be equal.

< (lower than) (Elementwise operator)

This operator performs “lower than” comparison operation between the element values of two data objects. The way to perform this comparison is shown in the following statement.

```
mat3 = mat1 < mat2;
```

In the statement above, elements of **mat1** are compared with elements of mat2 for **lower than** operation with use of **<** operator. The resultant data object **mat3** contains ‘1’ at the positions where the element values in data object mat1 are lower than those of mat2 at respective positions, otherwise it contains ‘0’ where this condition doesn’t satisfy. Sizes of all three data objects mat1, mat2 and mat3 should be the same.

<= (lower or equal to) (Elementwise operator)

This operator compares two data objects and helps us to find out which elements in one data object are “**lower or equal to**” the respective elements in the other data object in the following way.

```
mat3 = mat1 <= mat2;
```

Here, “<=” operator compares data object **mat1** with **mat2** and generates a resultant data object **mat3** such as, **mat3** contains **1s** at the positions where elements in **mat1** are lower or equal to the respective elements in **mat2**, otherwise contains **0s** where the condition doesn’t satisfy. Sizes of **mat1**, **mat2** and **mat3** should be equal.

> (bigger than) (Elementwise operator)

With the use of this over loaded operator, one can easily find out at which positions the values of elements in one data object are **bigger than** the values of the other data object in the following way.

```
mat3 = mat1 > mat2;
```

Here in the above statement, comparison operation between **mat1** and **mat2** is done with “>” operator and the final result is stored in data object **mat3**. As a result, **mat3** contains **1s** at the positions where the element values in **mat1** are bigger than the element values on the respective positions in **mat2**. Data sizes of **mat1**, **mat2** and **mat3** should be the same for the successful comparison with this operator.

>= (bigger or equal to) (Elementwise operator)

This operator is used to find out if the element values of one data objects are **bigger or equal to** the element values of the other data object.

Syntax to use this operator with data objects is shown below.

```
mat3 = mat1 >= mat2;
```

As shown above, element values of **mat1** are compared with the element values of **mat2** and the result is stored in **mat3**. As a result, **mat3** contains **1s** at the positions where the element values in **mat1** are **bigger or equal to** the element values in **mat2**, otherwise contains **0s** at the positions where the condition doesn’t satisfy. Sizes of **mat1**, **mat2** and **mat3** should be equal.

Shift Operators Shift operators play a significant role during some arithmetic operations (i.e. division/multiplication by 2), bit level calculations, etc. With the over loading of some shift operators to let them be used for data objects, many matrix calculations can be made much easier. We will go through such over loaded shift operators in this section.

<< (shift left)

<< or in words “**Shift Left**” operator is overloaded to work with data objects. It shifts the element values of the data object at the bit level by a required amount on left.

Syntax to use this operator with data objects is shown below.

```
mat2 = mat1 << 'x';
```

Here ‘**x**’ is the Shiftbits number. The values of the elements in **mat1** are shifted left by ‘**x**’ number of times and the resultant data object is stored in **mat2d**. Data type of the Shiftbits ‘**x**’ is const unsigned int.

<<= (shift left and assign)

This operator is used to **shift left** the element values on a data object at a bit level and **assign back** these values to the same data object.

Syntax to use this operator with data objects is shown below.

```
mat1 <<= 'x';
```

Here ‘**x**’ is the Shiftbits number. The values of the elements in **mat1** are shifted left by ‘**x**’ number of times and assigned to the same data object. Data type of the Shiftbits ‘**x**’ is const unsigned int.

>> (shift right)

This operator is used to **shift right** the elements of a data object at a bit level. As a result, this operation returns a data object with shifted element values which needs to be stored in some data object with same size and type.

Syntax to use this operator with data objects is shown below.

```
mat2 = mat1 >> 'x';
```

Here 'x' is the Shiftbits number. The values of the elements in mat1 are shifted right by 'x' number of times and the resultant data object is stored in mat2. Data type of the Shiftbits 'x' is const unsigned int.

>>= (shift right and assign)

This operator is used to **shift right** the elements of a data object at a bit level and **assign back** the shifted values to the same data object.

Syntax to use this operator with data objects is shown below.

```
mat1 >>= 'x';
```

Here 'x' is the Shiftbits number. The values of the elements in mat1 are shifted right by 'x' number of times and assigned to the same data object. Data type of the Shiftbits 'x' is const unsigned int.

Bitwise Operations Bitwise operators play important role in logical operations such as AND, OR and NOT. In this section, some of such over loaded logical operators for data objects are described.

'&' (Bitwise AND) Operator

This operator performs a **bitwise AND** operation between the elements of two data objects.

Syntax to use this operator with data objects is shown below.

```
d3 = d1 & d2 ;
```

As shown in the statement above, with the use of '&' operator, **elementwise AND** operation occurs at the bit level between the elements on the same locations of d1 and d2 and the resultant values are stored at the respective locations of d3.

Note: Sizes and data types of data objects d1, d2 and d3 should be same, otherwise exception is thrown.

"|" (Bitwise OR) Operator

This operator performs a **bitwise OR** operation between the elements of two data objects.

Syntax to use this operator with data objects is shown below.

```
d3 = d1 | d2 ;
```

With the use of '|' operator, **elementwise OR** operation occurs at the bit level between the elements on the same locations of d1 and d2 and the resultant values are stored at the respective locations of d3.

Note: Sizes and data types of data objects d1, d2 and d3 should be same, otherwise exception is thrown.

“^” (Bitwise NOT) Operator

This operator performs a **bitwise NOT** operation between the elements of two data objects.

Syntax to use this operator with data objects is shown below.

d3 = d1 ^ d2 ;

With the use of ‘^’ operator, **elementwise NOT** operation at the bit level occurs between the elements on the same locations of d1 and d2 and the resultant values are stored at the respective locations of d3.

Note: Sizes and data types of data objects d1, d2 and d3 should be same, otherwise exception is thrown.

“&=” (Bitwise AND and Assign) Operator

This operator performs a **bitwise AND** operation between the elements of two data objects and **assigns back** the result into the first data object.

Syntax to use this operator with data objects is given below.

d2 &= d1;

With the use of ‘&=’ operator, **elementwise AND** operation occurs at bit level between the elements on the same locations of d1 and d2 and the resultant values are assigned at the respective locations of d1 back.

Note: Sizes and data types of data objects d1 and d2 should be same, otherwise exception is thrown.

“|=” (Bitwise OR and Assign) Operator

This operator performs a **bitwise OR** operation between the elements of two data objects and **assigns back** the result into the first data objects.

Syntax to use this operator with data objects is given below.

d2 |= d1;

With the use of ‘|=’ operator, **elementwise OR** operation at bit level occurs between the elements on the same locations of d1 and d2 and the resultant values are assigned at the respective locations of d1 again.

Note: Sizes and data types of data objects d1 and d2 should be same, otherwise exception is thrown.

“^=” (Bitwise NOT and Assign) Operator

This operator performs a **bitwise NOT** operation between the elements of two data objects and **assign back** the result into the first data object.

Syntax to use this operator with data objects is given below.

d2 ^= d1;

With the use of ‘^=’ operator, **elementwise NOT** operation at bit level occurs between the elements on the same locations of d1 and d2 and the resultant values are assigned at the respective locations of d1 again.

Note: Sizes and data types of data objects d1 and d2 should be same, otherwise exception is thrown.

Combination of Different Operators Different operators (explained above) can be used in different possible combinations with data objects for speedy calculations.

Some of the examples of such combinations are shown below.

```

1      d4 = d1 + d2 - d3;
2      d3 = (d1 | d2).div((d1 << 1) + d1);
3      d4= (d1 & d2).mul(d3 >> 2);

```

With the close observation of the example statements above, one can get an idea on how to use these operators in different combinations to get the desired operation done.

Note: For a full reference of the class **DataObject** see *plugin-DataObject-Ref*.

RetVal - Reference

class ito::RetVal

Class for error value management.

The *RetVal* class is used for handling return codes. All classes should use this class. In case an error occurs, only the first error is stored and will not be overridden by potentially subsequent occurring errors.

Public Functions

RetVal(tRetVal retValue = retOk)

RetVal(int retValue)

RetVal(ito::tRetVal retValue, int retCode, const char * pRetMessage)

constructor with retValue, retCode and errorMessage

Parameters

- *retValue* - return Value constant, for possible values see tRetVal
- *retCode* - user definable return code
- *pRetMessage* - error message to be passed, string is copied Makes a deep copy of *RetVal*, i.e. a copy of the error message

RetVal(const *RetVal* & copyConstr)

Deep copy constructor Makes a deep copy of *RetVal*, i.e. a copy of the error message

~RetVal()

Destructor, in case frees errorMessage string

RetVal & **operator**=(const *RetVal* rhs)

assignment operator, copies values of rhs to current *RetVal*. Before copying current errorMessage is freed

RetVal & **operator**+=(const *RetVal* rhs)

Concatenation of *RetVal* “Adds” RetVals, i.e. returns the most serious error. In case of equally serious errors the first is retained

RetVal **operator**+(const *RetVal* rhs)

Concatenation of *RetVal* See operator *RetVal::operator*+=

char **operator**==(const *RetVal* rhs)

equality operator compares retValue with with retValue of rhs *RetVal*. For possible constant values see tRetVal

char **operator**!=(const *RetVal* rhs)

unequality operator compares retValue with with retValue of rhs *RetVal*. For possible constant values see tRetVal

char **operator**==(const tRetVal rhs)

equality operator compares retValue with tRetVal constant. For possible constant values see tRetVal

char **operator**!=(const tRetVal rhs)

unequality operator compares retValue with tRetVal constant. For possible constant values see tRetVal

int **containsWarning**()

checks if any warning has occurred in this return value (true), else (false)

int **containsError**()

checks if any error has occurred in this return value (true), else (false)

int **containsWarningOrError**()

checks if any warning or error has occurred in this return value (true), else (false)

char * **errorMessage**()

int **errorCode**()

void **appendRetMessage**(const char * addRetMessage)

Public Static Functions

RetVal **format**(ito::tRetValue retValue, int retCode, const char * pRetMessage, ...)

ito::DataObject *ito::DataObject*

7.3.7 Parameter-Container class of itom

Introduction

The base idea behind a container-class for parameters of varying types is to pass these parameters to methods without the need of extensive templating. Of course, both **Python** and **Qt** provide such classes, namely **PyObject** and **QVariant**. Nevertheless, **itom** provides an own, low-level container class, which is not dependent on any 3rd party library. This container only provides support for some specific types, that are widely used within **itom**. This parameter container is mainly used for the whole communication process between all types of plugins and **itom**. Examples for their use are:

- Parameters of each plugin, which can be read by *getParam* or set by *setParam*. Additionally these parameters of the plugin can be obtained by the python command `getParamList()`.
- Mandatory and optional parameters for the constructor of plugin instances.
- Parameter transfer between configuration dialog, docking widget toolbox and plugin itself.

The whole implementation of the container-class is found in the files *sharedStructures.h* and *sharedStructures.cpp*, both lying in the folder *common*, where you can also find the file *addInInterface.h*.

The possible types, covered by this container are:

- **char** (Type 1)
- **integer** (Type 1)
- **double** (Type 1)
- **char-array** (Type 1)
- **integer-array** (Type 1)
- **double-array** (Type 1)
- **string** (zero-terminated, Type 1)
- **DataObject** (pointer-only, Type 2)
- **PointCloud** (pointer-only, Type 2)

- **PolygonMesh** (pointer-only, Type 2)
- **HWRef** (pointer-only, points to an instance of `AddInBase`, Type 2)

Note: *Type 1* means, that these types are internally copied when calling a constructor, copy constructor or assignment operator of the parameter, where for parameters of *Type 2* only the pointer to the value itself is internally stored in the parameter, hence, only this pointer is copied at the above mentioned methods. The reason is, that a quick passing of the parameter is provided, on the other side, parameters of *Type 2* need some further attention concerning thread-safety and/or creation and deletion responsibility.

***ParamBase* and *Param* and the *Meta*-classes**

There are different classes, defined in *sharedStructures.h* which can be used for the parameter container:

- class **ParamBase** represents a pure multi-type container, which only contains the name of the parameter and its value. This is the quickest parameter container implementation and should always be used, if only the parameter should be passed and no further information are needed. In any other cases use an implementation of the derived class *Param*.
- class **Param** is derived from *ParamBase* and additionally contains a description string for the value (optional) and some further meta-information (optional) in form of an internal pointer, that points to an instance of class *MetaParam* or one of its derived classes.
- class **MetaParam** is the base class for all parameter type-dependent meta information classes, like *IntMeta*, *DoubleMeta*, *StringMeta*... The idea is to add some restrictions about value ranges, allowed values... to parameters if this is needed or available. Please consider, that the meta information class is not internally checked in the parameter classes, hence, you can assign every value you want to. However the programmer, that is using a parameter, can access the *MetaParam*-instance and program its own validator for the parameter or use one of the pre-defined methods.
- class **CharMeta**, **IntMeta**, **DoubleMeta** are meta information classes derived from *MetaParam* which contain a minimum and maximum value for the parameter. Parameters of array-types may also contain an instance of one of these classes in order to describe the allowed range of every element of the array.
- class **StringMeta** provides further information for the parameter of type *String* such that you restrict the string to certain values, which also can be evaluated in the sens of a regular expression or wildcard expression.
- class **HWMeta** provides restrictive information for a parameter of type *HWRef*

For more information about the meta information classes see [Parameters - Meta Information](#).

Usage and differences of the classes *ParamBase* and *Param*

Variables of class **Param** are used whenever you explicitly want to add further information to your parameter. Examples might be:

- Vectors of mandatory and optional parameters, used as template for creating an instance of a plugin.
- Vectors of mandatory and optional parameters, used as template for creating a widget defined in a plugin of type *AddInAlgo*.
- Vectors of mandatory, optional and out parameters, used as template for calling a filter.
- Plugin-internal parameters, stored in the Map *m_params*.
- *getParam*-method of plugins, which usually return one specific value of map *m_params*. Here an instance of class *Param* is returned and not *ParamBase* such that advanced information about the value can be presented.
- Vector of out-parameters of filters.

Variables of class *ParamBase* are used when you only need to transfer the parameter itself:

- Method *setParam* of plugins. The validation of the given value is done with respect to its corresponding value in map *m_params*.
- Vector of mandatory and optional parameters used for calling the constructor (method *init*) of plugins.
- Vector of mandatory, optional and out parameters used for calling filters in plugins.
- Vector of mandatory and optional parameters used for calling widgets, defined in plugins of type *AddInAlgo*.

In the case of the described mandatory and optional parameter vectors, **itom** is requesting the template version (class *Param*) from the plugin and has enough information, in order to check the user input (done in GUI or by python) with respect to the template. Finally a vector of type *ParamBase* is created, where all the default values, given by the templates, are overwritten by the user input. Then the filter, plugin constructor or widget constructor is called with the version of *ParamBase*.

Types and flags

The type as well as additional flags of each parameter is defined by an OR combination of values, contained in the enumeration **ito::ParamBase::Type**. The last 16 bit (bit 1-16) of this enumeration are reserved for type information, the first 16 bits may contain flags.

They can be separated using an AND-operation with the masks **ito::paramFlagMask** or **ito::paramTypeMask**:

```
int typesAndFlags
int types = typesAndFlags & ito::paramTypeMask
int flags = typesAndFlags & ito::paramFlagMask
```

The following (high-level) types are available:

```
enum Type {
    ...
    //type (bit 1-16)
    Pointer      = 0x000001, //do not use directly
    Char         = 0x000002, //Character-Parameter
    Int          = 0x000004, //Integer-Parameter
    Double       = 0x000008, //Double-Parameter
    String       = 0x000020 | Pointer, //zero-terminated String-Parameter
    HWRef        = 0x000040 | Pointer | NoAutosave, //pointer to plugin-instance
    DObjPtr      = 0x000010 | Pointer | NoAutosave, //pointer to dataObject
    CharArray    = Char      | Pointer, //array of characters
    IntArray     = Int       | Pointer, //array of integers
    DoubleArray  = Double    | Pointer, //array of doubles
    PointCloudPtr = 0x000080 | Pointer | NoAutosave, //pointer to point cloud
    PointPtr     = 0x000100 | Pointer | NoAutosave, //pointer to point
    PolygonMeshPtr = 0x000200 | Pointer | NoAutosave //pointer to polygon mesh
    ...
};
```

Note: All pointer-based types have the **NoAutosave**-flag, since a pointer can not be saved to harddrive. The arrays of **CharArray**, **IntArray** and **DoubleArray** are internally copied (e.g. in a copy-constructor), therefore only use them for smaller arrays and not for matrices with millions of entries. This might be an inefficient structure though.

The following flags are implemented in the **Type**-enumeration:

```
enum Type {
    NoAutosave,
    Readonly,
    In,
    Out
    ...
};
```

The behaviour of the **NoAutosave**-flag can be read in see [Automatic loading and saving of plugin parameters](#). The **readonly**-flag marks this parameter to be readonly. Please consider, that this flag is not evaluated in the classes **Param** or **ParamBase**, but the programmer has access to this flag must implement the necessary behaviour. The flags **In** and **Out** or their combination are important for the declaration of the default parameters for plugins or filter-calls. If none of them is set, the flag **In** is automatically set. **In** indicates, that the parameter is handled like an input-variable only, hence, the filter or plugin's init method will not change the value of this parameter. A variable of type **InOut** passes a value and the value might be changed within a filter call. This is a suitable form to pass a dataObject whose content and size might be changed by the filter. Parameters with flag **Out** only are only accepted in the parameter vector which is the default for the output-variables of a filter... It is only allowed to mark parameter of type **Char**, **Int**, **Double**, **String**, **CharArray**, **IntArray** or **DoubleArray** as **Out**- parameters.

Class *ParamBase*

The class `ParamBase` consists of the following main elements or member variables, which however are only accessible by corresponding getter- or setter-methods:

m_type

This variable contains an OR combination of the data type, covered by the parameter container as well as some additional flags (read-only, auto-save). Read the section [Types and flags](#) for more information about the type.

The type part of this member is obtained by `getType()`, the flags can be obtained by `getFlags()`.

char ***m_pName**

This char-pointer contains the name of the parameter. This name is for example used for accessing the parameter in the python's `setParam` or `getParam` method and usually you can also use this name as keyword in a python argument list of appropriate method calls.

Access the name of a parameter by using `getName()`.

values

There are three further member variables which are used in order to store the variable content of the parameter container. Reading and writing these values is only done by the constructor or the methods `getVal<_Tp>` and `setVal<_Tp>`.

Typical creations for parameters of class **ParamBase** are:

```
//empty parameter (name: NULL, type: 0)
ParamBase p1;

//creating an integer-parameter, flag: In, value: 2
ParamBase p2("IntParam", ito::ParamBase::Int | ito::ParamBase::In, 2);

//creating a double-parameter, flag: Readonly, value: -4.0
ParamBase p3("Name", ito::ParamBase::Double | ito::ParamBase::Readonly, -4.0);

//creating a string-parameter
ParamBase p4("Name", ito::ParamBase::String, "default text");

//creating an integer-array parameter
int size = 5;
int* a = new int[size];
//.. fill a with valid values
ParamBase p5("Array", ito::ParamBase::IntArray, size, a);
//a is copied, therefore delete it now
delete[] a;
a = NULL;

//passing a dataObject pointer as parameter
ito::DataObject *dObj = new ito::DataObject(...);
ParamBase p6("param", ito::ParamBase::DObjPtr, dObj);
//be careful: p6 only holds a pointer to dObj, therefore you can only delete it
// if p6 does not access it any more.
```

```
//passing a pointer to another actuator- or dataIO-instance to a parameter
ito::AddInActuator *aia = ...;
ParamBase p7("motor", ito::ParamBase::HWRef, aia);
//like with the dataObject. Be careful and make sure, that the pointer 'aia' remains
//accessible during the lifetime of p7.
```

The parameter **p1** has no suitable type or value right now. However, you can assign another parameter to **p1** by using the assignment operator:

```
p1 = ParamBase("newVal", ito::ParamBase::Char, 128)
```

If you have an array-parameter, you can access one single index of this array, which is then returned as new instance of **ParamBase**. If the index is out of range or the parameter is no array-type, an empty instance of **ParamBase** is returned:

```
//use p5 from the example above
ParamBase p5_0 = p5[0] //is a valid parameter of type Int
ParamBase p5_5 = p5[5] //error. empty ParamBase since index exceeded the maximum size.
```

Reading values from the parameter is done by the method **getVal**. This method must be called with a template parameter, that corresponds to the original data type, which is covered by the parameter.

```
//This example is based on the constructed params above
int p2_val = p2.getVal<int>();

double p3_val = p3.getVal<double>();

//the following examples return the internal pointer to the string or arrays.
//This pointer is no copy, therefore you are not allowed to delete the pointer.
char* p4_val = p4.getVal<char*>();

int* p5_val = p5.getVal<int*>();
//you can access the elements of p5 by
int temp = p5_val[0];
temp = p5_val[4];
//p5_val[5] is not allowed, since it exceeds the number of elements of this array

//in order to get the number of values in the parameter, use the following
//implementation
int length = 0;
p5_val = p5.getVal<int*>(length);
//now length is equal to 5.

//pointer-parameters are obtained by using the right template
//parameter of the getVal method. The internal pointer of the
//parameter is then casted to the template type.
ito::DataObject *dObj = p6.getVal<ito::DataObject*>();

ito::AddInActuator *aia = p7.getVal<ito::AddInActuator*>();

//If you are sure that the parameter contains at least any plugin, however you have no idea
//whether it is an acutator or an instance of dataIO, you could at first get the
//base instance to ito::AddInBase and then try to safely cast it to your requested type:
ito::AddInBase *aib = p7.getVal<ito::AddInBase*>();
ito::AddInActuator *aia = qobject_cast<ito::AddInActuator*>(aib);
//aia is NULL, if the cast failed.
```

If the given template parameter does not fit to the corresponding parameter type, the value of the parameter will be casted to the given template type. If this is not possible an exception is raised. The exception is of type **std::logic_error**.

Settings values to the parameter can be analogously done by the method **setVal**. This also is a template based method. The following code snippets show examples how to change values of the previously constructed param-

eters **p1** to **p7**:

```
ito::RetVal retVal;
retVal += p2.setVal<int>(5); //retVal remains retOk
retVal += p3.setVal<double>(-3.7); //retVal remains retOk

//p4 is a string-type. New values assigned to p4 (here: "new value")
// are internally copied.
retVal += p4.setVal<char*>("new value"); //retVal remains retOk

//for array-types, you can only assign the whole array and not change any
//elements. For changing values, use getVal<_Tp>(...) in order to obtain the pointer and
//change the values directly.
int values[] = {1,2,3,4,5};
retVal += p5.setVal<int*>(values,5); //always provide the length of the array
//again, the setVal-method above internally copies the array and you can destroy the
//source.
int length = p5.getLen(); //length of array

ito::DataObject dObj;
retVal += p6.setVal<void*>( (void*)&dObj );
//again remember, that p6 requires, that dObj remains accessible during the lifetime of p6.
```

Note: The method **setVal** will never change the assigned type of the parameter. If the new value can not be converted into the internal type of the parameter, **setVal** will return with a **RetVal**, that contains errors.

If you only want to copy the content of one parameter of type **ParamBase** to your parameter, then you can use the method **copyValueFrom**, which requires another instance of **ParamBase** or **Param** (since this is derived from **ParamBase**). The method returns an error if the parameters are not compatible:

For a full reference to all member function of class **ParamBase**, see [ParamBase - Reference](#).

Class **Param**

The class **Param** is derived from `:cpp:class'ParamBase'`. Therefore it has all features of **ParamBase** including two additional member variables:

ParamMeta* m_pMeta

This is a pointer to a struct containing type-dependent meta information about this plugin. This pointer may also be NULL, if no meta information is provided. The meta-information struct is always owned by the parameter and deeply copied when calling for instance a copy constructor. For more information see [Parameters - Meta Information](#).

Access to the meta information struct is given by

```
Param p;
ParamMeta* meta = p.getMeta();
//in this case meta is NULL, since no meta information has been set to 'p'.

//now we create a integer-variable with a min and max value
Param p2("var1",ParamBase::Int,2);
IntMeta meta2(-2,2);

//now we set the meta information of p2 to meta2. Since the ownership of meta2 should not
//be taken by p2, the second argument is false. Then, p2 makes a copy of meta2.
p2.setMeta(&meta2,false);

meta = p.getMeta();
//meta is now a pointer to a structure of type ParamMeta. It can be casted to IntMeta.
```

char* m_pInfo

This is the description string of the parameter. If no description is indicated, this pointer is

NULL, else it is a zero-terminated string, which is also copied, when the parameter is called using a copy constructor or assigned to another parameter.

The description can be obtained by

```
Param p("name", ParamBase::String, "content", "information")
const char* descr = p.getInfo(); //descr is 'information'
```

Note: Do not delete the char-pointer returned by `getInfo`, since this is only a reference to the internal description string of the parameter.

The description string is changed by

```
p.setInfo("new information")
```

The full reference of class `Param` is available in *Param - Reference*.

In the following, examples about how to create parameters and meta information of different types are shown:

- **Integer-Type (Type: Int)**

This is one fixed-point number in the integer-range.

```
//integer value between 0 and 10, default: 5
ito::Param param("intNumber", ito::ParamBase::Int, 0, 10, 5, "description");
//this default constructor automatically creates an internal meta-information struct
//of class IntMeta.

// or (here param becomes owner of IntMeta-instance)
ito::Param param("intNumber", ParamBase::Int, 5, new IntMeta(0,10), "description");

// or (integer-variable without meta information)
ito::Param param("intNumber", ParamBase::Int, 5, NULL, "description");
param.setMeta(new IntMeta(0,10), true); //take ownership of IntMeta-instance

int value = param.getVal<int>();           //returns 5
ito::RetVal retValue = param.setVal<int>(6); //returns ito::retOk
bool numeric = param.isNumeric()           //returns true, since integer is a numeric

// accessing the min-max-value is obtained by getting the IntMeta-struct
IntMeta *meta = dynamic_cast<IntMeta*>(param.getMeta());
if(meta) //meta is only valid, if it has been assigned.
{
    int min = meta->getMin()    //returns 0
    int max = meta->getMax()    //returns 10
}
int len = param.getLen()       //returns 1
```

- **Double-Type (Type: Double)**

This is one floating-point number in the double-range.

```
//integer value between 0.0 and 10.0, default: 5.0
ito::Param param("doubleNumber", ito::ParamBase::Double, 0.0, 10.0, 5.0, "description");
//this default constructor automatically creates an internal meta-information struct
//of class DoubleMeta.

// or (here param becomes owner of DoubleMeta-instance)
ito::Param param("doubleNumber", ParamBase::Double, 5.0, DoubleMeta::all(), "description");
// the command DoubleMeta::all() creates a new instance of DoubleMeta, where the boundaries
// are the minimum and maximum possible value of the double-range.

// or (double-variable without meta information)
ito::Param param("doubleNumber", ParamBase::Double, 5.0, NULL, "description");
param.setMeta(new DoubleMeta(0.0,10.0), true); //take ownership of DoubleMeta-instance
```

```

double value = param.getVal<double>();           //returns 5.0
ito::RetVal retValue = param.setVal<double>(6.0); //returns ito::retOk
bool numeric = param.isNumeric()                 //returns true, since integer is a number

// accessing the min-max-value is obtained by getting the DoubleMeta-struct
DoubleMeta *meta = dynamic_cast<DoubleMeta*>(param.getMeta());
if(meta) //meta is only valid, if it has been assigned.
{
    double min = meta->getMin();
    double max = meta->getMax();
}
int len = param.getLen()                        //returns 1

```

• String-Type (Type: String)

This is one zero-terminated String.

```

ito::Param param("string", ito::ParamBase::String, "", "description");

//if you want to provide a string-meta information, you must do it in the following separate
ito::StringMeta *meta = new ito::StringMeta(ito::StringMeta::String);
meta->addItem("yes");
meta->addItem("no");
//the meta information indicates, that only the exact matches of both values "yes" or "no"
//might be accepted by this parameter.
param.setMeta(meta, true); //takes ownership of meta

char* value = param.getVal<char*>();           //returns the pointer to the internally saved string
ito::RetVal retValue = param.setVal<char*>("test"); //should return ito::retOk, String is not numeric
bool numeric = param.isNumeric()               //returns false
int len = param.getLen()                       //0 if empty string, else length of string

```

Note: Please do not delete the pointer to the internally saved string, obtained by `getVal<char*>()`!

• Array of char values (Type: CharArray)

This is an array of character values. Consider that you should use the constructor where you can give the length of the array, else an error is returned.

```

char ptr = [0,56,127,-10,-20];
ito::Param param("array", ito::ParamBase::CharArray, 5, &ptr, "description");
//you can add a meta-information struct of class CharMeta to that char-array (if desired)

char* value = param.getVal<char*>();           //returns the pointer to the first element of the array
ito::RetVal retValue = param.setVal<char*>(ptr); //should return ito::retOk
bool numeric = param.isNumeric()               //returns false (even it is an array of numeric values)
int len = param.getLen()                       //5
ito::Param param0 = param[0];                  //returns a char-parameter with value 0

```

• Array of integer values (Type: IntArray)

This is an array of integer values. Consider that you should use the constructor where you can give the length of the array, else an error is returned.

```

int ptr = [1,2,3,4,5];
ito::Param param("array", ito::ParamBase::IntArray, 5, &ptr, "description");
//you can add a meta-information struct of class IntMeta to that integer array (if desired)

int* value = param.getVal<int*>();             //returns the pointer to the first element of the array
ito::RetVal retValue = param.setVal<int*>(ptr); //should return ito::retOk
bool numeric = param.isNumeric()               //returns false (even it is an array of numeric values)

```

```
int len = param.getLen() //5
ito::ParamBase param2 = param[1] //returns integer-parameter (casted to ParamBase)
```

- **Array of integer values (Type: DoubleArray)**

This is an array of double values. Consider that you should use the constructor where you can give the length of the array, else an error is returned.

```
double ptr = [1.2,2.3,3.4,4.1,5.2];
ito::Param param("array", ito::ParamBase::DoubleArray, 5, &ptr, "description");

double* value = param.getVal<double*>(); //returns the pointer to the first element of
ito::RetVal retValue = param.setVal<double*>(ptr); //should return ito::retOk
bool numeric = param.isNumeric() //returns false (even it is an array of numeric v
int len = param.getLen() //5
```

- **Reference to any initialized instance of dataIO or actuator (Type: HWRef)**

This is the reference to any other initialized instance of dataIO or actuator. The called method should check whether the instance has the necessary properties or type. Consider, that the flag *NoAutosave* is always set for that type. If such a parameter is passed to the *init*-method of a plugin, the reference of the passed plugin is automatically increased (marks the plugin as being used by the new plugin) and vice-versa decremented when the new plugin is closed again.

You can restrict the allowed plugin-references to types which have a minimum amount of bits in the plugin's type bitmask set. Further you can decline a restriction by indicating the exact name of a plugin. Please consider, that this check is passed in form of a class **HWMeta**, however the check is not executed by classes **Param** or **ParamBase**. You have to check this manually.

```
ito::Param param("serialPort", ito::ParamBase::HWRef, NULL, "description");

//additionally define the meta-information
ito::HWMeta *meta = new ito::HWMeta("SerialIO"); //restriction to plugins with name "SerialIO"
param.setMeta(meta, true); //takes ownership of meta-pointer. Do not delete meta from the stack

//returns the hardware pointer casted to ito::AddInBase*
ito::AddInBase* value = param.getVal<ito::AddInBase*>();

ito::RetVal retValue = param.setVal<void*>(ptr); //should return ito::retOk
bool numeric = param.isNumeric() //returns false
int len = param.getLen() // -1, since no length available
```

- **Reference to any initialized instance of DataObject (Type: DObjPtr)**

This is the reference to an instance of *DataObject*. The called method should check whether the instance has the necessary properties or type. Consider, that the flag *NoAutosave* is always set for that type. You can further give a meta information struct of class *DObjMeta* in order to specify the data object more in detail.

```
ito::DataObject *dObj = new ito::DataObject();
ito::Param param("image", ito::ParamBase::DObjPtr, dObj, "description");

//create a meta information where you only allow 2-dim data objects of type (u)int8.
//The necessary check is not automatically executed. You have to manually program it.
ito::DObjMeta *meta = new ito::DObjMeta(ito::tUInt8 | ito::tInt8, 2, 2);
param.setMeta(meta, true);

//returns the pointer casted to DataObject*
ito::DataObject* value = param.getVal<ito::DataObject*>();
ito::RetVal retValue = param.setVal<void*>(ptr); //should return ito::retOk
bool numeric = param.isNumeric() //returns false
int len = param.getLen() // -1, since no length available

//if you do not need param again, you can delete dObj:
```

```
delete dObj;
dObj = NULL;
```

- **Reference to any initialized instance of ito::pclPointCloud (Type: PointCloudPtr)**

This is the reference to an instance of *pclPointCloud*. The called method should check whether the instance has the necessary properties or type. Consider, that the flag *typeNoAutosave* is always set for that type.

```
ito::Param param("pcl", ito::ParamBase::PointCloudPtr, NULL, "description");

//returns the pointer casted to pclPointCloud*
ito::pclPointCloud* value = param.getVal<ito::pclPointCloud*>();
ito::RetVal retValue = param.setVal<void*>(ptr); //should return ito::retOk
bool numeric = param.isNumeric()                //returns false
int len = param.getLen()                        //-1, since no length available
```

- **Reference to any initialized instance of ito::pclPoint (Type: PointPtr)**

This is the reference to an instance of *pclPoint*. The called method should check whether the instance has the necessary properties or type. Consider, that the flag *NoAutosave* is always set for that type. Currently, there is no meta information struct available for that type.

```
ito::Param param("point", ito::ParamBase::PointPtr, NULL, "description");

//returns the pointer casted to pclPoint*
ito::pclPoint* value = param.getVal<ito::pclPoint*>();
ito::RetVal retValue = param.setVal<void*>(ptr); //should return ito::retOk
bool numeric = param.isNumeric()                //returns false
int len = param.getLen()                        //-1, since no length available
```

- **Reference to any initialized instance of ito::pclPolygonMesh (Type: PolygonMeshPtr)**

This is the reference to an instance of *pclPolygonMesh*. The called method should check whether the instance has the necessary properties or type. Consider, that the flag *NoAutosave* is always set for that type. Currently, there is no meta information struct available for that type.

```
ito::Param param("polygonMesh", ito::ParamBase::PolygonMeshPtr, NULL, "description");

//returns the pointer casted to pclPolygonMesh*
ito::pclPolygonMesh* value = param.getVal<ito::pclPolygonMesh*>();
ito::RetVal retValue = param.setVal<void*>(ptr); //should return ito::retOk
bool numeric = param.isNumeric()                //returns false
int len = param.getLen()                        //-1, since no length available
```

ParamBase - Reference

class ito::ParamBase

Public Type

Type enum

Values:

- NoAutosave = 0x010000 -
- Readonly = 0x020000 -
- In = 0x040000 -
- Out = 0x080000 -
- Pointer = 0x000001 -

- Char = 0x000002 -
- Int = 0x000004 -
- Double = 0x000008 -
- String = 0x000020 | Pointer -
- HwRef = 0x000040 | Pointer | NoAutosave -
- DObjPtr = 0x000010 | Pointer | NoAutosave -
- CharArray = Char | Pointer -
- IntArray = Int | Pointer -
- DoubleArray = Double | Pointer -
- PointCloudPtr = 0x000080 | Pointer | NoAutosave -
- IntPtr = 0x000100 | Pointer | NoAutosave -
- PolygonMeshPtr = 0x000200 | Pointer | NoAutosave -

Public Functions

ParamBase()

default constructor, creates “empty” tParam

ParamBase(const char * name)

constructor with name only creates a new *ParamBase* with name “name”, string is copied

Return

new *ParamBase* name “name”

Parameters

- name - name of new *ParamBase*

ParamBase(const char * name, const uint32 type)

constructor with name and type creates a new *Param* with name and type, string is copied

Return

new *Param* with name and type

Parameters

- name - name of new *ParamBase*
- type - type of new *ParamBase* for possible types see Type

ParamBase(const char * name, const uint32 type, const char * val)

constructor with name and type, char val and optional info creates a new *ParamBase* with name, type, string value. Strings are copied

Return

new *ParamBase* with name, type, string value

Parameters

- `name` - name of new *ParamBase*
- `type` - type of new *ParamBase* for possible types see Type
- `val` - character pointer to string pointer
- `info` - character pointer to string pointer holding information about this *ParamBase*

ParamBase(const char * name, const uint32 type, const double val)

constructor with name and type, double val creates a new *ParamBase* with name, type and val. Strings are copied.

Return

new *ParamBase* with name, type and val

Parameters

- `name` - name of new *ParamBase*
- `type` - type of new *ParamBase* for possible types see Type
- `val` - actual value

ParamBase(const char * name, const uint32 type, const int val)

constructor with name and type and int val creates a new *ParamBase* with name, type and val

Return

new *ParamBase* with name, type and val.

Parameters

- `name` - name of new *ParamBase*
- `type` - type of new *ParamBase* for possible types see Type
- `val` - actual value

ParamBase(const char * name, const uint32 type, const unsigned int size, const char * values)

array constructor with name and type, size and array creates a new *ParamBase* (array) with name, type, size and values.

Return

new *ParamBase* (array) with name, type, size and values.

Parameters

- `name` - name of new *ParamBase*
- `type` - type of new *ParamBase* for possible types see Type
- `size` - array size
- `val` - values

ParamBase(const char * name, const uint32 type, const unsigned int size, const int * values)

array constructor with name and type, size and array creates a new *ParamBase* (array) with name, type, size and values.

Return

new *ParamBase* (array) with name, type, size and values.

Parameters

- name - name of new *ParamBase*
- type - type of new *ParamBase* for possible types see Type
- size - array size
- val - values

ParamBase(const char * name, const uint32 type, const unsigned int size, const double * values)

array constructor with name and type, size and array creates a new *ParamBase* (array) with name, type, size and values.

Return

new *ParamBase* (array) with name, type, size and values.

Parameters

- name - name of new *ParamBase*
- type - type of new *ParamBase* for possible types see Type
- size - array size
- val - values

~ParamBase()

destructor

clear (frees) the name and in case a string value.

ParamBase(const *ParamBase* & copyConstr)

copy constructor creates *ParamBase* according to passed *Param*, strings are copied

Return

new *ParamBase* with copied values

Parameters

- copyConstr - *ParamBase* to copy from

const *ParamBase* **operator[]**(const int num)

braces operator for element-wise access in arrays

braces operator returns the value of the index num from the array

Return

new tParam with values of *ParamBase*[num] in the array

Parameters

- num - array index for which the value should be returned

ParamBase & **operator**=(const *ParamBase* & rhs)

assignment operator (sets values of lhs to values of rhs tParam, strings are copied)

assignment operator sets values of lhs to values of rhs *ParamBase*, strings are copied

Return

new *ParamBase* with copied values

Parameters

- rhs - *ParamBase* to copy from

ito::RetVal **copyValueFrom**(const *ito::ParamBase* * rhs)

just copies the value from the right-hand-side tParam (rhs) to this tParam.

bool **isNumeric**(void)

returns true if *Param* is of type char, int or double

int **addNameSuffix**(const char * suffix)

bool **isValid**(void)

returns whether tParam contains a valid type (true) or is an empty parameter (false, type == 0). The default tParam-constructor is always an invalid tParam.

uint32 **getType**(bool filterFlags = true)

returns parameter type (autosave flag and optional flag are only included if filterFlags is set false)

uint32 **getFlags**(void)

returns parameter flags (parameter type is not included)

uint32 **setFlags**(const uint32 flags)

sets parameter flags (parameter type remains untouched), for possible flags see `tParamType`

const char * **getName**(void)

returns parameter name (string is not copied)

uint32 **getAutosave**(void)

returns content of autosave flag - this flag determines whether the parameter value gets automatically saved to xml file when an instance of a plugin class is deleted (closed)

void **setAutosave**(const uint32 autosave)

sets content of autosave flag - this flag determines whether the parameter value gets automatically saved to xml file when an instance of a plugin class is deleted (closed)

int **getLen**(void)

template < typename _Tp >
ito::RetVal **setVal**(_Tp val)

setVal set parameter value - templated version

Return

RetVal with operation status sets the parameter value to the passed value, if the parameter type is inadequate it is set to the maximum value of template type

Parameters

- `val` - value to set to

template < typename _Tp >
ito::RetVal **setVal**(_Tp val, int len)

setVal set parameter value - templated version for arrays

Return

RetVal with operation status sets the parameter value to the passed value, if the length is below 1 or a Null pointer is passed an error is returned

Parameters

- `val` - value to set to
- `len` - length of array

```
template < typename _Tp >  
_Tp getVal(void)
```

`getVal` read parameter value - templated version returns the actual parameter value casted to the template parameter type. If the `tParam` has a non numeric type the largest value for the template type is passed.

Return

parameter value (numeric, casted)

```
template < typename _Tp >  
_Tp getVal(int & len)
```

`getVal` read parameter value - templated version for arrays returns the actual parameter value casted to the template parameter type. In '`len`' is returned what is supposed to be the length of the array. As only array references are used within `tParam` the actual size may differ.

Parameters

- `len` - length of array

Public Static Functions

```
uint32 typeFilter(uint32 type)
```

Param - Reference

```
class ito::Param
```

Public Functions

```
Param()
```

default constructor, creates "empty" `tParam`

```
Param(const char * name)
```

```
Param(const char * name, const uint32 type)
```

```
Param(const char * name, const uint32 type, const char * val, const char * info)
```

Param(const char * name, const uint32 type, const double minVal, const double maxVal, const double val, const char * info)

Param(const char * name, const uint32 type, const int minVal, const int maxVal, const int val, const char * info)

Param(const char * name, const uint32 type, const char minVal, const char maxVal, const char val, const char * info)

Param(const char * name, const uint32 type, const unsigned int size, const char * values, const char * info)

Param(const char * name, const uint32 type, const unsigned int size, const int * values, const char * info)

Param(const char * name, const uint32 type, const unsigned int size, const double * values, const char * info)

Param(const char * name, const uint32 type, const int val, *ParamMeta* * meta, const char * info)

Param(const char * name, const uint32 type, const double val, *ParamMeta* * meta, const char * info)

Param(const char * name, const uint32 type, const char val, *ParamMeta* * meta, const char * info)

Param(const char * name, const uint32 type, const unsigned int size, const double * values, *ParamMeta* * meta, const char * info)

Param(const char * name, const uint32 type, const unsigned int size, const int * values, *ParamMeta* * meta, const char * info)

Param(const char * name, const uint32 type, const unsigned int size, const char * values, *ParamMeta* * meta, const char * info)

~Param()

Destructor.

Param(const *Param* & copyConstr)

Copy-Constructor.

const *Param* **operator[]**(const int num)

braces operator for element-wise access in arrays

Param & **operator=**(const *Param* & rhs)

assignment operator (sets values of lhs to values of rhs *Param*, strings are copied)

ito::RetVal **copyValueFrom**(const *ParamBase* * rhs)

just copies the value from the right-hand-side *ParamBase* (rhs) to this tParam.

const char * **getInfo**(void)

< returns content of info string (string is not copied)

sets content of info string, if necessary the info buffer is freed first, passed string is copied

int **setInfo**(const char * info)

const *ParamMeta* * **getMeta**(void)

returns const-pointer to meta-information instance or NULL if not available

ParamMeta * **getMeta**(void)

returns pointer to meta-information instance or NULL if not available

bool **setMeta**(*ParamMeta* * meta, bool takeOwnership = false)

sets a new ParamMeta-instance as meta information for this *Param*

See

ito::ParamMeta

Parameters

- *meta* - is the pointer to any instance derived from *ParamMeta*
- *takeOwnership* - (default: false) defines, whether this *Param* should take the ownership of the ParamMeta-instance

bool **copyMetaFrom**(const *ParamMeta* * meta)

double **getMin**()

double **getMax**()

7.3.8 Parameters - Meta Information

Class ParamMeta

The class **ParamMeta** is the base class for all meta information classes. Parameters of class **Param** may contain pointers of that class, which then must be casted to the final implementation.

class **ito::ParamMeta**

Base class for all meta-information classes.

See

IntMeta, DoubleMeta, CharMeta, StringMeta, HWMeta, DObjMeta

Public Functions

ParamMeta()

ParamMeta(uint32 type)

~ParamMeta()

uint32 **getType**()

Class CharMeta, IntMeta and DoubleMeta

The classes **CharMeta**, **IntMeta** and **DoubleMeta** provide meta information for parameters of numeric types as well as their corresponding arrays.

class **ito::CharMeta**

Meta-information for *Param* of type Char or CharArray.

See

ito::Param

Public Functions

CharMeta(char minVal, char maxVal)

constructor with minimum and maximum value

char **getMin**()

returns minimum value

char **getMax**()

returns maximum value

void **setMin**(char val)

sets the minimum value

Parameters

- `val` - is the new minimum value, if this is bigger than the current maximum value, the maximum value is changed to `val`, too

void **setMax**(char val)

sets the maximum value

Parameters

- `val` - is the new maximum value, if this is smaller than the current minimum value, the minimum value is changed to `val`, too

Public Static Functions

CharMeta * **all**()

returns a new instance of *CharMeta*, where the min and max are set to the full range available for char.

class ito::IntMeta

Meta-information for *Param* of type Int or IntArray.

See

ito::Param

Public Functions

IntMeta(int minVal, int maxVal)

constructor with minimum and maximum value

int **getMin**()

returns minimum value

int **getMax**()

returns maximum value

void **setMin**(int val)

sets the minimum value

Parameters

- `val` - is the new minimum value, if this is bigger than the current maximum value, the maximum value is changed to `val`, too

void **setMax**(int val)

sets the maximum value

Parameters

- `val` - is the new maximum value, if this is smaller than the current minimum value, the minimum value is changed to `val`, too

Public Static Functions

IntMeta * **all**()

returns a new instance of *IntMeta*, where the min and max are set to the full range available for integers.

class **ito::DoubleMeta**

Meta-information for *Param* of type Double or DoubleArray.

See

ito::Param

Public Functions

DoubleMeta(double minVal, double maxVal)

constructor with minimum and maximum value

double **getMin**()

returns minimum value

double **getMax**()

returns maximum value

void **setMin**(double val)

sets the minimum value

Parameters

- *val* - is the new minimum value, if this is bigger than the current maximum value, the maximum value is changed to *val*, too

void **setMax**(double val)

sets the maximum value

Parameters

- *val* - is the new maximum value, if this is smaller than the current minimum value, the minimum value is changed to *val*, too

Public Static Functions

DoubleMeta * **all**()

returns a new instance of *DoubleMeta*, where the min and max are set to the full range available for double.

Class StringMeta

By this meta information you can give information about restrictions of strings to different strings. These strings can be interpreted as pure strings, as wildcard-expressions or regular expressions. The corresponding checks must be defined manually. If a string-parameter has an enumeration defined, where the strings are interpreted as strings, and if this parameter will automatically be parsed by any input mask in the GUI, the corresponding input text box becomes a drop-down menu with the given enumeration elements.

class **ito::StringMeta**

Meta-information for *Param* of type String.

See

ito::Param

Public Type

tType enum

Values:

- `String` - string elements should be considered as strings (exact match)
- `Wildcard` - string elements should be considered as wildcard-expressions (e.g. `*.doc`)
- `RegExp` - string elements should be considered as regular expressions (e.g. `^(.*)"[abc]{1,5}$`)

Public Functions

StringMeta(*tType* type)

constructor

Returns a meta information class for string-types.

See

tType

Parameters

- `type` - indicates how the string elements should be considered

StringMeta(*tType* type, const char * val)

constructor

Returns a meta information class for string-types.

See

tType

Parameters

- `type` - indicates how the string elements should be considered
- `val` - adds a first string to the element list

StringMeta(const *StringMeta* & cpy)

copy constructor

~StringMeta()

destructor

tType **getStringType()**

returns the type how strings in list should be considered.

See

iType

size_t **getLen()**

returns the number of string elements in meta information class.

const char * **getString**(size_t idx = 0)

returns string from list at index position or NULL, if index is out of range.

void **addItem**(const char * val)

Parameters

- `val` - adds another element to the string list.

StringMeta & **operator+=**(const char * val)

Class DObjMeta

This meta information class provides further information about allowed types and boundaries concerning the dimension of a data object.

class **ito::DObjMeta**

Meta-information for *Param* of type DObjPtr.

See

ito::Param

Public Functions

DObjMeta(uint32 allowedTypes = 0xFFFF, size_t minDim = 0, size_t maxDim = std::numeric_limits< int >::max())

int **getAllowedTypes**()

size_t **getMinDim**()

returns maximum allowed dimensions of data object

size_t **getMaxDim**()

returns minimum number of dimensions of data object

Class **HWMeta**

By that implementation of a meta information class you can provide information about references to other instantiated plugins. Every plugin is defined by a bitmask of enumeration **ito::tPluginType** (defined in **addInActor.h**). You can either add a minimum bitmask, that is required, to the **HWMeta**-instance or you can define an exact name of a plugin, which must be met.

class **ito::HWMeta**

Meta-information for *Param* of type HWPtr.

See

ito::Param

Public Functions

HWMeta(uint32 minType)

constructor

creates HWMeta-information struct where you can pass a bitmask which consists of values of the enumeration **ito::tPluginType**. The plugin reference of the corresponding *Param* should then only accept plugins, where all bits are set, too.

See

ito::Plugin, **ito::tPluginType**

HWMeta(const char * HWAddInName)

constructor

creates HWMeta-information struct where you can pass a specific name of a plugin, which only is allowed by the corresponding plugin-instance.

See

ito::Plugin

HWMeta(const *HWMeta* & cpy)

~HWMeta()

destructor

uint32 **getMinType**()

returns type-bitmask which is minimally required by plugin-reference. Default 0.

See

ito::tPluginType

```
char * getHWAddInName()
```

returns zero-terminated name of specific plugin-name or NULL if not specified.

7.3.9 Parameters - Validation

If a default-parameter (or let's say template) is given in form of an instance of class **Param** and have now a value in form of an instance of class **ParamBase**, you might be interested if the new value fits to the type and optionally the restrictions given by meta information of the template.

Therefore, the class **ParamHelper** (folder *helper*) provide method to validate ParamBase-instances with respect to a given meta information struct or to compare the compatibility of two different parameters.

Note: Since **ParamHelper** is not directly available for plugins, the most important methods are also made available by the **API** functions (see *itom API*). Therefore the API-call for the validator functions is also indicated below.

Validate with meta information

If you have access to any meta information instance (derived from class **ParamMeta**), you can check whether the value of an instance of class **ParamBase** or its derived class **Param** fits to the given requirements. There are different validator functions depending on the type of meta information.

```
static ito::RetVal validateStringMeta (const ito::StringMeta *meta, const char* value, bool mandatory = false)
```

This methods checks whether the string, passed by argument *value*, fits to the requirements of the string meta information *meta*. If it does not fit, *retError* is returned with an appropriate error message. If argument *mandatory* is false, *retOk* is also returned if the string is not given, hence, value is an empty string.

API-Call: ito::RetVal ito::itoValidateStringMeta(-same arguments-)

```
static ito::RetVal validateDoubleMeta (const ito::DoubleMeta *meta, double value)
```

This methods checks whether the number 'value' does not exceed the boundaries given by the double meta information 'meta'. If this is not the case *retError* with appropriate error message is returned, else *retOk*.

API-Call: ito::RetVal ito::itoValidateDoubleMeta(-same arguments-)

```
static ito::RetVal validateIntMeta (const ito::IntMeta *meta, int value)
```

This methods checks whether the number 'value' does not exceed the boundaries given by the integer meta information 'meta'. If this is not the case *retError* with appropriate error message is returned, else *retOk*.

API-Call: ito::RetVal ito::itoValidateIntMeta(-same arguments-)

```
static ito::RetVal validateCharMeta (const ito::CharMeta *meta, double value)
```

This methods checks whether the number 'value' does not exceed the boundaries given by the char meta information 'meta'. If this is not the case *retError* with appropriate error message is returned, else *retOk*.

API-Call: ito::RetVal ito::itoValidateCharMeta(-same arguments-)

```
static ito::RetVal validateHWMeta (const ito::HWMeta *meta, ito::AddInBase *value, bool mandatory = false)
```

This method checks whether the plugin given by 'value' fits to the requirements possibly defined in the 'meta'-plugin meta information struct. If this is the case *retOk* is returned, else *retError* with an appropriate error message. If 'value' is NULL *retOk* is only returned if argument 'mandatory' is *false*.

API-Call: ito::RetVal ito::itoValidateHWMeta(-same arguments-)

```
static ito::RetVal validateParam (const ito::Param &templateParam, const ito::ParamBase &param, bool strict = true, bool mandatory = false)
```

This method uses the methods above to check whether the value of 'param' is valid with respect to the type and meta information of 'templateParam'. If 'strict' is *false*, the type is tried to be converted to type

of 'templateParam' if possible. The 'mandatory' parameter is redirected to the corresponding validation methods above as therefore has the same meaning.

API-Call: -not yet provided-

The real documentation about the structure of plugins is organized as follows:

7.3.10 Basic programming structure of any itom-plugin

Introduction

The software **itom** obtains most functionality by mainly two concepts. On the one hand there is the python scripting language, which allows you to use almost all available python modules that are available for python 3.2 or higher. On the other hand, the entire measurement system becomes powerful by the possibility to enhance its functionality by several plugins.

These plugins are separated into three main groups:

- Type **actuator**: Plugins of this basic type should be used if you want to connect any actuator, like motor stages, piezo actuators, focussing systems, ... to **itom** (see [Plugin class - Basic information](#) and [Plugin class - Actuator](#))
- Type **dataIO**: Plugins of this basic type should be used for connecting any input or output device to **itom**. The main representative of this group are cameras as input device or the serial port as an input/output device (see [Plugin class - Basic information](#) and [Plugin class - DataIO](#)). This group is subdivided into the following sub-types:
 - **grabber** for cameras (Please consider that the class of the camera-plugin should not directly derive from *AddInDataIO* but from *AddInGrabber*, which is derived from the first.
 - **ADDA** for any analog-digital converters
 - **rawIO** for further input-output devices, like display windows for SLM or LCoS-sensors, serial ports or plugins which do not fit to any other group, since the type **dataIO** is the most flexible plugin type.
- Type **algo**: Plugins of this type mainly contain different algorithms and/or advanced user interfaces like dialogs, main windows, widgets, ... (see [Plugin class - Basic information](#) and [Plugin class - Algo](#))

Each plugin is a different project in your programming environment and is finally compiled as shared library (DLL on Windows).

Plugin load mechanism of itom

The **itom**-base directory contains a folder **plugins**. This folder itself usually consists of different subfolders each having the name of a specific plugin. The folder can then contain a release and/or debug-version of the specific plugin DLL as well as further files which are necessary for running the plugin. If your plugin is dependent on other files, please consider to read the specific information about how to publish dependencies of each plugin.

At the startup of **itom**, the application recursively scans the **plugins** folder and looks for any *DLL*-file on Windows machines or *a*-file on a Linux operating system. Then each DLL is tried to be loaded using the plugin system provided by the **Qt**-framework. The *DLL* can successfully be loaded if the following prerequisites are fulfilled:

- The plugin is a release version if **itom** is started in release mode OR
- The plugin is a debug version (this can for example be seen if the DLL-name ends with *...d.dll*) if **itom** is started in debug mode
- The plugin is compiled using the same major and minor version of **Qt** than **itom** (it is possible to load a plugin compiled with **Qt** 4.8.3 with **itom** compiled with 4.8.2)
- The plugin is compiled with the same compiler than **itom**

- If the plugin is dependent on other shared libraries which are not linked using a delay-load mechanism, the plugin can only be loaded if every necessary shared library can be found and successfully be loaded. If the dependency could not be loaded, the plugin-load fails with an error message *module could not be loaded*.
- The remarks contained in the plugin with respect to a minimum and maximum version number of **itom** must correspond to the version number of your **itom**
- The plugin must be compiled with the same version string of the class **ito::AddInInterface** than the version contained in **itom** (this is not the general version of **itom**). The version string of **AddInInterface** can be seen at the end of the file **addInInterface.h** in the **common**-folder.

An overview about the load status of all detected library files can be seen by calling the dialog **loaded plugins**, accessible by **itom**'s menu **help >> loaded plugins...**

Finally, every successfully loaded plugin is included in the dock-widget **Plugin** of **itom**.

Basic plugin structure

Every plugin consists at least of two classes, which are both derived from two different base classes. All possible base classes are provided in the files:

```
addInInterface.h
addInInterface.cpp
```

which are contained in the folder **common** of **itom**'s SDK. This folder contains further files which can be used in every plugin and contain interfaces and helper libraries with usefull functions for successfully and easily program a plugin. Additionally **itom** provides an application programming interface (API) such that plugins can access important methods of **itom**. For more information see [itom API](#).

The two classes of the plugin are as follows:

1. Interface- or factory-class (derived from class **AddInInterfaceBase**)

This class must be derived from the class **addInInterfaceBase** and is the communication tunnel between **itom** and the plugin itself using the plugin-framework of **Qt**. The plugin framework creates one single instances of this class when the plugin DLL is loaded (that means at startup of **itom**). Therefore this class is considered to be a singleton instance and since it is always loaded by **itom** even if it is not really needed, this class is kept small and only provides basic information about the plugin itself.

For further information about the structure of this interface class see [Plugin Interface Class](#).

2. Individual plugin class (derived from class **AddInDataIO**, **AddInGrabber**, **AddInActuator** or **AddInAlgo**)

This class is the main class of the plugin and should contain the main functionality of the plugin. Depending on the plugin type, this class is derived from any of the classes **AddInDataIO**, **AddInGrabber**, **AddInActuator** or **AddInAlgo**, which are also contained in the files mentioned above. All this classes internaly are derived from the base class **AddInBase**, which is the most general class used for plugin handling and organization in **itom**. Please do not directly derive from **AddInBase**.

In the case of an actuator, a camera or any other IO-device, every opened device is represented by one individual instance of its corresponding plugin class. Hence, it is possible to have multiple instances of every class opened in **itom**. The creation and deletion of any instance is at first requested by the **AddInManager** class (an internal class of **itom**) which itself redirects this request to the singleton instance of the interface class in the corresponding plugin (This is the interface class mentioned in point 1 above).

In the case of an algorithm-plugin, this class mainly contains a set of static methods, each being one individual algorithm or user interface. At startup of **itom** the singleton instance of the interface-class is created. Additionally, this individual plugin class also is instantiated once (singleton) at startup of **itom** and its internal *init*-method provides an overview (list) of all available algorithm and user-interface functions to **itom**. Additionally the default parameter sets for all

algorithms and widget-methods are requested by **itom** and startup and are then cached in order to provide faster access in any subsequent function calls.

Further information about the common parts of the plugin class, independent on the plugin's type, see *Plugin class - Basic information*. For detailed information about the implementation of the different plugin types, see *Plugin class - DataIO*, *Plugin class - Actuator* or *Plugin class - Algo*.

Communication between itom, Python and each plugin

The communication to plugins of type **actuator** and **dataIO** is only possible by calling the public methods defined in the base classes **AddInActuator** or **AddInDataIO**. In Python, there exist two classes **dataIO** and **actuator**. Both have an interface that is analog to the corresponding interface **AddInActuator** or **AddInDataIO** in C++. Therefore, if a certain method of these classes is called in python, the call is redirected to the corresponding plugin-method. However, this call is executed across a thread-change, since both python and each plugin (besides the algorithm-plugins) “live” in their own thread.

7.3.11 Plugin interface class

In order to give **itom** by the help of the **Qt**-plugin system access to your plugin, you must create a class which is inherited from the class **AddInInterfaceBase**. This base class is part of the **ito**-namespace and is defined in the files:

```
addInInterface.h  
addInInterface.cpp
```

These files lie in the **common** folder.

In your main header file or your plugin, called **MyPlugin** in the following, you can use the following code example in order to create that class:

```
1 //myPlugin.h  
2  
3 #include "../common/addInInterface.h" //adapt the path depending on the location of your plugin  
4  
5 class MyPluginInterface : public ito::AddInInterfaceBase  
6 {  
7     Q_OBJECT  
8     Q_INTERFACES(ito::AddInInterfaceBase)  
9  
10    public:  
11        MyPluginInterface(QObject *parent = 0);  
12        ~MyPluginInterface() {};  
13        ito::RetVal getAddInInst(ito::AddInBase **addInInst);  
14  
15    private:  
16        ito::RetVal closeThisInst(ito::AddInBase **addInInst);  
17 };
```

In the code example above, the macro directives *Q_OBJECT* and *Q_INTERFACES(ito::AddInInterfaceBase)* (lines 7+8) force the compiler in the pre-compilation step to create the necessary code (done by the **Qt** framework) such that the class fits to the **Qt**-plugin system and is able to communicate by the common signal-slot-system of **Qt**. Remember that every class which is finally derived from the *QObject*-class (like *AddInInterfaceBase* is, too) must have the *Q_OBJECT* macro defined.

The constructor *MyPluginInterface(...)*, defined in line 11 is called once by the AddIn-Manager of **itom** at startup in order to create the singleton instance of the class *MyPluginInterface*. In the body of this method you should provide basic information about your plugin (see section *The constructor of the plugin interface class*).

The destructor in line 12 usually does not require further implementation, such that the empty body already can be given in the header file.

Finally, there are also the methods *getAddInInst* and *closeThisInst* which are the most important methods. If an user or some other part of **itom** request an instance of this plugin (that means not an instance of the interface we are talking in this section, but of the real plugin), the AddInManager of **itom** calls the method *getAddInInst* of the corresponding interface class. Then this interface has to create an instance of the plugin and set the given double-pointer parameter to the pointer of this newly created instance.

Inversely, the AddInManager of **itom** will call *closeThisInst* of an interface in order to force the plugin interface class to delete the plugin instance, given by the *addInInst* parameter. This mechanism is usually used by so-called factory-classes. Therefore we can consider the interface class to be a factory for one or more instances of the plugin itself (For information about the plugin class see *Plugin class - Basic information*).

The constructor of the plugin interface class

In your main source file of your plugin you can implement the constructor of the plugin interface class in the following exemplary way:

```
MyPluginInterface::MyPluginInterface(QObject *parent)
{
    m_type          = ito::typeActuator; //or: ito::typeAlgo, ito::typeDataIO, ito::typeDataIO /
    setObjectName("MyPlugin"); //this is the name of the plugin how it appears in itom

    m_description   = QObject::tr("Description of MyPlugin");
    m_author        = "Author's name";
    m_version       = CREATEVERSION(0,1,0);
    m_minItomVer    = CREATEVERSION(1,0,0);
    m_maxItomVer    = MAXVERSION;

    m_autoLoadPolicy = ito::autoLoadKeywordDefined;
    m_autoSavePolicy = ito::autoSaveAlways;

    //initialize mandatory parameters for creating an instance of MyPlugin
    m_initParamsMand.append( ito::Param("param1", ito::Param::String, \
        "defaultValue", tr("translatable description").toAscii().data()) );
    ...

    //initialize optional parameters for creating an instance of MyPlugin
    m_initParamsOpt.append( ito::Param("optParam1", ito::Param::Int, 0, 10, 5, \
        tr("translatable description of optParam1").toAscii().data()) );
    ...
}
```

At first, the constructor consists of a section where you define basic information about the plugin itself. In the second part you will define a list of mandatory and optional parameters which are required if any user wants to create an instance of the plugin, e.g. the user wants to open a new camera or connect any motor.

Part 1 (Basic information):

int **m_type**

Type of this plugin. Possible types are an OR-combination of the enumeration **ito::tPluginType**:

- typeActuator for actuator-plugins
- typeDataIO | typeGrabber for cameras and other grabbing devices
- typeDataIO | typeADDA for any analog/digital converters
- typeDataIO | typeRawIO for any other input-output-devices, like serial ports, display windows...
- typeAlgo for a plugin providing algorithms, filters or any other methods as well as graphical user interfaces, dialogs, ... which enhance the functionality of **itom**

void **setObjectName** (const QString &name)

use this method to set the name of your plugin. This name should be simple and should not contain special characters, since it not only appears in the list of plugins but is also the string used for initializing a plugin by the python scripting language.

QString **m_description**

Give an advanced description of your plugin.

QString **m_author**

Use this string to denote the author(s) of this plugin

int **m_version**

This integer variable contains the version of your plugin. A version string always consists of a major, minor and patch value. All these values are combined in the integer variable and can be created using the macro **CREATEVERSION(major,minor,patch)** (defined in *sharedStructures.h*), where the values major, minor and patch are integer values, too.

int **m_minItomVer**

Use this variable to denote the minimum version number of **itom** which is necessary to run this plugin. If you don't have any specific minimum version, use the macro **MINVERSION**, defined in *sharedStructures.h* (folder *common*).

int **m_maxItomVer**

Use this variable to denote the maximum version number of **itom**. Versions higher than this value do not allow to run this plugin. If you don't care about any maximum version, use the macro **MAXVERSION**, defined in *sharedStructures.h* (folder *common*).

ito::tAutoLoadPolicy **m_autoLoadPolicy**

Depending on the value of this variable, the internal parameters of the plugin can be loaded from a *xml*-file and set after the plugin's *init*-method has been called. The possible values for that variable are given by the enumeration **ito::tAutoLoadPolicy** and are

```
enum tAutoLoadPolicy {
    autoLoadAlways          = 0x1, /*!< always loads xml file by addInManager */
    autoLoadNever           = 0x2, /*!< never automatically loads parameters from xml-file (default) */
    autoLoadKeywordDefined = 0x4  /*!< only loads parameters if keyword autoLoadParams=1 exists */
};
```

For more information about the loading and/or saving of plugin's parameters, see [Automatic loading and saving of plugin parameters](#).

ito::tAutoSavePolicy **m_autoSavePolicy**

Depending on the value of this variable, the internal parameters of the plugin can be saved to a *xml*-file at shutdown of a plugin instance. The possible values for that variable are given by the enumeration **ito::tAutoSavePolicy** and are

```
enum tAutoSavePolicy {
    autoSaveAlways          = 0x1, /*!< always saves parameters to xml-file at shutdown */
    autoSaveNever           = 0x2  /*!< never saves parameters to xml-file at shutdown (default) */
};
```

For more information about the loading and/or saving of plugin's parameters, see [Automatic loading and saving of plugin parameters](#).

bool **m_callInitInNewThread**

Usually, the plugin's *init* method, where for instance the hardware is started and initialized, is called in a new thread in order to keep the GUI reactive during the whole process. If you change this member from its default value **true** to **false**, *init* is executed in the main thread and afterwards the plugin is moved to the new thread. For more information, see [Plugin class - Basic information](#) or the info box *plugin-class-callInitThread*.

Part 2 (mandatory and optional parameters):

Note: This part is only important if you build plugins of the basic types **dataIO** or **actuator**, since only plugins of these types can have multiple parameters, hence, it is usefull to parametrize their constructors. For algorithm- or filter-plugins, you can let the vectors **m_initParamsMand** and **m_initParamsOpt** unchanged (hence empty).

If you create an instance of a plugin using the python language, you have mainly two possibilites:

- Plugins of type **dataIO** are addressed using the python type **dataIO**, which is a class of the module `itom`:

```
from itom import * # usually this import already has been done for you
variable = dataIO(PluginName,mandatoryParam1, ..., mandatoryParamN, optionalParam1, ...,
```

OR

```
import itom
variable = itom.dataIO(PluginName, mandatoryParam1, ..., optionalParam1, ...)
```

- Plugins of type **actuator*** are addressed using the python type ****actuator**, which is a class of the module `itom`, too. The call is then analogous to the examples above:

```
variable = actuator(PluginName,mandatoryParam1, ..., mandatoryParamN, optionalParam1, ...)
```

- Plugins of type **algo** do not have any corresponding class in `itom`, since they are globally organized by **itom**. Algorithms can be called using the method `filter()`, windows, dialogs or further user interfaces provided by plugins are loaded using the static method `createNewPluginWidget()` of class `itom.uiDialog`.

The constructor of each plugin can have a list of mandatory and optional parameters, which must or can be provided if creating an instance of the plugin. Internally, each parameter is a value of type **Param**, which is a class of **itom** and provides values of different types. Each value has a specific name, a default value and a description string, which should be given or set to NULL. Additionally, depending on the parameters type, a minimum and maximum value can be indicated. For more information about class *Param* see [Parameter-Container class of itom](#).

The mandatory parameters are contained in the vector

```
QVector<ito::Param> m_initParamsMand
```

Using the methods *append* or *insert* you can add an arbitrary number of values (type *tParam*) to this vector. The type *QVector* is a Qt-specific class which is similar to *std::vector*. The optional parameters are analogously contained in the vector

```
QVector<ito::Param> m_initParamsOpt
```

If one is creating an instance of the plugins, e.g. using the python commands above, **itom** is reading the given vector of mandatory of optional parameters. The first parameter of the constructors of the python class `itom.dataIO` or `itom.actuator` stands for the name of the plugin. The number of the following parameters must be equal or bigger than the length of the mandatory parameter vector. The first *n* parameters must exactly fit to the type, order and possible boundary values of the mandatory parameter vector. This vector is then copied and the values are replaced by the values given by the python-constructors.

If the following parameters in the constructor don't have any keywords, they must also fit to the types,... of the optional parameter vector. If there are not enough parameters given, the default value will be taken. Additionally, if the user gives keywords to the parameters, each parameter will be checked against its corresponding value in the optional parameter vector where keyword and parameter-name are equal. After the first parameter having a keyword no keyword-less parameters are accepted.

This is an example of creating a plugin with a set of parameters, where the last two parameters are tagged with their keywords:

```
variable = dataIO("MyPlugin",2.0,"test",delay=1000,file="C:\\test.dat")
```

After that the mandatory and optional parameter vectors are read, copied and that their values are replaced by the values given by the constructor, the instance of the plugin is created and the method **init** of the plugin class is called with the mandatory and optional parameter vector as argument. That's the basic way such a plugin instance is created and initialized.

The set of mandatory and optional parameters of each plugin, including their default, minimum and maximum value, their name and description string, can be returned in python using the method `itom.pluginHelp()`.

Method *getAddInInst* of the plugin interface class

As default implementation, you can copy the following code block for your implementation of the *getAddInInst*-method:

```
1  ito::RetVal MyPluginInterface::getAddInInst(ito::AddInBase **addInInst)
2  {
3      MyPlugin* newInst = new MyPlugin();
4      newInst->setBasePlugin(this);
5      *addInInst = qobject_cast<ito::AddInBase*>(newInst);
6      m_InstList.append(*addInInst);
7      return ito::retOk;
8  }
```

Since your plugin instance (**MyPlugin**) is finally derived from **AddInBase**, its private member **m_uniqueID** is automatically given an auto-incremented, unique number. Additionally you can also assign a unique identification string (member **m_identifier** of class **AddInBase**) to your instance which is then displayed in **itom**. Please only change **m_identifier** in the constructor or **init**-method of your actual plugin.

In the method above, it is assumed that your main class of your plugin *MyPlugin* is called *MyPlugin*, too. Then in line 3, a new instance of that class is created and this new instance is noticed about its own factory class in line 4. The factory class is hereby the pointer to this singleton interface class instance. Finally the given double pointer is set to the pointer of the newly created plugin instance. Finally, every plugin interface class has a protected member vector called *m_InstList* which contains a list of plugin instances opened by this interface (or factory). The newly created plugin is added to this list in line 6.

The return value of this method is of type **ito::RetVal**, which is set to the status **Ok**. For more information about the return value class **ito::RetVal** see *RetVal - The return type of itom methods*.

Method *closeThisInst* of the plugin interface class

For this method, you can basically copy the following default implementation:

```
1  ito::RetVal MyPluginInterface::closeThisInst(ito::AddInBase **addInInst)
2  {
3      if (*addInInst)
4      {
5          m_InstList.removeOne(*addInInst);
6          delete ((MyPlugin *)*addInInst);
7      }
8      return ito::retOk;
9  }
```

The **AddInManager** of **itom** is calling this method if the given plugin instance (parameter **addInInst**) should be deleted. If the parameter pointer is available, the plugin instance is removed from the list of loaded plugin instances (see *Method getAddInInst of the plugin interface class*) and the plugin instance is deleted.

7.3.12 AddInBase

class **ito::AddInBase**

Base class for all plugins.

The common methods and members are defined here. The available plugin type (actuator **AddInAcuator**, dataIO **AddInDataIO** and algo **AddInAlgo**) are derived from this class. At the main program startup all available plugins located in the plugin directory are searched and matched against the current plugin interface version. Then all compatible plugins can be check with the **AddInManager**. Up to that stage for each plugin only a lightweight **AddInInterface** **AddInInterfaceBase** class has been loaded. To use a plugin instances of the plugin class have to be instantiated. The **AddInInterface** is run in the calling thread whilst the plugin classes are run in separate threads. Therefore the plugin

functions are implemented as slots which can be used e.g. with the `invokeMethod` function. The base functionality included in this base class is getting the plugin's parameter list, getting the classes `uniqueID` (which is used e.g. for saving the parameter values) and optionally to bring up a configuration dialog.

Public Functions

const *Param* **getParamRec**(const QString name, bool * nameCheckOk = NULL)

method to retrieve a parameter from the parameter map (`m_params`)

returns parameter from `m_params` vector. If the parameter could not be found or if the given name is invalid an invalid *Param* is returned. If you provide the `nameCheckOk`-pointer, you will return a boolean value describing whether your name matched the possible regular expression.

The parameter name, that is search can have the following form:

until now, the `Idx` is ignored by this method.

Return

Param as copy of the internal `m_params`-map or empty *Param*, if name could not be resolved or found

Parameters

- `name` - is the name of the parameter
- `nameCheckOk` - returns true if name corresponds to the necessary syntax, else false

AddInInterfaceBase * **getBasePlugin**(void)

returns the interface of this instance.

See

AddInInterfaceBase

ito::RetVal **MoveToThread**(void)

creates new thread for the class instance and moves this instance to the new thread

const *ito::RetVal* **getParamList**(QMap< QString, *Param* > ** paramNames)

returns a map with the parameters of this plugin.

const *ito::RetVal* **getExecFuncList**(QMap< QString, ExecFuncParams > ** funcs)

returns list of registered additional functions

int **getID**()

retrieve the `uniqueID` of this instance

QString **getIdentifier()**

retrieve the unique identifier of this instance

int **hasConfDialog**(void)

determine if a configuration dialog is available

method indicates whether this plugin instance has a configuration dialog.

Overwrite this method if your plugin provides such a configuration dialog by simply returning 1 instead of 0.

Return

0 since the base implementation of a plugin does not have a configuration dialog. If there is a configuration dialog overwrite this method and return 1.

const *ito::RetVal* **showConfDialog**(void)

open configuration dialog

method called if the configuration dialog of the plugin should be shown.

Overwrite this method if your plugin provides a configuration dialog. This method is directly called by the main (GUI) thread. Therefore you can directly show your configuration dialog, connect it with signals and slots of your plugin instance (which possibly is executed in another thread), wait for any commands or that the dialog is closed and handle the result.

Return

retWarning since you have to overwrite this method in order to show your configuration dialog.

int **createdByGUI**()

returns true if this instance has firstly been created by the GUI

void **setCreatedByGUI**(int value)

method to set whether this instance has been firstly created by the GUI (true) or by any other component (Python, C++, other plugin,...) (false)

int **getRefCount**(void)

Returns the reference counter of this instance.

bool **hasDockWidget**(void)

Returns true if this plugin provides a dock widget, that can be shown in the main window.

QDockWidget * **getDockWidget**(void)

Returns the reference to the dock widget of this plugin or NULL, if no dock widget is provided.

void **dockWidgetDefaultStyle**(bool & floating, bool & visible, Qt::DockWidgetArea & defaultArea)

returns default style properties for dock-widget of plugin

int **isAlive**(void)

returns the alive-flag of this plugin

void **setAlive**(void)

sets the alive-flag to 1 ("still alive")

bool **isInitialized**(void)

returns in a thread-safe way the status of the m_initialized-member variable. This variable should be set to true at the end of the init-method.

void **setInitialized**(bool initialized)

sets in a thread-safe way the status of the m_initialized-member

QVector< *ito::AddInBase::AddInRef* * > * **getArgAddIns**(void)

returns vector of *AddInRef* instances.

Public Slot

ito::RetVal **init**

method for the initialisation of a new instance of the class (must be overwritten)

ito::RetVal **close**

method for closing an instance (must be overwritten)

ito::RetVal **getParam**

method for the retrieval of a parameter. The actual value is always passed as *ito::Param* (must be overwritten). See also *getParam*

ito::RetVal **setParam**

method to set a parameter. The actual value is always passed as *ito::tParam* (must be overwritten). See also *getParam*

ito::RetVal **setParamVector**

method for setting various parameters in a sequence

Using this method, only one over-thread call needs to be executed in order to set various parameters by calling `setParam` for each parameter.

See

`setParam`, `ParamBase`

Parameters

- `values` - is a vector of parameters to set
- `waitCond` - is the locked semaphore that is released at the end of the method.

ito::RetVal **execFunc**

this method can handle additional functions of your plugin.

Use `registerExecFunc` to register a specific function name and a set of mandatory and optional default parameters. If one of those functions is called (for instance by a python-call), this method is executed. Implement a switch case for the function name and start the execution. The mandatory and optional parameters are handled like it is the case for the creation (`init`-method) of a plugin. Additionally define an optional vector of output parameters, that are finally filled with a valid input during the execution of the function.

See

`registerExecFunc`, `init`

Parameters

- `funcName` - is the function name
- `paramsMand` - is the vector of mandatory parameters for the specific function name
- `paramsOpt` - is the vector of optional parameters for the specific function name
- `paramsOut` - is the vector of parameters (must have flag OUT, not IN), that are the return value(s) of the specific function call
- `waitCond` - is the semaphore in order guarantee, that the caller of this method waits until the function has been executed.

Signal

void **parametersChanged**

This signal usually is emitted if the vector `m_params` is changed.

Emit this signal for instance in `setParam` if the parameter has been changed in order to inform connected dock-widgets... about the change. This signal is also emitted if you invoke the slot `sendParameterRequest`.

See

`m_params`, `sendParameterRequest`

Parameters

- `params` - is the parameter-vector to send (usually `m_params`)

Protected Functions

AddInBase()

Constructor.

This constructor is called by any constructor of classes `AddInActuator`, `AddInDataIO` or `AddInAlgo`. Please make sure to call these constructors in your plugin constructor, since this is not automatically be done (due to the single parameter of the constructor).

This constructor simply initializes several member variables of this class.

Parameters

- `uniqueID` - is the unique identifier of this plugin instance. This identifier can be changed in the constructor or finally at the beginning of in the `init`-method. Afterwards it is used by different organizers and GUI components.

`~AddInBase()`

Destructor.

This destructor is automatically called if any plugin instance is destroyed. It does the following steps:

```
void createDockWidget(QString title, QDockWidget::DockWidgetFeatures features,
Qt::DockWidgetAreas allowedAreas = Qt::AllDockWidgetAreas, QWidget * content =
NULL)
```

Creates the dock-widget for this plugin.

```
ito::RetVal registerExecFunc(const QString funcName, const QVector< ito::Param > &
paramsMand, const QVector< ito::Param > & paramsOpt, const QVector< ito::Param >
& paramsOut, const QString infoString)
```

Registers an additional function with specific name and default parameters.

```
void setBasePlugin(AddInInterfaceBase * base)
```

sets the interface of this instance to base.

See

`AddInInterfaceBase`

Protected Attributes

`QThread * m_pThread`

the instance's thread

`AddInInterfaceBase * m_pBasePlugin`

the `AddInInterfaceBase` instance of this plugin

`QMap< QString, Param > m_params`

map of the available parameters

`QString m_identifier`

unique identifier (serial number, com-port...)

int **m_refCount**

reference counter, used to avoid early deletes (0 means that one instance is holding one reference, 1 that two participants hold the reference...)

int **m_createdByGUI**

1 if this instance has firstly been created by GUI, 0: this instance has been created by c++ or python

QVector< *ito::AddInBase::AddInRef* * > **m_hwDecList**

list of hardware that was passed to the plugin on initialisation and whose ref-counter was incremented

QMap< QString, ExecFuncParams > **m_execFuncList**

map with registered additional functions. funcExec-name -> (default mand-Params, default optParams, default outParams, infoString)

class **AddInRef**

Public Functions

AddInRef()

AddInRef(void * p, int t)

Public Members

int **type**

void * **ptr**

7.3.13 Plugin class - Basic information

All plugins, especially hardware-related ones (derived from **AddInActuator**, **AddInDataIO**, ...), have some similar or identical base components. These are described in this chapter. Therefore it is recommended to read this chapter before continuing with the specific documentation about the type of plugin you want to create. The information given in this chapter is mainly important for plugins finally derived from classes **AddInActuator** (actuator-plugins) or **AddInDataIO** (displays, cameras, AD-converter, ...). Algorithm or filter-plugins behave a little bit different, however it remains worth to read this document first.

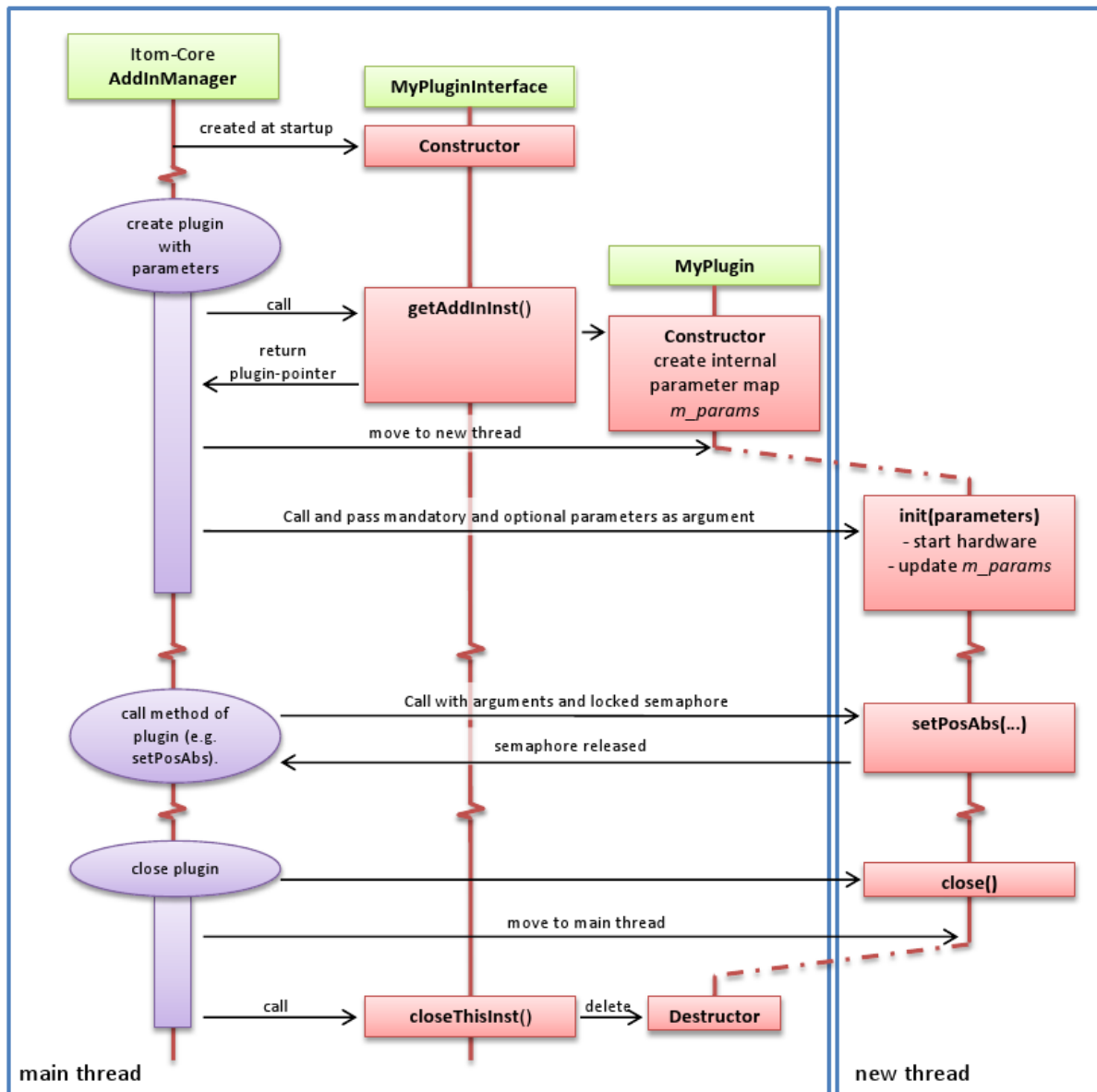
Common components for actuator and dataIO plugins

The following components and structures are equal for all plugins of type *actuator* or *dataIO* (including grabbers):

1. Every plugin has an internal set of parameters, that can be read or changed by the method **getParam** and **setParam** (class **AddInBase**).
2. These parameters are saved in the map **m_params**, member of class **AddInBase**.
3. Each parameter is an instance of class **ito::Param**.
4. The plugin itself is responsible if the new input to any parameter is valid or if the parameter is readonly as has to reject wrong input. For this purpose the parameters' meta information or flags can be used.
5. Every plugin executes its methods mainly in an own thread.

6. The communication between the plugin and its GUI-components (Config-Dialog, Dock-Widget in main window) must therefore be done using a signal-slot-connection or a simple invoke, that allows to call methods in another thread.
7. The plugin's **init** method is called with the mandatory and optional parameters, those default is set the in the plugin's interface and which are updated by the user input.

The basic scheme of the live-cycle of any plugin is depicted in figure *plugin-plugincall-scheme1*:



At startup of **itom**, one single instance (singleton) of each plugin interface class is created and managed by the **AddInManager**. Later, an instance of the real plugin can be requested, for instance in a python-script or by a C-function call. The arguments to this request are two vectors of mandatory and optional parameters (instances of class **ParamBase**), that are passed to the initialization of the plugin. The default state of these vectors is obtained by the corresponding interface class (vectors **m_initParamsMand** and **m_initParamsOpt**). These default values have or can be replaced by the values, e.g. indicated in the python script (this is automatically done by **itom**).

Then **AddInManager** calls **getAddInInst()** of the plugin interface, that is creating a new instance of the plugin by calling its constructor. Please consider, that the mandatory and optional parameters are not passed to the constructor yet. Then, the pointer to the plugin is passed back to the **AddInManager**, that now moves the plugin to a newly created thread. From that point on, methods of the plugin should only be called using a thread-communication technique, provided by **Qt**, e.g. using the method **invokeMethod** or the signal-slot-system of

Qt. Then the method **init** of the plugin is “invoked” by the **AddInManager** and the mandatory and optional initialization parameters are passed as arguments.

Next, the plugin can be used. Whenever a method of the plugin is called, either by python or any C-method, this call has to be executed again by an “over-thread” communication technique (**invokeMethod** or signal-slot). The last argument of such a call always is a locked instance of **ItomSharedSemaphore**. The caller then waits until the semaphore will be released in the plugin’s method or a timeout occurred. Please make sure, that you release the semaphore in your plugin’s method only at the point, when you don’t need any parameters given by the caller any more. Usually it is released at the bottom of your method. Only in special cases (like method **waitForDone** (for actuators) it makes sense to release it earlier. For more information about the semaphore see [ItomSharedSemaphore](#).

Finally, if the plugin should be closed, the inverse function calls with respect to the initialization are executed. That means, the **AddInManager** is at first invoking the method **call** of the plugin (still in its own thread). Then the plugin is moved back to the main thread and then **AddInManager** forces the deletion of the plugin by the method **closeThisInst** of the corresponding interface class.

Note: In some cases, it is not possible to initialize the hardware (some cameras) in another thread than the main thread. Since this initialization should be done in the **init** method of the plugin, **itom** provides a possibility to call **init** before moving the plugin to the new thread. Hence, those both steps in scheme *plugin-plugincall-scheme1* are switched. This can be done by setting the member **m_callInitInNewThread** of the plugin interface class from its default value **true** to **false**. However, only use this possibility if there is no other chance, since the GUI is completely blocked during an initialization executed in the main thread (see also [Plugin interface class](#)).

Plugin-Framework

The bare framework of any plugin-class of type **DataIO**, **Actuator**, ... (but not **Algo**), looks like this:

```
1  class MyPlugin : public ito::AddInActuator OR public ito::AddInDataIO OR public ito::AddInGrabber
2  {
3      Q_OBJECT
4
5      public:
6          friend class MyPluginInterface;
7
8      protected:
9          ~MyPlugin(){};
10         MyPlugin();
11
12     public slots:
13         ito::RetVal init(QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *paramsOpt,
14         ito::RetVal close(ItomSharedSemaphore *waitCond);
15
16         ito::RetVal getParam(QSharedPointer<ito::Param> val, ItomSharedSemaphore *waitCond = NULL);
17         ito::RetVal setParam(QSharedPointer<ito::ParamBase> val, ItomSharedSemaphore *waitCond = NULL);
18
19         ito::RetVal execFunc(const QString funcName, QSharedPointer<QVector<ito::ParamBase> > paramsMand,
20         ... QSharedPointer<QVector<ito::ParamBase> > paramsOpt, QSharedPointer<QVector<ito::ParamBase> > paramsGrab,
21         ... ItomSharedSemaphore *waitCond = NULL);
22
23     private slots:
24         void dockWidgetVisibilityChanged(bool visible);
25 };
```

In this chapter, hints about implementing the methods in the definitions above are given. In the detailed chapters about every type of plugin, these class definitions will then be extended by the type-specific methods.

At first, your plugin class must be derived from class **ito::AddInActuator** if you want to create an actuator plugin, from class **ito::AddInGrabber** if you want to create a camera or grabber plugin and finally class **ito::AddInDataIO** for other hardware-related plugins. All those classes are derived from **AddInBase**. Next,

the plugin's class must have the macro **Q_OBJECT** defined, such that the plugin is able to participate at the signal-slot-system of **Qt**. That is necessary for a multi-thread-communication with the plugin.

The constructor and destructor of the plugin are defined in the **protected** section. Only the interface class is able to call these methods, since the interface class is declared as friend of the plugin in line 6.

Then, there are five methods in the **public slots** section, which should be overwritten from their base definition in the class **AddInBase**. The methods **init** and **close** are important for the startup and shutdown of an instance of the plugin. **getParam** and **setParam** provide read and/or write access to parameters of the plugin, that are usually stored in the member-map **m_params** (however you can also use **getParam** and **setParam** to read and write other parameters. In some special cases, it is necessary to add further, arbitrary functions to the plugin, for instance in order to start an extra calibration process. These calls are executed by the general method **execFunc**, those first argument is the name of the function. You only have to implement this function, if you need this feature. Until now, this has only been necessary for a low amount of plugins.

Before the details about the implementation for these methods are explained, let us also take a look at the definition of the class **AddInBase** itself. Only few public or protected member functions of this class must be overwritten by the plugin (those, that have been described above). However, there are still a lot of further member methods which can be overwritten or useful for the functionality of your plugin. The description of the accessible member functions and attributes (public, protected, signals, public slots) is described in [AddInBase](#).

Now, let us give some hints about the implementation of the methods, that have to be overwritten:

Constructor

The constructor is a protected member method and should usually only be called by the corresponding interface class. A exemplary implementation is:

```

1  MyPlugin::MyPlugin(... further parameters ...) :
2      AddInActuator()
3  {
4      //create internal parameter map
5      ito::Param paramVal("name", ito::ParamBase::String | ito::ParamBase::Readonly, "MyPluginName");
6      m_params.insert(paramVal.getName(), paramVal);
7      //... add further parameters to map
8
9      //now create dock widget for this plugin (if available)
10     MyPluginDockWidget *myDockWidget = new MyPluginDockWidget(this);
11
12     //here: connect signals from the dock widget to the plugin
13     // connections that inform the dock widget about changes in parameters, status... should be c
14     // and destroyed in the method 'dockWidgetVisibilityChanged'.
15
16     //if you can and want, you can assign the unique identification string for this plugin here:
17     //m_idenfier = QString("my unique plugin nr: %1").arg(yourSerialNumber)
18     //you can also set this in the init-method.
19
20     Qt::DockWidgetAreas areas = Qt::LeftDockWidgetArea | Qt::RightDockWidgetArea;
21     QDockWidget::DockWidgetFeatures features = QDockWidget::DockWidgetClosable | QDockWidget::Dock
22     createDockWidget("MyDockWidgetName", features, areas, myDockWidget);
23 }
```

For the constructor consider the following remarks:

1. The constructor is executed in the main thread and therefore is able to create widgets like the dock widget.
2. The constructor can have an arbitrary amount and type of parameters; the method **getAddInInst** of the interface class must be adapted to them.
3. Call the constructor of the base class (e.g. **AddInActuator**, **AddInDataIO**, **AddInGrabber**) in your constructor.
4. The constructor does not get the mandatory and optional parameters given by to user for initializing the plugin.

5. Create all internal parameters that are part of the parameter-map **m_params** and provide some default values.
6. If your plugin should provide a dock widget (that can be shown as dockable toolbox in the main window of **itom**), follow the snippet from the example above.
7. If your dock widget should be visible at initialization of the plugin or should be undocked as default state, overwrite and change the method **dockWidgetDefaultStyle** of class **AddInBase**.
8. See the specific documentation of each plugin to see which internal parameters must be available and which conventions exist for some specific parameters, that can be created.
9. If you want, set the unique identification string **m_identifier** in this constructor. You can also set it later in the **init**-method. If you don't set it, the auto-assigned unique ID is used for identifying your instance.

Destructor

Use the destructor (executed in main thread) for some final deletion work. Usually an empty body of the destructor is sufficient.

```
MyPlugin::~MyPlugin() {}
```

Init

The method **init** has the following bare framework:

```

1  ito::RetVal MyPlugin::init(QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *paramsOpt
2  {
3      ItomSharedSemaphoreLocker locker(waitCond);
4      ito::RetVal retValue;
5
6      //use the content of paramsMand and paramsOpt (order and type
7      // with respect to m_initParamsMand and m_initParamsOpt of
8      // interface class) in order to initialize the hardware and
9      // change values of the m_params-map if necessary.
10
11     //if you want you can set the unique identification string here:
12     //m_identifier = QString("IAmAUniqueStringDescribingThisPlugin")
13
14     // emit signal about changed parameters
15     emit parametersChanged(m_params);
16
17     //release the wait condition and set its returnValue before
18     if (waitCond)
19     {
20         waitCond->returnValue = retValue;
21         waitCond->release();
22     }
23
24     setInitialized(true); //plugin is initialized
25     return retValue;
26 }
```

Consider the following hints for the implementation of the **init**-method:

1. Usually it is executed in new-thread (see *plugin-class-callInitThread*)
2. The parameters are the filled mandatory and optional parameter vectors, those default implementation has been given in the constructor of the interface class.
3. You can be sure, that both the order and type of those plugins remains invariant.
4. Initialize the plugin with respect to these parameters.

5. If necessary change the value of the internal parameters **m_params**.
6. If so, emit the signal **parametersChanged(m_params)**.
7. At the end of the method release the wait condition.
8. Afterwards call **setInitialized(true)** in order to confirm the initialization of the plugin.
9. See the specific documentation of each plugin to see, what else has to be done in the method **init**.

Close

The method **close** is always executed as last method in the plugin thread. Disconnect your hardware in this method. A simple exemplary implementation is:

```

1  ito::RetVal MyPlugin::close(ItomSharedSemaphore *waitCond)
2  {
3      ItomSharedSemaphoreLocker locker(waitCond);
4      ito::RetVal retValue;
5
6      //your code comes here
7
8      if (waitCond)
9      {
10         waitCond->release();
11         waitCond->returnValue = retValue;
12     }
13     return retValue;
14 }
```

getParam

This method is the getter-method for reading the current value of internal parameters, that usually are an item of the map **m_params**. In this method (like in others, too) methods provided by the **itom** API are used. Therefore, you need to include the API header in your source file, e.g. by

```
#include "common/apiFunctionsInc.h"
```

The prototype for the method *getParam* then looks like this:

```

1  ito::RetVal MyPlugin::getParam(QSharedPointer<ito::Param> val, ItomSharedSemaphore *waitCond)
2  {
3      ItomSharedSemaphoreLocker locker(waitCond);
4      ito::RetVal retValue;
5      QString key;
6      bool hasIndex = false;
7      int index;
8      QString suffix;
9      QMap<QString, ito::Param>::iterator it;
10
11     //parse the given parameter-name (if you support indexed or suffix-based parameters)
12     retValue += apiParseParamName(val->getName(), key, hasIndex, index, suffix);
13
14     if(retValue == ito::retOk)
15     {
16         //gets the parameter key from m_params map (read-only is allowed, since we only want to g
17         retValue += apiGetParamFromMapByKey(m_params, key, it, false);
18     }
19
20     if(!retValue.containsError())
21     {
22         //put your switch-case.. for getting the right value here
```

```
23
24     //finally, save the desired value in the argument val (this is a shared pointer!)
25     *val = it.value();
26 }
27
28 if (waitCond)
29 {
30     waitCond->returnValue = retValue;
31     waitCond->release();
32 }
33
34 return retValue;
35 }
```

The **getParam** is either called directly, by changing the thread or by the **Python** script execution, if the appropriate method **getParam** in **Python** is called. For guaranteeing a thread-safe implementation, the first argument is a shared pointer to a value of class **ito::Param**. In case of success, the requested value must be saved into this given parameter. The second argument is a string containing the parameter name.

It is possible to pass a single parameter name or to follow a given string-syntax in order to also give a certain array index as well as an additional suffix string. Rules for this syntax are given in the section *Possible parameter names for parameter-map `m_params`*. In order to split the given raw name into its possible components **name**, **index** and **suffix**, the method **parseParamName** defined in **helperCommon.h** can be used. In the exemplary implementation above, the parameter-name is finally searched in the map **m_params** and if found, the value of this parameter is copied to the given argument **val**. However, this implementation is the most basic one and you can also implement further lines of code.

Since this method can also be called from another thread, an instance of **ItomSharedSemaphore** is passed, that finally needs to be released. The destruction of the pointer **waitCond** is done by the locker-instance **locker**.

setParam

The method **getParam** is the analog method to **setParam**. However, in **setParam** you will not only passed a parametername but also the new value for this parameter. Then you need to implement the following checks:

- Check whether your internal parameter is **read-only** and if so, reject the set-command.
- Verify the compatibility of the parameter-type of the new and internal parameter.
- Consider further restrictions given by a possible meta information that is appended to the internal parameter and check if the new value fits to these restrictions.
- If any internal parameter changed, emit the signal **parametersChanged** with **m_params** as argument. This signal is globally defined in the base class **AddInBase** and is for instance connected with GUI-elements like the possible dock-widget (toolbox) of this plugin.

Finally, an exemplary (simplified) version for the method **setParam** is:

```
1  ito::RetVal MyPlugin::setParam(QSharedPointer<ito::ParamBase> val, ItomSharedSemaphore *waitCond)
2  {
3      ItomSharedSemaphoreLocker locker(waitCond);
4      ito::RetVal retValue(ito::retOk);
5      QString key;
6      bool hasIndex;
7      int index;
8      QString suffix;
9      QMap<QString, ito::Param>::iterator it;
10
11     //parse the given parameter-name (if you support indexed or suffix-based parameters)
12     retValue += apiParseParamName( val->getName(), key, hasIndex, index, suffix );
13
14     if(isMotorMoving()) //this if-case is for actuators only.
15     {
```



```

16         retValue += ito::RetVal(ito::retError, 0, tr("any axis is moving. Parameters cannot be set"));
17     }
18
19     if(!retValue.containsError())
20     {
21         //gets the parameter key from m_params map (read-only is not allowed and leads to ito::retError)
22         retValue += apiGetParamFromMapByKey(m_params, key, it, true);
23     }
24
25     if(!retValue.containsError())
26     {
27         //here the new parameter is checked whether it's type corresponds or can be cast into the
28         // value in m_params and whether the new type fits to the requirements of any possible
29         // meta structure.
30         retValue += apiValidateParam(*it, *val, false, true);
31     }
32
33     if(!retValue.containsError())
34     {
35         if(key == "async")
36         {
37             //check the new value and if ok, assign it to the internal parameter
38             retValue += it->copyValueFrom( &(*val) );
39         }
40         else if(key == "demoKey")
41         {
42             //check the new value and if ok, assign it to the internal parameter
43             retValue += it->copyValueFrom( &(*val) );
44         }
45         else
46         {
47             //all parameters that don't need further checks can simply be assigned
48             //to the value in m_params (the rest is already checked above)
49             retValue += it->copyValueFrom( &(*val) );
50         }
51     }
52
53     if(!retValue.containsError())
54     {
55         emit parametersChanged(m_params); //send changed parameters to any connected dialogs or d
56     }
57
58     if (waitCond)
59     {
60         waitCond->returnValue = retValue;
61         waitCond->release();
62     }
63
64     return retValue;
65 }

```

In this base implementation, the parametername is firstly searched in the **m_params**-map. If found, the internal parameter is checked for the **read-onlyness**. Next, the method distinguishes between numeric and non-numeric parameters, since the numeric-ones can possibly be casted from one numeric type to another one. If these prerequisites are met, you should then check the new value for certain restrictions and if this is met, too, the internal parameter can be filled with the content of the given, new value. Else, appropriate error messages should be returned and the semaphore **waitCond** must finally be released.

If you want to set multiple parameters in an uninterrupted sequence, you can also call or invoke the method **setParamVector** defined in **AddInBase**. You don't have to implement this method, since this is already done. For more infos about this method, see the definition of **AddInBase** ([AddInBase](#)).

Extend your plugin by registering further functions

As already mentioned above, you can extend the set of methods of each plugin, defined by their base classes, by registering further functions. This is for instance useful, if you want to provide methods for starting a gamma correction, starting a specific calibration, ...

Each such method has the same arguments than the **init**-method of the plugin:

- A vector of mandatory parameters (class **ParamBase**)
- A vector of optional parameters (class **ParamBase**)
- A Vector of out-only parameters (class **ParamBase**)

In the constructor of your plugin, you need to register each of those methods by also predefining the default set of all parameter-vectors (mandatory, optional, out). Only, if this step has been done, the corresponding method can be called, for instance by using the method **exec** in the **Python** scripting language. The call to this method finally leads to a call of the method **execFunc**, defined in the class **AddInBase**, which has to be overwritten in your plugin, if you want to provide at least one additional method.

The first parameter of **execFunc** is the name of the additional method. Therefore you need to check for this name and then execute the corresponding algorithm.

An exemplary implementation of the method **execFunc** is

```
1  ito::RetVal AmmsMachine::execFunc(const QString funcName, QSharedPointer<QVector<ito::ParamBase> >
2      ...QSharedPointer<QVector<ito::ParamBase> > paramsOpt, QSharedPointer<QVector<ito::ParamBase> >
3  {
4      ito::RetVal retValue = ito::retOk;
5      ito::ParamBase *param1 = NULL;
6      ito::ParamBase *param2 = NULL;
7
8      if(funcName == "saveXMLParams")
9      {
10         param1 = ito::getParamByName(&(*paramsMand), "filename", &retValue);
11         param2 = ito::getParamByName(&(*paramsOpt), "overwriteIfExists", &retValue);
12
13         if(!retValue.containsError())
14         {
15             retValue += XmlParameter::saveXmlParams( param1->getVal<char*>(), m_xmlParams, static
16         }
17     }
18     else
19     {
20         retValue += ito::RetVal::format(ito::retError,0,tr("function name '%s' does not exist").t
21     }
22
23     if(waitCond)
24     {
25         waitCond->returnValue = retValue;
26         waitCond->release();
27         waitCond->deleteSemaphore();
28         waitCond = NULL;
29     }
30
31     return retValue;
32 }
```

In this method, one additional function (name **saveXMLParams**) is integrated, which only is called with one mandatory and optional parameter and no further output parameter. The corresponding registration of this method is integrated in the constructor of your plugin:

```
1  //register exec functions
2  QVector<ito::Param> pMand;
3  pMand << ito::Param("filename", ParamBase::String | ParamBase::In, NULL, tr("absolute filename to
```

```

4 QVector<ito::Param> pOpt;
5 pOpt << ito::Param("overwriteIfExists", ParamBase::Int | ParamBase::In, 0, 1, 1, tr("parameter de
6 QVector<ito::Param> pOut;
7 registerExecFunc("saveXMLParams", pMand, pOpt, pOut, tr("description"));

```

Possible parameter names for parameter-map **m_params**

The parameters which you add to the **m_params** map, must have a name which fits to the following rules:

- The name starts with a lower or upper case character, hence a value between *a-z* or *A-Z*
- The first character of the name can be followed by an infinite number of alpha-numerical characters (including characters like *_* or *-*).

If the user tries to get or set a parameter e.g. using the python-commands **setParam** or **getParam**, the parameter name firstly is checked if it exists in the **m_params** vector before the corresponding getter or setter methods in the plugin are called. However not the full given parameter string has to match any available key-name in the **m_params** map, since there are further information which can be encoded in the parameter name:

Sometimes it is useful to create a parameter as integer or double array, e.g. for the speed values of every single axis. In this case you will create a parameter with the name **speed** having a type **typeDoubleArray**. Then the user can either access the whole array using the real parameter name **speed**, or the user can access one single value of the array by appending the index within brackets at the parameter name. In this case, it is the programmers task to parse the given parameter string and separate the parameter's real name (here: *speed*) and the index. Then the index has to be checked with respect to its lower and upper bound.

As further possibility, it is also allowed to append further information to a parameter. This is done by appending a colon-character to the parameter name followed by any string, which is the additional information string.

To sum this description up, let us assume that the parameter in **m_params** has the name **PARAMNAME**, which fits to the rules above. Then **itom** will accept parameter names, which correspond to the following rules:

- **PARAMNAME**
- **PARAMNAME[INDEX]**
- **PARAMNAME[INDEX]:ADDITIONALTAG**
- **PARAMNAME:ADDITIONALTAG**

INDEX has to be any fixed-pointer number, the **ADDITIONALTAG** can be any string.

It is the programmer's responsibility to split the given parameter name in the three components **PARAMNAME**, **INDEX** and **ADDITIONALTAG** if the corresponding parameter has the ability to handle indexed-values or even additional information (encoded in **ADDITIONALTAG**).

In order to execute this split you can use the method

```
ito::RetVal parseParamName(const QString &name, QString &paramName, bool &hasIndex, int &index, Q
```

defined in **helperCommon.h** and **helperCommon.cpp**.

Furthermore you can use the following regular expression:

```

1 QString regularExpression = "^[a-zA-Z]+\\w* (\\[ (\\d+) \\]) {0,1} (: (.*) ) {0,1} $";
2 QRegExp rx(regularExpression)
3 if(rx.indexIn(yourString) == -1)
4 {
5     //yourString does not match the regular expression
6 }
7 else
8 {
9     QStringList components = rx.capturedTexts();
10    //components consists of the following sub-strings:
11    /*
12    [0] full string

```

```
13     [1] PARAMNAME
14     [2] [INDEX] or empty-string if no index is given
15     [3] INDEX or empty-string if no index is given
16     [4] :ADDITIONALTAG or empty-string if no tag is given
17     [5] ADDITIONALTAG or empty-string if no tag is given
18     */
19 }
```

7.3.14 Plugin class - DataIO

Base idea behind any DataIO

- The DataIO operates with plugged systems require input and output, in generally cameras and frame grabbers but also serial ports or AD-DA-converter.
- One opened DataIO handles only one connected device. More cameras or ADDA-converter of the same type should be managed with a corresponding number of DataIO-ports.
- Depending on the plugin, more than one DataIO can be separately controlled by the plug-number or the vendor/camera-ID.
- Like any other plugin, every DataIO has a set of parameters, which can be set/get using python by the commands **setParam** and **getParam**.
- Every DataIO is executed in its own thread.
- Every DataIO can have one configuration dialog and one docking widget toolbox.
- All parameters are stored in **m_params**. They can be read or changed by methods **getParam** and **setParam**. Some parameters are read only!
- If parameters of DataIO changes, they also must be updated in the **m_params**-map.
- To connection between the dataIO-device (e.g. the camera) is done within the c++-method **init**. The initialization parameter should be given to **m_params**. In the end it must be disconnected by method **close**.
- The data acquisition is performed according to the grabber subtype. These subtypes are 'typeGrabber', 'typeADDA' and 'typeRawIO'.

The DataIO | typeGrabber plugin class

This is a subtype of DataIO for camera / framegrabber communication. The data acquisition is managed as followed:

- The methods **startDevice** and **stopDevice** opens and closes the capture logic of the devices to reduce CDU-load. For serial ports these functions are unnecessary.
- The method **acquire** starts the DataIO grabbing a frame with the current parameters. The function returns after sending the trigger. The function should be callable several times without calling get-/copyVal().
- The methods **getVal** and **copyVal** are the external interfaces for data grabbing. They call **retrieveData**. The function should not be callable without a previous call of **acquire** and than only once.
- In **retrieveData** the data transfer is done and frame has to copied. The function blocks until the triggered data is copied. In case retrieveData is called by getVal the frame has to be copied to **m_data**, an internal **dataObject**.
- The function **getVal** overwrites the IO-**dataObject** by a shallow copy of the internal **dataObject**. Empty objects are allowed. Warning read shallow copy of dataObject before usage.
- The function **copyVal** deep-copies data to the external IO-**dataObject**. The **dataObject** must have the right size and type. **dataObject** with ROI should not be overwritten. The ROI should be filled. Empty objects are allowed. In case of empty **dataObject** a new object with right size is allocated.

- The internal **dataObject** is checked after parameter changes by **checkData** (sizeX, sizeY and bpp) and, if necessary, reallocated.

A typical sequence in python is

```
1 device.startDevice()
2 device.acquire()
3 device.getVal(dObj)
4 device.acquire()
5 device.getVal(dObj)
6 device.stopDevice()
```

A sample header file of the DataIO's plugin class is illustrated in the following code snippet:

```
1  #include "../common/addInGrabber.h"
2  #include "dialogFireGrabber.h"
3  #include <qsharedpointer.h>
4
5  class FireGrabber : public ito::AddInGrabber
6  {
7      Q_OBJECT
8
9      protected:
10         ~MyDataIO(); /*! < Destructor*/
11         MyDataIO(); /*! < Constructor*/
12         ito::RetVal checkData(ito::DataObject *externalDataObject = NULL); /*!<
13         ito::RetVal retrieveData(ito::DataObject *externalDataObject = NULL); /*!< Wa
14
15
16     public:
17         friend class MyDataIOInterface;
18         const ito::RetVal showConfDialog(void); /*! Open the config nonmodal dialog t
19         int hasConfDialog(void) { return 1; }; /*!< indicates that this plugin has go
20
21
22
23     private:
24         cv::Mat m_pDataMatBuffer;
25
26         int m_CCD_ID; /*!< Camera ID */
27         bool m_isgrabbing; /*!< Check if acquire was called */
28
29         static int m_numberOfInstances;
30         ito::RetVal AlliedChkError(int errornumber); /*!< Map Allied-Error-Number to
31         const char* errmsg;
32         UINT8 *m_pImage; /*!< Pointer to actual image
33
34     public slots:
35         /*!< Get Camera-Parameter
36         ito::RetVal getParam(QSharedPointer<ito::tParam> val, ItomSharedSemaphore *wa
37         /*!< Set Camera-Parameter
38         ito::RetVal setParam(QSharedPointer<ito::tParam> val, ItomSharedSemaphore *wa
39         /*!< Initialise board, load dll, allocate buffer
40         ito::RetVal init(QVector<ito::tParam> *paramsMand, QVector<ito::tParam> *param
41         /*!< Free buffer, delete board, unload dll
42         ito::RetVal close(ItomSharedSemaphore *waitCond);
43
44         /*!< Start the camera to enable acquire-commands
45         ito::RetVal startDevice(ItomSharedSemaphore *waitCond);
46         /*!< Stop the camera to disable acquire-commands
47         ito::RetVal stopDevice(ItomSharedSemaphore *waitCond);
48         /*!< Softwaretrigger for the camera
49         ito::RetVal acquire(const int trigger, ItomSharedSemaphore *waitCond = NULL);
50         /*!< Wait for acquired picture, copy the picture to dObj of right type and si
```

```
51         ito::RetVal getVal(void *vpdObj, ItomSharedSemaphore *waitCond);
52
53         ito::RetVal copyVal(void *vpdObj, ItomSharedSemaphore *waitCond);
54
55         ito::RetVal ConvertY16(UINT32 XSize,UINT32 YSize,UINT8 *pBuf,UINT8 *pBGR);
56
57         void updateParameters(QMap<QString, ito::tParam> params);
58
59     private slots:
60
61 };
62
63 //-----
64
65 class DataIOInterface : public ito::AddInInterfaceBase
66 {
67     Q_OBJECT
68     Q_INTERFACES(ito::AddInInterfaceBase)
69
70     protected:
71
72     public:
73         DataIOInterface();
74         ~DataIOInterface();
75         ito::RetVal getAddInInst(ito::AddInBase **addInInst);
76
77     private:
78         ito::RetVal closeThisInst(ito::AddInBase **addInInst);
79
80     signals:
81
82     public slots:
83
84 };
```

Camera / Frame Grabber Parameters and Units

To achieve a unified interface and allow fast exchange between the different plugIns all dataIO have to follow the predefined conventions.

- Integrationtime named as “integration_time” in [s]
- Colordepth in bit [8,10-16,24]

Mandatory keywords for setParam / getParam

- “name”: Name of the plugIn as “typeString” || “readOnly”
- “integration_time”: The integration time as “typeDouble”
- “frame_time”: The time between to frames as “typeDouble” (usually “readOnly”)
- “gain”: The gain of the AD-Converter normed between 0..1 as “typeDouble”
- “offset”: The offset of the AD-Converter normed between 0..1 as “typeDouble”
- “x0”: first pixel of ROI as “typeInt”
- “x1”: last pixel of ROI as “typeInt”
- “y0”: first pixel of ROI as “typeInt”
- “y1”: last pixel of ROI as “typeInt”
- “sizex”: Current width of the grabber-ROI as “typeInt” || “readOnly”

- “sizey”: Current height of the grabber-ROI as “typeInt” || “readOnly”
- “bpp”: Current bits per pixel of the grabber as “typeInt”. Usual values are 8-bit, 10-bit - 16-bit, 24-bit color.

Optional but reserved keywords for `setParam` / `getParam`

- “binning”: Binning of the pixels in both directions as “typeInt” in form [BinningX? BinningY?] between “0101” and “xy”

The `DataIO` | `typeADDA` plugin class

The `DataIO` | `typeRawIO` plugin class

7.3.15 Plugin class - Actuator

If you want to create a plugin in order to access piezo- or motor-stages, multi-axes-machines,... it is intended to derive your plugin class from `ito::AddInActuator`.

Base idea of any actuator

The actuator interface has been developed with the following base ideas:

- The actuator can consist of different axes. The first axis is indexed with the number 0.
- The plugin contains a set of parameters (like every other plugin), which can be set or get using the public methods `setParam` or `getParam` or the appropriate methods in Python.
- Some of those parameters must be available, others are optional, but if you implement them, you should follow some rules concerning name and type of the parameter, and of course you can add an infinite list of further parameters.
- Every actuator is executed in its own thread.
- Every actuator can have one configuration dialog and one docking widget toolbox, that is directly included in the GUI of `itom`.
- The current position of all axes should be stored in the vector `m_currentPos`, that is a member of class `AddInActuator`.
- The new target position for all axes must be stored in the vector `m_targetPos`, that is also a member of class `AddInActuator`.
- The current status of all axes should be stored in the vector `m_currentStatus` (member of class `AddInActuator`).
- Make sure at the initialization of your plugin that all three member vectors are initialized with the size of the numbers of axes.
- Changes to the current status or position are signaled by the signal `actuatorStatusChanged` (class `AddInActuator`).
- Changes to the target position vector is signaled by the signal `targetChanged` (class `AddInActuator`).
- Any GUI elements should only get live information about the position and status of the actuator by connecting to these signals (`actuatorStatusChanged` or `targetChanged`), since the communication to GUI elements must be executed across multiple threads.
- Try to only connect to these signals if you really need this information, since the request of live-status and -position-information is time-consuming for certain motors. For example, a dock widget should only connect to these signals, if it is visible. This can be done by overwriting the slot `dockWidgetVisibilityChanged` of class `AddInBase`.

- Methods like **setPosAbs**, **setPosRel**, **setParam**, **calib** or **setOrigin** should only execute their given task if the motor is not moving at this moment. This can be checked using the method **isMotorMoving()**, defined in **AddInActuator**.
- The method **waitForDone** (pure virtual method of class **AddInActuator**) has to be overwritten. This method should only return if the requested axes reached their target position, reached a end switch or a timeout occurred. If the hardware is able to give sophisticated live information about the current status and position of each axis, you could continuously adapt the values in the members **m_currentStatus** or **m_currentPos**; else you have to guess these values. If the axes are asynchronously moved, the semaphore **waitCond** has to be released immediately before the loop waiting for the target position starts. Then the caller can directly continue working. In synchronous mode (default behaviour), **waitCond** is only released, if all axes are not moving any more.

Programming steps

In order to program the actuator plugin, follow these steps:

1. Create the header and source file for your plugin “MyAlgoPlugin”.
2. Create the interface (or factory) class “MyAlgoPluginInterface”. For details about how to create such an interface class, see *Plugin interface class*.
3. Create the plugin class “MyAlgoPlugin” with respect to the exemplary implementation, given in the next section.
 - Consider which internal parameters, that can be read and/or written by the user, your plugin has. Add these parameters in the constructor of your plugin to the **m_params**-vector.
 - Implement the **init**-method that gets the initial parameters, defined in the interface class.
 - Implement the methods **getParam** and **setParam**, which are the getter- and setter-methods for the internal parameters.
 - Implement the motor-specific methods, including **waitForDone**

Actuator plugin class

A sample header file of the actuator’s plugin class is illustrated in the following code snippet:

```
1  #include "../common/addInInterface.h"
2
3  #include "dialogMyMotor.h"
4  #include "dockWidgetMyMotor.h"
5
6  class MyMotor : public ito::AddInActuator
7  {
8      Q_OBJECT
9
10     protected:
11         ~MyMotor() {};          /*! < Destructor*/
12         MyMotor(int uniqueID); /*! < Constructor*/
13
14
15     public:
16         friend class MyMotorInterface;
17         const ito::RetVal showConfDialog(void);    /*!< Opens the modal configuration dialog (ca
18         int hasConfDialog(void) { return 1; }; /*!< indicates that this plugin has got a configur
19
20     private:
21         ito::RetVal waitForDone(int timeoutMS = -1, QVector<int> axis = QVector<int>() /*if empty
22
23     signals:
24         void parametersChanged(QMap<QString, ito::tParam> params); /*!< Sends a signal if parame
```


25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63

```

public slots:
    /// get/set parameters
    ito::RetVal getParam(QSharedPointer<ito::tParam> val, ItomSharedSemaphore *waitCond = NULL);
    ito::RetVal setParam(QSharedPointer<ito::tParam> val, ItomSharedSemaphore *waitCond = NULL);

    /// init/close method
    ito::RetVal init(QVector<ito::tParam> *paramsMand, QVector<ito::tParam> *paramsOpt, ItomSharedSemaphore *waitCond);
    ito::RetVal close(ItomSharedSemaphore *waitCond);

    /// calibration for single or multiple axis
    ito::RetVal calib(const int axis, ItomSharedSemaphore *waitCond = NULL);
    ito::RetVal calib(const QVector<int> axis, ItomSharedSemaphore *waitCond = NULL);

    /// current axis position is new zero-position
    ito::RetVal setOrigin(const int axis, ItomSharedSemaphore *waitCond = NULL);
    ito::RetVal setOrigin(const QVector<int> axis, ItomSharedSemaphore *waitCond = NULL);

    /// Reads out status request answer and gives back ito::retOk or ito::retError
    ito::RetVal getStatus(QSharedPointer<QVector<int> > status, ItomSharedSemaphore *waitCond = NULL);

    /// get current position of single or multiple axis (in mm or degree)
    ito::RetVal getPos(const int axis, QSharedPointer<double> pos, ItomSharedSemaphore *waitCond = NULL);
    ito::RetVal getPos(const QVector<int> axis, QSharedPointer<QVector<double> > pos, ItomSharedSemaphore *waitCond = NULL);

    /// move one or more axis to certain absolute positions (in mm or degree)
    ito::RetVal setPosAbs(const int axis, const double pos, ItomSharedSemaphore *waitCond = NULL);
    ito::RetVal setPosAbs(const QVector<int> axis, QVector<double> pos, ItomSharedSemaphore *waitCond = NULL);

    /// move one or more axis by certain relative distances (in mm or degree)
    ito::RetVal setPosRel(const int axis, const double pos, ItomSharedSemaphore *waitCond = NULL);
    ito::RetVal setPosRel(const QVector<int> axis, QVector<double> pos, ItomSharedSemaphore *waitCond = NULL);

    /// if this slot is triggered, the current status and position is emitted (e.g. for actual position)
    ito::RetVal RequestStatusAndPosition(bool sendActPosition, bool sendTargetPos);

private slots:
    void dockWidgetVisibilityChanged( bool visible ); /*!< this slot is invoked if the visibility of the dock widget changes */
};

```

Unit Conventions

In order to have a unique behaviour of all plugins, respect the following unit conditions:

- Length values in **mm**
- Angles in **degree**
- Velocity in **mm/sec** or **degree/sec**
- Acceleration in **mm/sec^2** or **degree/sec^2**

Mandatory keywords for setParam / getParam

- “name”: Name of the plugIn as “typeString” || “readOnly”
- “numAxis”: Number of axis as “typeInt” || “readOnly”
- “async”: Toggle wait behavior during movement. As “typeInt”
- “speed”: Speed for ever axis of the stage as “typeDoubleArray”. For single axis stages as “typeDouble”. Signle axis must not throw an error of called with speed[0]

Optional but reserved keywords for setParam / getParam

- “accel”: Acceleration for ever axis of the stage as “typeDoubleArray”. For single axis stages as “typeDouble”. Signle axis must not throw an error of called with accel[0]
- “decel”: Deceleration for ever axis of the stage as “typeDoubleArray”. For single axis stages as “typeDouble”. Signle axis must not throw an error of called with decel[0]

7.3.16 Plugin class - Algo

An **algorithm plugin** can provide an arbitrary number of filter-methods and widgets, hence external windows, dialogs... which can be displayed by **itom**.

Filter-Method A filter-method is any algorithm, which is parameterized with a set of mandatory and optional parameters and may return a set of output parameters. Similar to the flexible initialization of plugins of type **dataIO** or **actuator**, the default-values of these parameters are given by the plugin itself and can also be different for each filter-method. Please consider that a filter-method can finally be called from different threads, therefore it is not allowed to directly use any GUI-elements in such plugins. If the appropriate method is called by the python function **filter**, it is executed in the python thread, if the method is called form any toolbox or dialog, it is usually called in their context (main thread) but it can also be, that the filter-method is executed by any other worker thread. If a filter-method only consists of ordinary algorithmic components, you should not get any problems with these kind of flexibility.

Widgets The widget-methods of the plugin will be called in the same way than filter-methods with a set of mandatory and optional parameters. Then, the widget-method creates a new instance of a widget, window or any other GUI element, derived from **QWidget** is returned and can then be displayed. This method will always be called in the context of the main thread (GUI-thread).

Plugin-Structure

Like all the other types of plugins, the **Algo**-plugin must at least consist of two classes. The first (*here*: **MyAlgoPluginInterface**) is the interface, or factory class, necessary for the successful load of the plugin by the **Qt**-plugin system. The “real” plugin class (*here*: **MyAlgoPlugin**) is then accessed by **itom** through appropriate methods in class **MyAlgoPluginInterface**.

Factory-Class

The structure and exemplary implementation of a factory class **MyAlgoPluginInterface** is mainly explained in *Plugin interface class*. Since the main class **MyAlgoPlugin** is mainly behaving like a static, singleton class, it will never be instantiated by the user, like it is the case for plugins of type **dataIO** or **actuator**. Therefore it makes no sense to define some default mandatory and optional parameters for the constructor of **MyAlgoPlugin**, such that the programmer should the vectors **m_initParamsMand** and **m_initParamsOpt** be unchanged.

Plugin-Class

The raw scheme for your plugin-class is as follows:

Header-File (myPluginAlgo.h)

```
1  #ifndef MYALGOPLUGIN_H
2  #define MYALGOPLUGIN_H
3
4  #include "../common/addInInterface.h" //or similar path
5
6  class MyPluginAlgoInterface : public ito::AddInInterfaceBase
7  {
8  ... //see in documentation for interface-class
9  };
```

```

10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39

```

```

class MyPluginAlgo : public ito::AddInAlgo
{
    Q_OBJECT

protected:
    MyPluginAlgo(int uniqueID);
    ~MyPluginAlgo() {};

public:
    friend class MyAlgoPluginInterface;

    //filter-method1
    static ito::RetVal filter1(QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *paramsOpt, ItoSharedSemaphore *waitCond);
    static ito::RetVal filter1Params(QVector<ito::Param> *paramsMand, QVector<ito::Param> *paramsOpt, ItoSharedSemaphore *waitCond);

    //for every further filter-method define another pair like above

    //widget-method1
    static QWidget* widget1(QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *paramsOpt, ItoSharedSemaphore *waitCond);
    static ito::RetVal widget1Params(QVector<ito::Param> *paramsMand, QVector<ito::Param> *paramsOpt, ItoSharedSemaphore *waitCond);

    //for every futher widget-method define another pair like above

public slots:
    ito::RetVal init(QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *paramsOpt, ItoSharedSemaphore *waitCond);
    ito::RetVal close(ItoSharedSemaphore *waitCond);
};

#endif

```

First of all, our algorithm plugin class is derived from the class **AddInAlgo** from within the **ito**-namespace. This base class is defined in the *addInInterface.h* header that has to be included. Again, our plugin is ultimately derived from **QObject** and the *Q_OBJECT* macro must appear in the class definition in order to be able to use any services provided by Qt's meta-object system, such as the signal slot mechanisms. Additionally our plugin class has to be a friend of its interface class (see section *Plugin interface class*), such that the factory (interface) class is allowed to access the protected constructor.

Both, the pair of methods for one filter-method and one widget-method consists of the real filter- or widget-method (*here*: named with *filter1* and *widget1*) and their corresponding parameter-methods which generates the default vectors for the mandatory, optional and output (filters only) parameters. This method is necessary, since the **itom** base application and the python scripts have no way of knowing what parameters the filter methods or widget generation methods expect.

These default-parameter methods have the following implementation:

```

1  ito::RetVal MyAlgoPlugin::filter1Params(QVector<ito::Param> *paramsMand, QVector<ito::Param> *paramsOpt, ItoSharedSemaphore *waitCond)
2  {
3      ito::Param param;
4      ito::RetVal retval = ito::retOk;
5      retval += prepareParamVectors(paramsMand, paramsOpt, paramsOut);
6      if(retval.containsError()) return retval;
7
8      param = ito::Param("mand1", ito::ParamBase::DObjPtr | ito::ParamBase::In, NULL, tr("description of mandatory parameter 1"));
9      paramsMand->append(param);
10     param = ito::Param("mand2", ito::ParamBase::String | ito::ParamBase::In, NULL, tr("description of mandatory parameter 2"));
11     paramsMand->append(param);
12     param = ito::Param("opt1", ito::ParamBase::Double | ito::ParamBase::In, 0.0, tr("description of optional parameter 1"));
13     paramsOpt->append(param);
14
15     return retval;
16 }

```

In this exemplary case, the method **filter1** requires two mandatory parameters, where the first one is a dataObject-

pointer and the second one is a string. Additionally one can give one optional parameter, namely a double-value. This filter does not have output-parameters.

Note: Please don't forget to specify your parameters with the flags **In**, **Out** or **In/Out**. Mandatory and optional parameters can not be **Out** only, however output parameters must be **Out** only. Please consider that pure **Out** values is only allowed for non-pointer-type values (hence: Char, Int, Double, String, IntArray, CharArray and DoubleArray are allowed). For widget-methods, the output-value default implementation is ignored and therefore the corresponding vector should not be changed.

In line 3 of the method above, the three arguments are checked for not being NULL and are cleared within the method **prepareParamVectors**, which is already defined in class **ito::AddInAlgo**. Next, the default implementations of parameters (see *Parameter-Container class of itom*) are created and appended to the corresponding vector. Finally the method is called in order to get the three default vectors. Then, the values are merged with the values given by the user or other plugins and the real filter- or widget-method is called with the same vectors, however each parameter is now casted from **Param** to **ParamBase**.

Note: In order to have an efficient call, the parameter method is only called once at startup of **itom** and the default implementations are hashed by **ito::AddInManager**. The consequence for the plugin programmer is, that it is not allowed to have time- or situation-dependent changes in the default-values. This might not be considered. Only the implementation at startup is relevant and must not be changed!

The *plugin*-methods give the user additionally the possibility to get a readable output of the set of desired parameters including their descriptions and types in the command line by using the following python-command:

```
filterHelp("MyAlgoPlugin")
```

If the argument string does not fit to any specific filter, an enumeration of all filters containing this string will be printed.

Now the filter-method or widget-method itself can be implemented:

Filter-Methods

After that you implemented the parameter-method in order to generate default parameters for your filter-method (see section above), you can now implement the filter-method itself. The implementation might follow this scheme:

```

1  ito::RetVal MyAlgoPlugin::filter1(QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *p
2  {
3      ito::RetVal retval = ito::retOk;
4
5      //1. Section. Getting typed in or in/out parameters from paramsMand and paramsOpt
6      // Make sure that you access only parameters, that have been defined in the corresponding pa
7      // The order and type is important.
8
9      //possibility 1 (index-based access):
10     ito::DataObject *dObj = (*paramsMand)[0].getVal<ito::DataObject*>();
11     const char *filename = (*paramsMand)[1].getVal<char*>(); //don't delete this pointer (borrowed
12     double opt1 = (*paramsOpt)[0].getVal<double>();
13
14     //possibility 2 (name-based access):
15     ito::DataObjec *dObj = (ito::DataObject*)ito::getParamByName(paramsMand, "mand1", &retval)->
16     const char *filename = ito::getParamByName(paramsMand, "mand2", &retval)->getVal<char*>();
17     double opt1 = ito::getParamByName(paramsOpt, "opt1", &retval)->getVal<double>();
18
19     //2. Section. Algorithm.
20     // include here your algorithm. make sure, that you only change values of pointer-based para
21     // that you have defined with the flags In/Out.
22

```

```

23     //3. Section. Optionally put results into the paramsOut-vector
24     // Make sure that you defined the corresponding parameter with right type in the correspondi
25     // The implementation in the next lines is only one example and has not be defined before. Be
26     (*paramsOut)[0].setVal<int>(2);
27     (*paramsOut)[1].setVal<char*>("we are done");
28
29     return retval;
30 }

```

For the parsing of the given input parameters, you can (like stated above) either directly access them using their index in the vector or you can use the high-level method **getParamByName**, which is provided in the file **helperCommon.h** and **helperCommon.cpp** in the **common**-folder. If you want to use this method, integrate both files in your project and include the header file in your plugin file.

Todo

Move `getParamByName` into the API and remove it from `helperCommon`. Correct the documentation.

Note: Always make sure, that if you are access (read or write) any parameter in any of the three vectors, you must have the specific parameter defined in the corresponding parameter method. If you defined there a parameter of certain type and appended it to one of the vectors, you can be sure, that a parameter of same type is available at the same position in the arguments of the filter-method call.

Note: If you want to use methods provided by the **itom**-API, see [itom API](#) and consider the additional lines of code in your implementation.

Widget-Method (GUI-Extensions)

If you want to provide a user-defined window, dialog or widget (which is then rendered into a dialog), you have to implement an appropriate method which follows this base structure:

```

1  QWidget* MyAlgoPlugin::widget1(QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *paramsOpt)
2  {
3      //1. Section. Getting typed in or in/out parameters from paramsMand and paramsOpt
4      // Make sure that you access only parameters, that have been defined in the corresponding pa
5      // The order and type is important. Do it like in the method 'filter1' above.
6
7      //2. Pre-requisite: You have in your plugin project a class, which is derived from QMainWindow
8      // This class can also include an ui-file, which has been designed using the QtDesigner. The
9
10     //Create an instance of that class and return it. The instance is deleted by the caller of the
11     Widget1 *win = new Widget( /* your parameters */ );
12     QWidget *widget = qobject_cast<QWidget*>(win); //cast it to QWidget, if it isn't already.
13     if(widget == NULL)
14     {
15         retValue += ito::RetVal(ito::retError,0,tr("The widget could not be loaded").toAscii().data());
16     }
17     return widget; //NULL in case of error
18 }

```

The widget will then be shown if the user created it by the GUI or the widget is wrapped by an instance of the python class **ui** (part of module **itom**). Then you can interact with elements of the widget using python like you can do it with every user interface created with **QtDesigner**. You can also connect some python methods with signals of your widget or call slots of your widget. This is also the same behaviour like dialogs have, which only have been created with **QtDesigner** and then loaded by an instance of class **ui** using python.

Publish Filter- and Widget-Methods at Initialization

The most important step in the development of an algorithm plugin is to publish all created filter- and widget-methods. By that process, the methods will be made available to **itom**, such that they can be used by the python scripting language, the GUI or other plugins. The publishing is done in the method **init** of your plugin. A exemplary implementation is as follows:

```
1  ito::RetVal MyAlgoPlugin::init(QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *param
2  {
3      ItomSharedSemaphoreLocker locker(waitCond);
4
5      ito::RetVal retval = ito::retOk;
6      FilterDef *filter = NULL;
7      AlgoWidgetDef *widget = NULL;
8
9      //publish your filter-methods here, example:
10     filter = new FilterDef(filter1, filter1Params, tr("description"), ito::catNone, ito::iNotSpecified);
11     m_filterList.insert("filterName", filter);
12
13     //publish your dialogs, main-windows, widgets... here, example:
14     widget = new AlgoWidgetDef(widget1, widget1Params, tr("description"), ito::AddInAlgo::catNone);
15     m_algoWidgetList.insert("widgetName", widget);
16
17     if (waitCond)
18     {
19         waitCond->returnValue = retval;
20         waitCond->release();
21     }
22
23     return retval;
24 }
```

For registering filter- and widget-methods, you have to create a new instance of the classes **FilterDef** or **AlgoWidgetDef** respectively and insert these newly created instances to the maps **m_filterList** or **m_algoWidgetList** respectively. Their name in the map finally is the name of the filter or widget and must be unique within **itom**; else the filter-method or widget-method can not be loaded at startup of **itom**.

The constructor of class **FilterDef** has the following arguments:

- **AddInAlgo::t_filter filterFunc** is the pointer to the static filter-method (*here: filter1*)
- **AddInAlgo::t_filterParam filterParamFunc** is the pointer to the corresponding parameter method (*here: filter1Params*)
- **QString description** is a description string of the filter
- **ito::AddInAlgo::tAlgoCategory category** is an optional value of the category enumeration, the filter belongs too (*default: ito::AddInAlgo::catNone*)
- **ito::AddInAlgo::tAlgoInterface interf** is an optional value of the interface enumeration, the filter fits to (*default: ito::AddInAlgo::iNotSpecified*)
- **QString interfaceMeta** is depending on the chosen interface an additional meta string (*default: QString()*)

For a full reference of class **FilterDef**, see [FilterDef](#).

The constructor of class **AlgoWidgetDef** has the following arguments:

- **AddInAlgo::t_algoWidget algoWidgetPFunc** is the pointer to the static widget-method (*here: widget1*)
- **AddInAlgo::t_filterParam algoWidgetParamFunc** is the pointer to the corresponding parameter method (*here: widget1Params*)
- **QString description** is a description string of the widget or GUI-element.
- **ito::AddInAlgo::tAlgoCategory category** is an optional value of the category enumeration, the widget-method belongs too (*default: ito::AddInAlgo::catNone*)

- `ito::AddInAlgo::tAlgoInterface` **interf** is an optional value of the interface enumeration, the widget-method fits to (*default*: `ito::AddInAlgo::iNotSpecified`)
- `QString` **interfaceMeta** is depending on the chosen interface an additional meta string (*default*: `QString()`)

For a full reference of class **AlgoWidgetDef**, see [AlgoWidgetDef](#).

For information about available algorithm categories and interfaces, see

Algorithm Categories and Interfaces

You can assign to every filter- or widget-method a certain category and/or algorithm interface, the method belongs or fits to. This is useful such that **itom** can ...

- categorize your filter or widget.
- is able to integrate the filter in specific parts of the GUI, e.g. the file load or save process or dialog.
- might use your filters in image processing chains.
- ...

Algorithm Categories

Algorithm Interfaces A programmer who wants to implement a filter or widget-method that fits to a specific algorithm interface can be forced by the interface to consider a certain set of rules. These rules can be:

- The first **n** mandatory parameters can be determined. For each of these parameters the type and optionally some constraints by adding a meta information instance to the parameter are given.
- The first **m** output parameter can be determined. Type and meta information (optional) are given as well).
- The maximum number of mandatory parameters is determined, but can also be set to infinity (`INT_MAX`).
- The maximum number of optional parameters is determined, but can also be set to infinity (`INT_MAX`).
- The maximum number of output parameters is determined, but can also be set to infinity (`INT_MAX`).
- The syntax or general form of the meta information string, added to the classes **FilterDef** or **AlgoWidgetDef** can be given.

If a filter or widget-method pretends to implement a certain interface, **itom** checks at startup if the constraints are fulfilled or if this is not the case, the filter or widget-method is rejected and cannot be used within **itom**. The reason for the rejection can be seen by open the dialog **loaded plugins...** within the **help**-menu of **itom**.

The following interfaces are available (in the enumeration `tAlgoInterface`, member of class `ito::AddInAlgo`:

iNotSpecified This is the default implementation and indicates that your filter or widget-method does not fit to any interface.

iReadDataObject Filters fitting to this interface provide the functionality to read a certain file and load its content to an instance of `dataObject`.

Argument limitations

Parameter group	Max. number of parameters
mandatory parameters	infinity
optional parameters	infinity
output parameters	0

Mandatory parameters

Parameters	Type	Description	Further limitations (meta information)
#1 dataObject	DObjPtr, In, Out	DataObject, where the file is loaded to	None
#2 filename	String, In	absolute filename of the file to load	None

Output parameters

- No -

Meta information string

This string must contain the file-filters with file-endings that this filter is able to load. Different file filters are separated by a double-semicolon (;). Each filter begins with its name (arbitrary string), followed by a space and a sequence of file-endings within a pair of brackets. Examples are:

- Images (*.bmp *.png *.jpg *.pgm)
- Bitmap (*.bmp);;JPEG (*.jpg)
- Text-Files (*.txt)

iWriteDataObject Filters fitting to this interface provide the functionality to export a dataObject with a specific file format.

Argument limitations

Parameter group	Max. number of parameters
mandatory parameters	infinity
optional parameters	infinity
output parameters	0

Mandatory parameters

Parameters	Type	Description	Further limitations (meta information)
#1 dataObject	DObjPtr, In	DataObject that should be exported	None
#2 filename	String, In	absolute filename of the file	None

Output parameters

- No -

Meta information string

This string must contain the file-filters with file-endings that this filter is able to load. Different file filters are separated by a double-semicolon (;). Each filter begins with its name (arbitrary string), followed by a space and a sequence of file-endings within a pair of brackets. Examples are:

- Images (*.bmp *.png *.jpg *.pgm)
- Bitmap (*.bmp);;JPEG (*.jpg)
- Text-Files (*.txt)

iReadPointCloud Filters fitting to this interface provide the functionality to read a certain file and load its content to an instance of pointCloud.

Argument limitations

Parameter group	Max. number of parameters
mandatory parameters	infinity
optional parameters	infinity
output parameters	0

Mandatory parameters

Parameters	Type	Description	Further limitations (meta information)
#1 pointCloud	PointCloudPtr, In, Out	PointCloud, where the file is loaded to	None
#2 filename	String, In	absolute filename of the file to load	None

Output parameters

- No -

Meta information string

This string must contain the file-filters with file-endings that this filter is able to load. Different file filters are separated by a double-semicolon (;). Each filter begins with its name (arbitrary string), followed by a space and a sequence of file-endings within a pair of brackets. Examples are:

- Images (*.bmp *.png *.jpg *.pgm)
- Bitmap (*.bmp);;JPEG (*.jpg)
- Text-Files (*.txt)

iWritePointCloud Filters fitting to this interface provide the functionality to export a pointCloud with a specific file format.

Argument limitations

Parameter group	Max. number of parameters
mandatory parameters	infinity
optional parameters	infinity
output parameters	0

Mandatory parameters

Parameters	Type	Description	Further limitations (meta information)
#1 dataObject	PointCloudPtr, In	PointCloud that should be exported	None
#2 filename	String, In	absolute filename of the file	None

Output parameters

- No -

Meta information string

This string must contain the file-filters with file-endings that this filter is able to load. Different file filters are separated by a double-semicolon (;). Each filter begins with its name (arbitrary string), followed by a space and a sequence of file-endings within a pair of brackets. Examples are:

- Images (*.bmp *.png *.jpg *.pgm)
- Bitmap (*.bmp);;JPEG (*.jpg)
- Text-Files (*.txt)

iReadPolygonMesh Filters fitting to this interface provide the functionality to read a certain file and load its content to an instance of polygonMesh.

Argument limitations

Parameter group	Max. number of parameters
mandatory parameters	infinity
optional parameters	infinity
output parameters	0

Mandatory parameters

Parameters	Type	Description	Further limitations (meta information)
#1 pointCloud	PolygonMeshPtr, In, Out	PolygonMesh, where the file is loaded to	None
#2 filename	String, In	absolute filename of the file to load	None

Output parameters

- No -

Meta information string

This string must contain the file-filters with file-endings that this filter is able to load. Different file filters are separated by a double-semicolon (;). Each filter begins with its name (arbitrary string), followed by a space and a sequence of file-endings within a pair of brackets. Examples are:

- Images (*.bmp *.png *.jpg *.pgm)
- Bitmap (*.bmp);;JPEG (*.jpg)
- Text-Files (*.txt)

iWritePolygonMesh Filters fitting to this interface provide the functionality to export a polygonMesh with a specific file format.

Argument limitations

Parameter group	Max. number of parameters
mandatory parameters	infinity
optional parameters	infinity
output parameters	0

Mandatory parameters

Parameters	Type	Description	Further limitations (meta information)
#1 polygonMesh	PolygonMesh-Ptr, In	PolygonMesh that should be exported	None
#2 filename	String, In	absolute filename of the file	None

Output parameters

- No -

Meta information string

This string must contain the file-filters with file-endings that this filter is able to load. Different file filters are separated by a double-semicolon (;). Each filter begins with its name (arbitrary string), followed by a space and a sequence of file-endings within a pair of brackets. Examples are:

- Images (*.bmp *.png *.jpg *.pgm)
- Bitmap (*.bmp);;JPEG (*.jpg)
- Text-Files (*.txt)

Definition of new algorithm interfaces At first the categories and interfaces are determined as an enumeration value in the enumerations **ito::AddInAlgo::tAlgoCategory** and **ito::AddInAlgo::tAlgoInterface** (file **addInInterface.h**).

The constraints and rules for every interface are implemented in the methods **init** and **getTags** of class **AlgoInterfaceValidator**. Besides the syntax of the meta information string all necessary information is given in the method **init**.

The method-types **t_filter**, **t_algoWidget** and **t_filterParam** are defined by the following *typedef*:

```

typedef ito::RetVal (* t_filter) (QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *pa
typedef QWidget* (* t_algoWidget) (QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase>
typedef ito::RetVal (* t_filterParam) (QVector<ito::Param> *paramsMand, QVector<ito::Param> *param

```

Finish and close plugin

Finally, the **close** method can be implemented, those structure is usually unchanged:

```

1  ito::RetVal MyAlgoPlugin::close (ItomSharedSemaphore *waitCond)
2  {
3      ItomSharedSemaphoreLocker locker (waitCond);
4      ito::RetVal retval = ito::retOk;
5
6      if (waitCond)
7      {
8          waitCond->returnValue = retval;
9          waitCond->release();
10     }
11
12     return retval;
13 }
```

The **close**-method does nothing but immediately releasing the parameer *waitCond*. For more information about *ItomSharedSemaphore*, see [ItomSharedSemaphore](#).

FilterDef

class **ito::AddInAlgo::FilterDef**

container for publishing filters provided by any plugin

Public Functions

FilterDef()

< empty, default constructor

constructor with all necessary arguments.

FilterDef(AddInAlgo::t_filter filterFunc, AddInAlgo::t_filterParam filterParamFunc,
QString description = QString(), ito::AddInAlgo::tAlgoCategory category =
ito::AddInAlgo::catNone, ito::AddInAlgo::tAlgoInterface interf =
ito::AddInAlgo::iNotSpecified, QString interfaceMeta = QString())

~FilterDef()

Public Members

t_filter **m_filterFunc**

function pointer (unbounded, static) for filter-method

t_filterParam **m_paramFunc**

function pointer (unbounded, static) for filter's default parameter method

ito::AddInInterfaceBase * **m_pBasePlugin**

interface (factory) instance of this plugin (will be automatically filled)

QString **m_name**

name of filter

QString **m_description**

description of filter

ito::AddInAlgo::tAlgoCategory **m_category**

category, filter belongs to (default: catNone)

ito::AddInAlgo::tAlgoInterface **m_interface**

algorithm interface, filter fits to (default: iNotSpecified)

QString **m_interfaceMeta**

meta information if required by algorithm interface

AlgoWidgetDef

class ito::AddInAlgo::AlgoWidgetDef

container for publishing widgets provided by any plugin

Public Functions

AlgoWidgetDef()

< empty, default constructor

constructor with all necessary arguments.

AlgoWidgetDef(AddInAlgo::t_algoWidget algoWidgetFunc, AddInAlgo::t_filterParam
algoWidgetParamFunc, QString description = QString(), ito::AddInAlgo::tAlgoCategory
category = ito::AddInAlgo::catNone, ito::AddInAlgo::tAlgoInterface interf =
ito::AddInAlgo::iNotSpecified, QString interfaceMeta = QString())

~AlgoWidgetDef()

destructor

Public Members

t_algoWidget **m_widgetFunc**

function pointer (unbounded, static) for widget-method

t_filterParam **m_paramFunc**

function pointer (unbounded, static) for widget's default parameter method

ito::AddInInterfaceBase * **m_pBasePlugin**

interface (factory) instance of this plugin (will be automatically filled)

QString **m_name**

name of widget

QString **m_description**

description of widget

ito::AddInAlgo::tAlgoCategory **m_category**

category, widget belongs to (default: catNone)

ito::AddInAlgo::tAlgoInterface **m_interface**

algorithm interface, widget fits to (default: iNotSpecified)

QString **m_interfaceMeta**

meta information if required by algorithm interface

7.3.17 Create a new plugin via CMake

Plugins for **itom** are also created using **CMake**. Therefore the sources and the project files of the plugins will also be separated, like it is also the case for **itom**, the designer plugins and all other plugins. You can either put your source files in any subfolder of the location where all sources of your plugins lie or you can put it at any arbitrary location on your harddrive.

The source folder of your plugin mainly consists of these files:

1. **CMakeLists.txt**. This is the project file of your plugin, where you insert the files that are included in your plugin or libraries, the plugin should link with. This file is finally interpreted by **CMake** in order to create the real project files, adapted to your generator (e.g. Visual Studio).
2. **yourPlugin.h** and **yourPlugin.cpp**. This are the main header and source files of your plugin.
3. **dialogYourPlugin.h** and **dialogYourPlugin.cpp** (optional, not for algorithms). Use these files if you want to provide a configuration dialog for your plugin (can also be added later).
4. **dockWidgetYourPlugin.h** and **dockWidgetYourPlugin.cpp** (optional, not for algorithms). Use these files if you want to provide a dock widget of your plugin that is inserted into **itom**'s main window (can also be added later).
5. **pluginVersion.h** (optional). This header contains some defines for your current plugin version. It should be included in **yourPlugin.cpp** and under MSVC / windows in **version.rc**.
6. **version.rc** (optional, only under MSVC / windows). Under windows the content of this file will be automatically added to the meta-Data of your DLL.

Templates

You will find templates for the most important files needed for creating a new plugin in the folder **pluginTemplates** of the SDK-directory of **itom**.

Copy the file **CMakeLists.txt** from this template-directory and copy it to the source directory of your plugin. Open this file with an arbitrary editor. The commands, starting with #, give you hints where you need to adapt this file. Further details about the syntax of these files can be found under <http://www.cmake.org/cmake/help/documentation.html>.

Furthermore, you will find template implementations of an actuator, algorithm and camera plugin in the **plugin-Templates** folder. Copy the corresponding header and source file of your type into your source directory and start replacing the template strings by your versions, e.g. you need to change the name of your plugin hence the class names.

Generate project

If you placed the source files of your plugin into a subdirectory of an existing multi-plugin project, then you need to add the subfolder of your plugin into the file **CMakeLists.txt** of the parent-folder. This file usually already

contains a lot of subdirectories, added by the CMake-command **ADD_SUBDIRECTORY** and has the following form:

```
project(itom_plugins) #name of the overall project

cmake_minimum_required(VERSION 2.8)

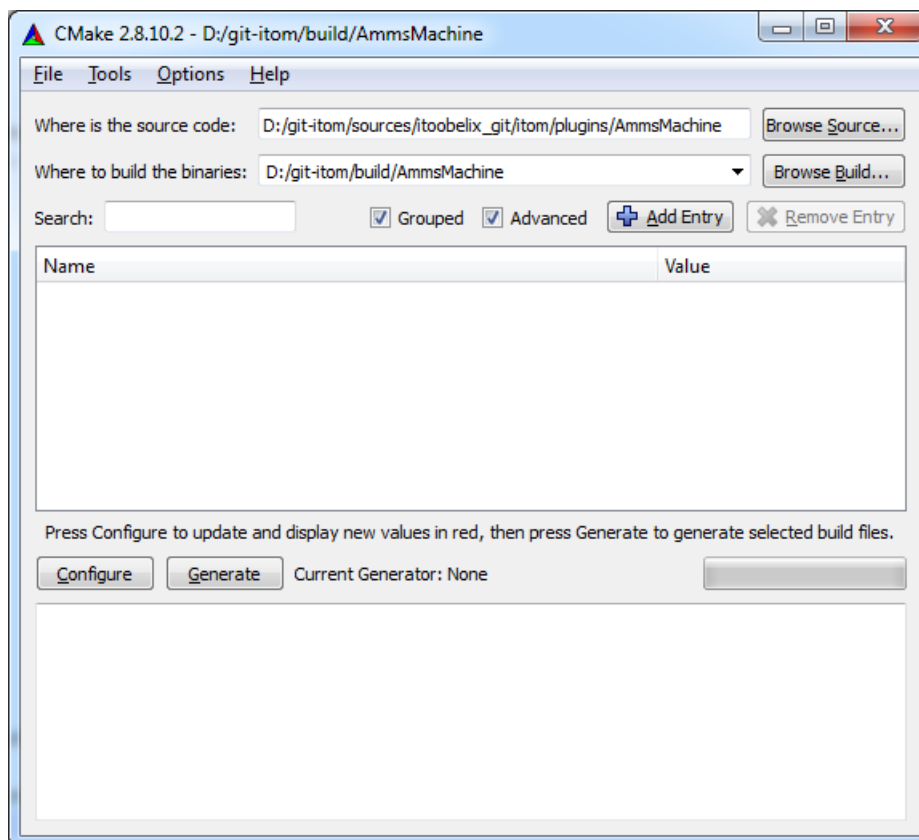
OPTION(BUILD_UNICODE "Build with unicode charset if set to ON, else multibyte charset." ON)
OPTION(BUILD_SHARED_LIBS "Build shared library." ON)
OPTION(BUILD_TARGET64 "Build for 64 bit target if set to ON or 32 bit if set to OFF." OFF)
SET(ITOM_DIR "" CACHE PATH "base path to itom")

# Insert the following section for your plugin

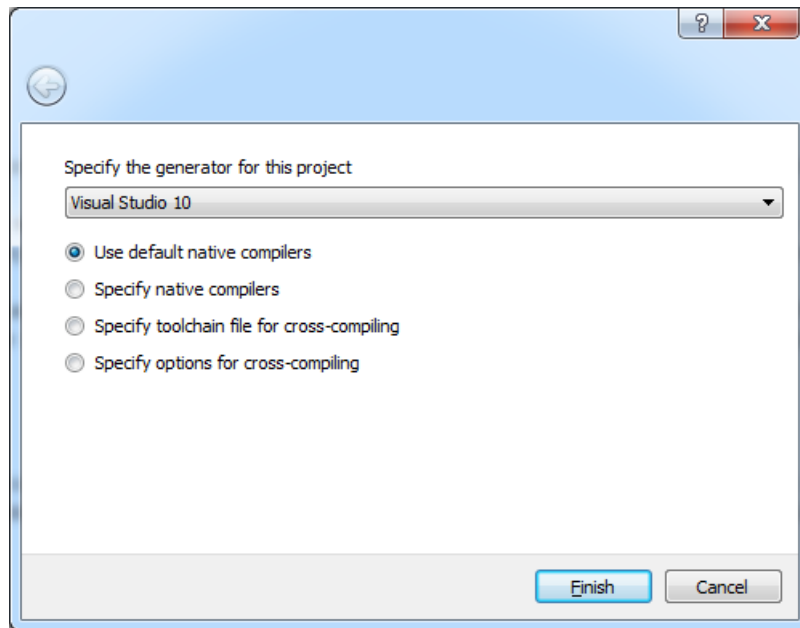
# by this option, the plugin can be unchecked in order to
# avoid its generation in cmake.
OPTION(PLUGIN_YOURNAME "Build with this plugin." ON)
if (PLUGIN_YOURNAME)
    ADD_SUBDIRECTORY(YOURSUBDIRECTORY)
endif (PLUGIN_YOURNAME)
```

Then, reconfigure and regenerate the overall plugin project or simply run the project **ZERO_CHECK** of your overall plugin solution such that the new plugin is generated and added to the overall solution. If you have no idea about configuring and generating a **CMake** project, continue reading or check the big [CMake-section](#) about generating **itom** itself.

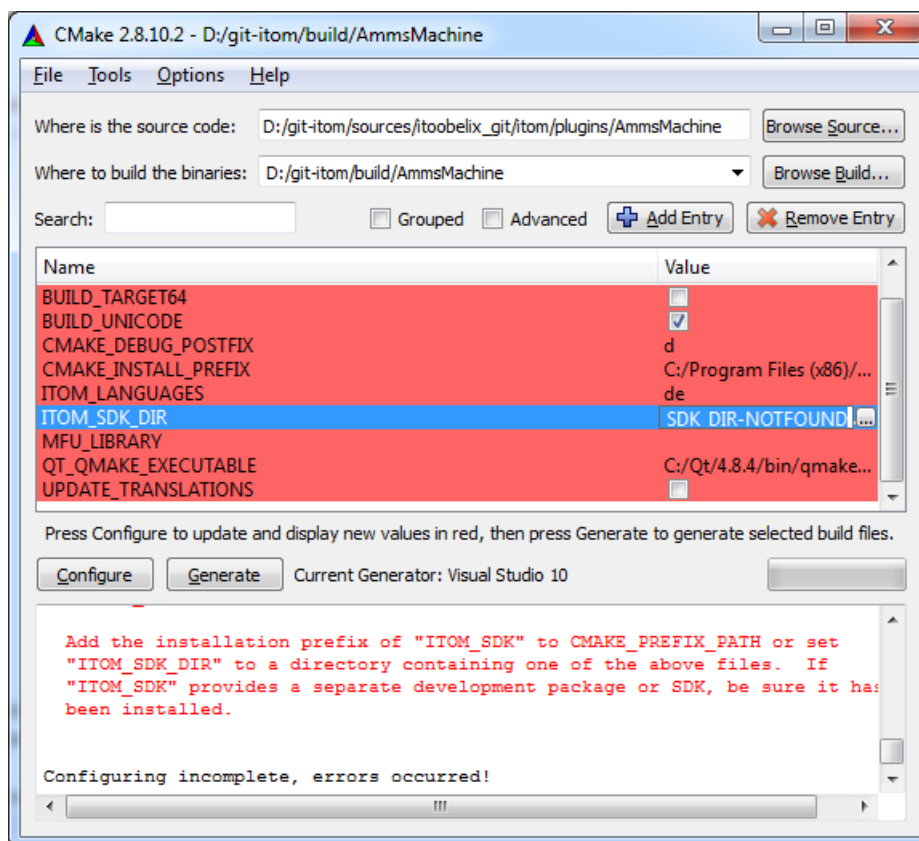
In order to generate your plugin as single solution, open the **CMake GUI** and choose the source directory of your plugin as source directory and any arbitrary folder as build directory. The solution and project files are then generated in this build directory.



Then click the configure button to start the configuration. At first, you will be asked for a generator. See [Build with CMake](#) for more information about generators.



Then, another configuration process is started. Usually, it is now necessary to set the variable **ITOM_SDK_DIR** to the directory of **itom**'s SDK, usually located in **itom**'s build directory. Then click **configure** again.



Next, you probably need to indicate the location of the build-directory of **OpenCV** on your harddrive (variable **OpenCV_DIR** in the group **OpenCV** or **Ungrouped Entries** if you activated the **Grouped** checkbox).

Continue clicking **configure** until there are no more warnings. Then you can press **Generate** in order to build the project files. Change then to the build-directory and open the recently build solution file. In you afterwards change the **CMakeLists.txt**-file, you don't need to explicitly run the CMake-GUI again, since Visual Studio is also able to directly run CMake and update its project files.

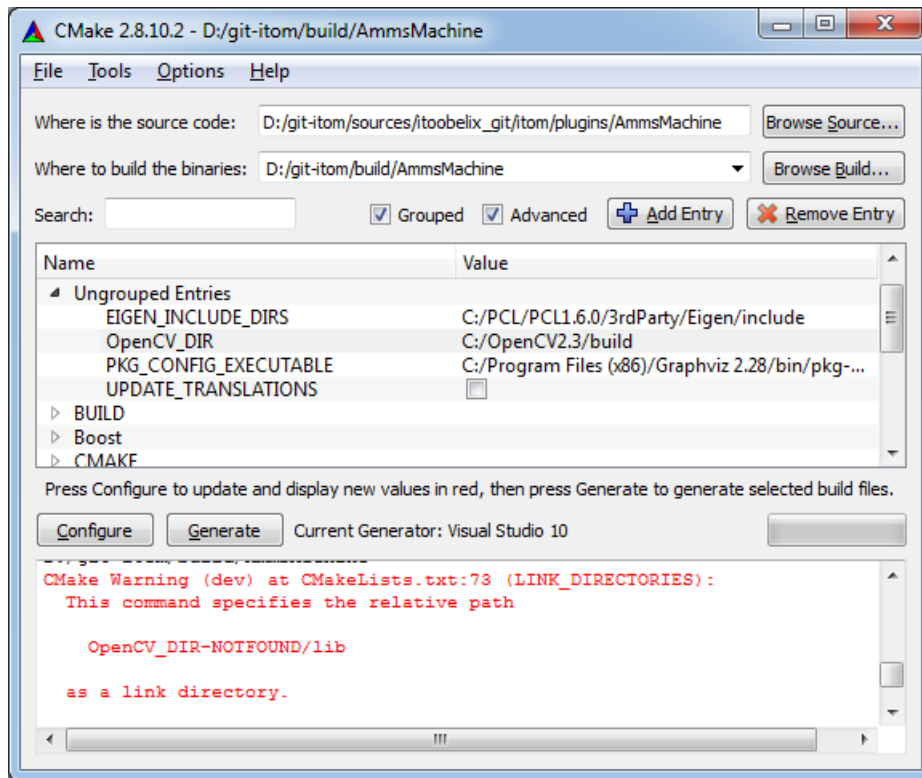
7.3.18 Automatic loading and saving of plugin parameters

itom has the optional ability, that all plugin parameters (type *tParam*), which are part of the **m_params** vector of the plugin-class and do not have the *typeNoAutoSave*-flag defined, can be stored in a plugin-specific xml-file when the plugin instance is closed. This saving is not only dependent on the plugin but also on its unique identifier. **itom** has the optional ability, that all plugin parameters (of type **Param** or **ParamBase**), which are part of the **m_params** vector of the plugin-class and do not have the *typeNoAutoSave*-flag defined, can be stored in a plugin-specific xml-file when the plugin instance is closed. This saving is not only dependent on the plugin but also on its unique identifier.

Additionally, the decision whether these parameters should be saved or not, is set by the member variable **m_autoSavePolicy** of the interface class of the plugin (see *Plugin interface class*). This variable can have the following values:

```
enum tAutoSavePolicy {
    autoSaveAlways      = 0x1, /*!< always saves parameters to xml-file at shutdown */
    autoSaveNever       = 0x2  /*!< never saves parameters to xml-file at shutdown (default)
};
```

If the member has the value *autoSaveAlways*, all parameters contained in the vector *m_params* of the plugin instance are saved in a xml-subtree. This subtree is dependent on the unique identifier of the plugin. Remember, that only plugins which do not have the flag *typeNoAutoSave* will be saved (see *Parameter-Container class of itom*).



Inversely, saved parameters in a plugin specific xml-file can also be set after that the plugin instance has been created and initialized (with the mandatory and optional parameters given by the constructor in python). The loading of these xml-parameters is dependent on the value of the member variable **m_autoLoadPolicy** of the plugin's interface class. This variable can have the following values:

```
enum tAutoLoadPolicy {
    autoLoadAlways          = 0x1, /*!< always loads xml file by addInManager */
    autoLoadNever           = 0x2, /*!< never automatically loads parameters from xml-file (default) */
    autoLoadKeywordDefined = 0x4, /*!< only loads parameters if keyword autoLoadParams=1 exists */
};
```

If the variable is set to *autoLoadAlways*, all parameters in the xml-subtree, having the same unique identifier (ID) than the recently opened plugin instance, are set in the instance. This is done by calling the method **setParam** of the instance for every specific parameter. It is up to you to decide whether you accept the given parameter from the xml-file or not. Remember that **setParam** is called after that the plugin instance is created and the init-method of the plugin has been called. Of course, the init-method gets the mandatory and optional parameters, which are indicated when you create the plugin instance for example in python.

If the variable is set to *autoLoadNever*, no parameters of the xml-file will be set to the plugin's instance.

If the variable is set to *autoLoadKeywordDefined*, the parameters from the xml-subtree are only set to the plugin's instance, if the plugin has been created in python with the last keyword-defined parameter **autoLoadParams=1**.

Example:

```
a = dataIO("DummyGrabber", param1, ..., paramN, autoLoadParams=1)
```

7.3.19 itom API

Usually, plugins only have access to the sources which are defined in the **SDK** of **itom**. These are for instance the header and source files contained in the folder **common** (e.g. **addInInterface.h**) and in the folder **plot** (only relevant for developing designer plugins (hence plots, figures...)). Further, the **SDK** contains the header-files and corresponding libraries for the **dataObject** and the **pointCloud** library.

However, it is not desired at all, that plugins also include header or source files from **itom** itself. The intention is, that it should be possible to develop plugins without the need to compile **itom** from its sources or have the source code available.

Therefore, **itom** provides an **application programming interface** (API). The API has methods defined, which give plugins the possibility to use important functionality of **itom**. All available methods of the **API** are defined in the file **apiFunctionsInc.h**, that also lies in the folder **common** of the SDK. For methods concerning plots and figures, there is an additional API definition file **apiFunctionsGraphInc.h**.

Initialization

If your plugin or designer plugin widget is derived from classes **AddInBase** or **AbstractFigure**, which is usually the case, you can use the **API** methods without further steps or includes.

When you are programming a plugin, derived from **AddInBase**, you can access any **API**-method at any time in your plugin (even in the constructor). When programming a designer plugin widget that is handled as plotting plugin (derived from **AbstractFigure**) the APIs only become valid after the construction of your plugin.

Note: Due to the software structure, the valid pointer is transmitted by **itom** sending the event with type **QEvent::User+123**, that is caught by your plugin after the construction.

Usage

You can call the methods, defined in **apiFunctionsInc.h** or **apiFunctionsGraphInc.h** like normal function calls. In the following example, the filter **saveRPM**, which is defined in another plugin should be called. Since, the mandatory, optional and output parameters of this filter are unknown, we will first request their default values, then change their values and finally we call the filter-method. If you have knowledge about the parameters, you just can implement the second part.

```
ito::RetVal retVal;
QVector<ito::ParamBase> mandParams, optParams, outParams; //define three, empty parameter vectors

//now get default values
retVal = apiFilterParamBase("saveRPM", &mandParams,&optParams,&outParams);
//retVal is retOk if the filter 'saveRPM' could be found. If the external plugin does not exist, it returns retErr

//change the value of some values
mandParams[0].setVal<void*>( yourDataObjectPtr ); //set data object to save as first mandatory parameter
mandParams[1].setVal<char*>( "C:\\test.rpm" );

//now call the filter
retVal += apiFilterCall("saveRPM", &mandParams, &optParams, &outParams);

if(!retVal.containsError())
{
    //filter could be successfully executed
    //you can now evaluate the output parameters, if the filter provided some.
}
```

7.3.20 Link or load external libraries

In your plugin, you have different possibilities to use external or 3rd party libraries:

Link to a static library

If you add an appropriate entry to the linker settings of your project file, it is possible to link to external, static libraries. This is for example the case when linking to the **dataObject.lib** or **dataObject.so** (Linux) or if using

any components of the *PointCloudLibrary*. When compiling your plugin, the whole implementation of the library is compiled into your plugin-library and you don't need to distribute the external file.

Link to a shared library

You can also link to an external shared library by adding the corresponding entry to the linker settings, too. This is for example the case when using *OpenCV*-methods, which use implicitly do, if you link against the dataObject. Then you must add the corresponding **.lib** or **.so** files to your linker-settings and provide the corresponding **.dll** or **.a** file (in debug or release, if possible). Then, you have to take care that **itom** is able to find the external library. Please consider, that this external library file is not detected relatively to the location of your plugin but relatively to the executable of **itom** itself. Therefore, you have these possibilities to distribute the external library file:

- Add its containing folder to the path variable of your operating system.
- Add it to any path which already is contained in the path variable of your OS (e.g. **system32** on Windows).
- Directly add it to the **itom**-folder (not recommended, since leads to "crazy" folder structure)
- Directly add it to the folder **lib** contained in the **itom**-folder. The *lib**-folder is added to the path variables that are passed to **itom** at startup. This is a good possibility to provide the external library file, however it can also lead to conflicts with other plugins, that need the same external library, however in an other version. Therefore check whether you can share the same version with other plugins. In the default implementation of **itom**, there is also some default libraries of **Glut**, **FFTW**... that should be used.
- Try to indicate the shared library as delay-loaded module. Then you also adapt the path variable of your plugin to a folder of your choice, before the plugins tries to load the shared library. This is a conflict-free way how to access shared libraries. Let's make an example: Your plugin **MyPlugin** lies after compilation in the folder **plugin/myPlugin**, that is a subfolder of **itom**. Then put your external shared library file in the subfolder **lib** of **plugin/myPlugin**. Then add this path to the current path variable of the application by adding the following code for instance in the constructor or the **init**-method of your plugin implementation:

```
// Get the path to the plugin directory
QDir dllDir = QApplication::applicationDirPath();
if( !dllDir.cd("plugins/MyPlugin") )
{
    dllDir.cdUp();
    dllDir.cd("plugins/MyPlugin");
}
dllDir.cd("lib"); //move to lib folder
QString dllDir2 = QDir::cleanPath(dllDir.filePath(""));

// Add the plugin path to the path environment variable
char *oldpath = getenv("path");
char *newpath = (char*)malloc(strlen(oldpath) + dllDir2.size() + 10);
newpath[0] = 0;
strcat(newpath, "path=");
strcat(newpath, dllDir2.toAscii().data());
strcat(newpath, ";");
strcat(newpath, oldpath);
_putenv(newpath);
free(newpath);
```

Note: The path variable of your operating system is always copied and then passed to a newly started application. Therefore you can adapt this copy without influencing the overall path environment variable.

Load external library at runtime

The most complicated way to access an external library with respect to the programming cost is to use the command **LoadLibrary** or the platform-independent Qt-class **QLibrary** in order load an external library at runtime of your plugin. Then you need to resolve the symbols in the library in order to access them afterwards in a

function-call. The advantage of this method however is, that the library can be at any location since you are able to load the library with its absolute filename. See the **Qt** documentation for details about the class **QLibrary**, that is recommended to use.

Note: If you link to external libraries, please consider always the license requirements of the external library.

7.3.21 Project settings for plugins

Since you are building your plugins using **CMake** most of the following settings are automatically set. However, this document gives some hints about properties, which can maybe be helpful in case of any problems.

Prerequisites

For programming a plugin, you need at least the following things:

- A C++-compatible IDE. On Windows-machines it is recommended to program with *Visual Studio Professional 2010*, since **itom** is developed with this IDE, too. On Linux-machines you can for instance use the *QtCreator* or *Eclipse*. It is difficult to develop with *Visual Studio 2010 Express*, since you should install the *Qt AddIn for Visual Studio* in order to have a good support of **Qt** within *Visual Studio*. In the case you don't have the professional version of *Visual Studio*, better consider to use the *QtCreator* for Windows. You must have the **Qt** version installed, whose major and minor version number is equal than the version **itom** has been built with. Nevertheless the debugging of your plugin only is possible if you also have a debug-version of **itom** available on your computer, hence, you built it by yourself from the sources. Else, you only can test your plugin by extensively streaming debugging messages to the `std::cerr` or `std::cout` stream, which finally are displayed in the command line of **itom**.
- For running *itom*, you need *Python 3.2* installed on your machine.
- If you want to support the *itom*-internal *DataObject* (matrix structures), it is highly recommended to install *OpenCV2.3* on your machine.

Please consider to have all libraries, which you need, installed in the same version with respect to the processor type (32bit or 64bit).

General settings

- compile your plugin as dynamic library (*dll* or *a* on linux).
- for the code generation use as runtime library the setting **Multithreaded-DLL (/Md)** or **Multithreaded-Debug-DLL (/MDd)** respectively.
- Don't use any precompiled headers on Windows.
- You can switch *openMP* on in order to support multi-processor calculations for parallelizable algorithms.
- Call your DLL "[yourName].dll" for the release-version and "[yourName]d.dll" for the debug-version.

Qt-dependent settings

itom is written in C++ using the **Qt**-framework. **Qt** provides platform-independent modules and classes which extend the possibilities of native C++. For example, **Qt** gives the opportunity to build GUI-applications, have network and graphics support or to establish a platform-independent plugin system.

On the one hand, some functionalities of **Qt** can be used by the help of native Qt-applications, like the designer to build "what-you-see-is-what-you-get" user interfaces, the translator to create translations of the application..., on the other hand C++ is enlarged by **Qt** by writing specific pre-compiler commands in the code. In both cases, these features have to be translated into native C++-code during the pre-compiling process. Therefore the project files have to be adapted, such that the **Qt**-specific pre-steps will be triggered once the project's compilation process is started. All this is done if you install the **Qt-AddOn** for Visual Studio (if developing with Visual Studio IDE).

Since the plugin, you will write, is based on *Qt*'s plugin system, these steps also have to be added to the plugin's pre-compiling steps. This can be realized by different ways:

1. You use the **QtCreator** as IDE and everything works fine (if the path to **Qt** is contained in the path variable and the environmental variable *QTDIR*)
2. You can use the professional version of Visual Studio together with the installed add-in **Qt Visual Studio Add-In**
3. You can use any other development environment and you have to add the necessary pre-compilation step by yourself in the appropriate project file.

The pre-processor-step contains the following steps:

1. In a folder "generated-files" additional files will be created for each class, containing the macro *Q_OBJECT* (moc-process).
2. Any user-interface file (*.ui) will be transformed into an additional C++-class file, that is also contained in the "generated-files" folder (uic-process).
3. The translation tables will be created.
4. The resource-files will be parsed and an appropriate C++-file is created (rcc-process).

Qt is shipped with a number of different libraries (lying in the folder **\$QTDIR\$bin**). You must link your application against the libraries, whose function you will need in your plugin. It is always necessary to link against the library **QtCore** and **QtGui** if your plugin contains any user interface functionality. Other important libraries are **QtOpenGL** for OpenGL-support, **QtSvg** for Svg-support or **QtXml**, **QtSql** or **QtNetwork**. For each of these libraries your plugin must have an entry in the *include*-directories and the *linker*-commands.

The pre-processor-definitions must contain the following entries: * WIN32 or _WIN64 * QT_LARGEFILE_SUPPORT * QT_PLUGIN * QT_DLL * QDESIGNER_EXPORT_WIDGETS

and for every Qt-library you need (in capital letters) * QT_CORE_LIB * QT_GUI_LIB * ...

Include settings

Add the following include-directories (\$(ITOM_QTDIR) is the path to the **Qt**-source directory - having subfolders like include or bin):

- .GeneratedFiles
- .GeneratedFilesDebug or Release
- \$(ITOM_QTDIR)include
- \$(ITOM_QTDIR)includeqtmain

and for every further Qt-library * \$(ITOM_QTDIR)includeQtCore * \$(ITOM_QTDIR)includeQtGui * ...

Additionally you should add the path to OpenCV's include directory, if you are using or linking against the *DataObject*.

Linker settings

Add as linker directory:

- The directory, where itom is saving the library to the dataObject.lib...
- The library-path of OpenCV
- The directory \$(ITOM_QTDIR)bin

Your plugin should at least link against the following libraries:

DEBUG:

- qtmaind.lib

- QtCored4.lib
- QtGui4.lib
- ...further Qt libraries
- opencv_core\$(ITOM_OPENCV_VER)d.lib
- DataObjectd.lib
- ...

RELEASE:

- qtmain.lib
- QtCore4.lib
- QtGui4.lib
- ...further Qt libraries
- opencv_core\$(ITOM_OPENCV_VER).lib
- DataObject.lib
- ...

Note: For more information about the deployment of plugins, including notes about the **Qt**-version compatibility, see [link to Qt-documentation](#)

PYTHON SCRIPTING LANGUAGE

8.1 Introduction

One main component of **itom** is the integrated scripting language **Python**. In order to make **itom** ready for the future, **Python** is based on the version-family 3. If you already know **Python** or already have used **Python**, you probably know about the huge functionality which already is provided by **Python** itself or by one of its various modules, that are available in the internet. The good is, that you can execute the same **Python**-scripts in **itom**. There are only few limitations concerning some **Python** modules (see *Python Limitations in itom*).

If you are a **Python** beginner, you probably should have a look at our tutorial page that also contains links to recommended tutorials in the internet:

```
it> >.. include:: /include/global.inc
```

8.1.1 Python tutorial

The goal of this tutorial is to give prospective ITOM users a quick overview about the basic **Pythonv3** commands and how to use them within **litoml**.

Content

```
it> >.. include:: ../include/global.inc
```

Creating a project folder and run a first program

Create a project folder

Keeping track of more difficult programs is inconvenient when only using the **litoml** terminal. That's why you will learn how to use script files in this chapter. Basically, script files are just regular text files that happen to have the extension *.py*.

We will store all our script files in a project folder within the **litoml** installation directory called *python_tutorial*. This folder can be created within **litoml** by right-clicking on the **litoml** folder in the File System Panel and choosing *create new folder*. If you've used the default installation directory, you should end up with a folder like this: *C:\Program Files\iTOM\python_tutorial*.

Creating a .py file and running it

Now create a new python file in *C:\Program Files\iTOM\python_tutorial* and rename the default file name (*New Script.py*) to *Hello World.py*. When double-clicking the file name, the Script Editor will pop up, which we will use for all upcoming examples. For a start, type the following line of code into the Script Editor and save the file.

```
1 print("Hello World")
```

To run the code you’ve just inserted, press *F5* or click on the *run* button.

As a result, you will see the output ‘Hello World’ in the **itoml** terminal. As you’ve probably figured out, the command `print()` is used to output certain characters and, hence, can be used to give the user some sort of feedback. Please note that the print command changed from earlier 2.X versions of **Python**.

```
ï»¿.. include:: ../../include/global.inc
```

Variables and names

Using variables

Now you can output things with `print()` and you can do math. The next step is to learn about variables. In programming, a variable is nothing more than a name for something and helps making the code easier to read. In **Python** the equal sign (=) is used to assign a value to a variable.

```
1 width = 10
2 depth = 5
3 height = 20
4
5 area_size = width * depth
6 volume = area_size * height
7
8 # Variables can be output with print()
9 print(area_size)
10
11 # The output of strings and variables can be combined
12 print("Volume:", volume)

50
Volume: 1000
```

A value can be assigned to several variables simultaneously as seen below. Additionally, you will learn how to make strings that have variables embedded in them. You embed variables inside a string by using specialized format sequences (in this case %d) and then putting the variables at the end with a special syntax.

```
1 width = depth = height = 10
2
3 print("The edge lengths of the cube are %d, %d and %d, respectively." % (width, depth, height))
4 print("Still the same volume: %d" % (width * depth * height))
```

```
The edge lengths of the cube are 10, 10 and 10, respectively.
Still the same volume: 1000
```

Note: Remember to put `# -- coding: utf-8 --` at the top of your script file if you use non-ASCII characters and get an encoding error.

```
ï»¿.. include:: ../../include/global.inc
```

Strings and text

Strings

Besides numbers, **Python** can also manipulate strings, which can be expressed in several ways. A string is usually a bit of text you want to display to someone, or “export” out of the program you are writing. They can be enclosed in single quotes or double quotes:


```

1 print('It works.')
2 print('It doesn\'t matter.')
3 print("It doesn't matter.")
4 print(' "Yes", he said.')
5 print("\ "Yes,\" he said.")
6 print(' "It\'s not", she said.')

```

```

It works.
It doesn't matter.
It doesn't matter.
"Yes", he said.
"Yes," he said.
"It's not", she said.

```

String formatting

Additionally, Strings may contain format characters such as `%d` from the previous section to output or convert to integer decimals. Here are some more:

Conversion	Meaning
<code>%d</code> or <code>%i</code>	Signed integer decimal
<code>%x</code> or <code>%X</code>	Signed hexadecimal (lowercase/uppercase)
<code>%e</code> or <code>%E</code>	Floating point exponential (lowercase/uppercase)
<code>%f</code> or <code>%F</code>	Floating point decimal format
<code>%c</code>	Single character (accepts integer or single character string)
<code>%r</code>	String (converts any Python object using <code>repr()</code>)
<code>%s</code>	String (converts any Python object using <code>str()</code>)

Here are some examples to try for yourself:

```

1 x = "There are %d types of people." % 10
2 binary = "binary"
3 do_not = "don't"
4 y = "Those who know %s and those who %s." % (binary, do_not)
5
6 print(x)
7 print(y)
8
9 print("I said: %r." % x)
10 print("I also said: '%s'." % y)

```

```

There are 10 types of people.
Those who know binary and those who don't.

```

Note: The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). Using the newer `str.format()` interface helps avoiding these errors. Check the official [documentation](#) for more on the topic.

More on strings

String literals can span multiple lines in several ways. Continuation lines can be used, with a backslash as the last character on the line indicating that the next line is a logical continuation of the line:

```

1 hello = "This is a rather long string containing\n\
2 several lines of text just as you would do in C.\n\
3     Note that whitespace at the beginning of the line is \
4 significant."

```

```
5
6 print(hello)
```

This is a rather long string containing several lines of text just as you would do in C. Note that whitespace at the beginning of the line is significant.

Or, strings can be surrounded in a pair of matching triple-quotes: `"""` or `'''`. End of lines do not need to be escaped when using triple-quotes, but they will be included in the string. So the following uses one escape to avoid an unwanted initial blank line.

```
1 print("""\
2 Usage: thingy [OPTIONS]
3     -h                Display this usage message
4     -H hostname       Hostname to connect to
5 """)
```

```
Usage: thingy [OPTIONS]
    -h                Display this usage message
    -H hostname       Hostname to connect to
```

If we make the string literal a “raw” string, `n` sequences are not converted to newlines, but the backslash at the end of the line, and the newline character in the source, are both included in the string as data. Thus, the example:

```
hello = r"This is a rather long string containing\n\
several lines of text much as you would do in C."
```

```
print(hello)
```

```
"This is a rather long string containing\n\several lines of text much as you would do in C."
```

Byte array....

```
.)
```

```
ï»¿.. include:: ../../include/global.inc
```

Dictionaries, Lists and Tuples

Dictionaries

One of Python’s built-in datatypes is the dictionary, which defines one-to-one relationships between keys and values.

Defining Dictionaries It is best to think of a dictionary as an unordered set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{ }`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

```
1 d = {"lens": "zoom", "software": "itom"}
2 print(d)
3
4 # 'lens' is a key and its associated value is 'zoom'. It can be referenced by d["lens"].
5 print(d["lens"])
6
7 # 'software' is a key and its associated value is 'itom'. It can be referenced by d["software"].
8 print(d["software"])
9
10 # You can get values by key, but you can't get keys by value.
```

```

11 # So d["lens"] is 'zoom', but d["zoom"] raises an exception, because 'zoom' is not a key.
12 print(d["zoom"])

{'lens': 'zoom', 'software': 'itom'}
zoom
itom
Traceback (most recent call last):
  File "..", line ?, in <module>
KeyError: 'zoom'

```

Modifying Dictionaries You can not have duplicate keys in a dictionary. Assigning a value to an existing key will wipe out the old value. You can add new key-value pairs at any time. This syntax is identical to modifying existing values. Dictionaries have no concept of order among elements.

```

1 d = {"lens": "zoom", "software": "itom"}
2 print(d)
3
4 d["software"] = "itom"
5 print(d)
6
7 d["institute"] = "ITO"
8 print(d)
9
10 # Dictionary keys can be modified. The old value is simply replaced with a new one.
11 d["institute"] = "none"
12 print(d)
13
14 # Dictionary keys are case-sensitive. Here, a new key-value pair will be created.
15 d["Institute"] = "another"
16 print(d)

{'software': 'itom', 'lens': 'zoom'}
{'software': 'itom', 'lens': 'zoom'}
{'software': 'itom', 'institute': 'ITO', 'lens': 'zoom'}
{'software': 'itom', 'institute': 'none', 'lens': 'zoom'}
{'Institute': 'another', 'software': 'itom', 'institute': 'none', 'lens': 'zoom'}

```

Dictionaries aren't just for strings. Dictionary values can be any datatype, including strings, integers, objects, or even other dictionaries. Within a single dictionary, the values don't all need to be the same type; you can mix and match as needed. Dictionary keys are more restricted, but they can be strings, integers, and a few other types. You can also mix and match key datatypes within a dictionary.

```

1 d = {'lens': 'zoom', 'institute': 'ITO', 'software': 'itom'}
2 print(d)
3
4 d["version"] = 3
5 print(d)
6
7 d[42] = "douglas"
8 print(d)

{'lens': 'zoom', 'institute': 'ITO', 'software': 'itom', 'version': 3}
{'lens': 'zoom', 'institute': 'ITO', 'software': 'itom', 42: 'douglas', 'version': 3}

```

Deleting Items From Dictionaries

```

1 d = {'lens': 'zoom', 'institute': 'ITO', 'software': 'itom', 42: 'douglas', 'retrycount': 3}
2 print(d)
3
4 # delete individual items from a dictionary by key
5 del d[42]

```

```
6 print(d)
7
8 # delete all items from a dictionary
9 # Note that the set of empty curly braces in the output signifies a dictionary without any items.
10 d.clear()
11 print(d)

{'lens': 'zoom', 'institute': 'ITO', 'software': 'itom', 42: 'douglas', 'retrycount': 3}
{'lens': 'zoom', 'institute': 'ITO', 'software': 'itom', 'retrycount': 3}
{}
```

Lists

Lists are Python's workhorse datatype. Variables can be named anything, and Python keeps track of the datatype internally.

Defining Lists A list is an ordered set of elements enclosed in square brackets. It can be used like a zero-based array.

```
1 # A list of five elements is defined. Note that they retain their original order.
2 li = ["a", "b", "zoom", "z", "example"]
3 print(li)
4
5 # The first element of any non-empty list is always li[0]
6 print(li[0])
7
8 # The last element of this five-element list is li[4].
9 print(li[4])

['a', 'b', 'zoom', 'z', 'example']
a
example
```

Negative List Indices A negative index accesses elements from the end of the list counting backwards. The last element of any non-empty list is always `li[-1]`. If the negative index is confusing to you, think of it this way: `li[-n] == li[len(li) - n]`.

```
1 li = ["a", "b", "zoom", "z", "example"]
2 print(li)
3 print(li[-1])
4
5 # In this list, li[-3] == li[5 - 3] == li[2]
6 print(li[-3])

['a', 'b', 'zoom', 'z', 'example']
'example'
'zoom'
```

Slicing a List You can get a subset of a list, called a *slice* by specifying two indices. The return value is a new list containing all the elements of the list, in order, starting with the first slice index up to but not including the second slice index.

If it helps, you can think of it this way: reading the list from left to right, the first slice index specifies the first element you want, and the second slice index specifies the first element you don't want. The return value is everything in between.

```

1 li = ["a", "b", "zoom", "z", "example"]
2 print(li)
3
4 # A slice of li[1] up to but not including li[3] will be created
5 print(li[1:3])
6
7 # Slicing works if one or both of the slice indices is negative.
8 print(li[1:-1])
9
10 # Lists are zero-based, so li[0:3] returns the first three elements of the list, starting at li[0].
11 print(li[0:3])
12
13 # If the left slice index is 0, you can leave it out, and 0 is implied. So li[:3] is the same as li[0:3].
14 print(li[:3])
15
16 # Similarly, if the right slice index is the length of the list, you can leave it out. So li[3:] is the same as li[3:len(li)].
17 print(li[3:])
18
19 # If both slice indices are left out, all elements of the list are included.
20 # li[:] is shorthand for making a complete copy of a list.
21 print(li[:])

['a', 'b', 'zoom', 'z', 'example']
['b', 'zoom']
['b', 'zoom', 'z']
['a', 'b', 'zoom']
['a', 'b', 'zoom']
['z', 'example']
['a', 'b', 'zoom', 'z', 'example']

```

Adding Elements to Lists

```

1 li = ["a", "b", "zoom", "z", "example"]
2 print(li)
3
4 # Single elements can be added to the end of the list with append
5 li.append("new")
6 print(li)
7
8 # insert inserts a single element into a list. The numeric argument is the index of the first element to be inserted.
9 li.insert(2, "new")
10 print(li)
11
12 # Lists can be concatenated with extend. Note that you do not call extend with multiple arguments.
13 li.extend(["two", "elements"])
14 print(li)

['a', 'b', 'zoom', 'z', 'example']
['a', 'b', 'zoom', 'z', 'example', 'new']
['a', 'b', 'new', 'zoom', 'z', 'example', 'new']
['a', 'b', 'new', 'zoom', 'z', 'example', 'new', 'two', 'elements']

```

Difference between append und extend Lists have two methods, `extend()` and `append()`, that look like they do the same thing, but are in fact completely different. `extend()` takes a single argument, which is always a list, and adds each of the elements of that list to the original list. On the other hand, `append()` takes one argument, which can be any data type, and simply adds it to the end of the list.

```

1 # extend method
2 li = ['a', 'b', 'c']
3
4 # li is extended with a list of another three elements ('d', 'e', and 'f'), so you now have a list of six elements

```

```

5  li.extend(['d', 'e', 'f'])
6  print(li)
7  print(len(li))
8  print(li[-1])
9
10 #append method
11 li = ['a', 'b', 'c']
12
13 # append method is called with a single argument, which is a list of three elements
14 # Now the list contains four elements because the last element appended is itself a list. Lists c
15 li.append(['d', 'e', 'f'])
16 print(li)
17 print(len(li))
18 print(li[-1])

['a', 'b', 'c', 'd', 'e', 'f']
6
f
['a', 'b', 'c', ['d', 'e', 'f']]
4
['d', 'e', 'f']

```

Searching Lists

```

1  li = ['a', 'b', 'new', 'zoom', 'z', 'example', 'new', 'two', 'elements']
2  print(li)
3
4  # index finds (only) the first occurrence of a value in the list and returns the index
5  print(li.index("example"))
6  print(li.index("new"))
7
8  # If the value is not found in the list, Python raises an exception.
9  # To test whether a value is in the list, use in, which returns True if the value is found or Fal
10 print(li.index("c"))

5
2
Traceback (most recent call last):
  File "...", line ?, in ?
ValueError: 'c' is not in list

```

Deleting List Elements

```

1  li = ['a', 'b', 'new', 'zoom', 'z', 'example', 'new', 'two', 'elements']
2  print(li)
3
4  # remove (only) the first occurrence of a value from a list.
5  li.remove("z")
6  print(li)
7
8  # removes only the first occurrence of a value
9  li.remove("new")
10 print(li)
11
12 # pop removes the last element of the list, and it returns the value that it removed
13 li.pop()
14 print(li)
15
16 # If the value is not found in the list, Python raises an exception.
17 li.remove("c")

```

```

['a', 'b', 'new', 'zoom', 'z', 'example', 'new', 'two', 'elements']
['a', 'b', 'new', 'zoom', 'example', 'new', 'two', 'elements']
['a', 'b', 'zoom', 'example', 'new', 'two', 'elements']
['a', 'b', 'zoom', 'example', 'new', 'two']
Traceback (most recent call last):
  File "...", line ?, in ?
ValueError: list.remove(x): x not in list

```

Using List Operators Lists can also be concatenated with the + operator. `list = list + otherlist` has the same result as `list.extend(otherlist)`. But the + operator returns a new (concatenated) list as a value, whereas `extend` only alters an existing list. This means that `extend` is faster, especially for large lists.

```

1 li = ['a', 'b', 'zoom']
2 li = li + ['example', 'new']
3 print(li)
4
5 # li += ['two'] is equivalent to li.extend(['two'])
6 li += ['two']
7 print(li)
8
9 # The * operator works on lists as a repeater: li = [1, 2] * 3 is equivalent to li = [1, 2] + [1,
10 li = [1, 2] * 3
11 print(li)

['a', 'b', 'zoom', 'example', 'new']
['a', 'b', 'zoom', 'example', 'new', 'two']
[1, 2, 1, 2, 1, 2]

```

Tuples

A tuple is an immutable list and can not be changed in any way once it is created. A tuple is defined in the same way as a list, except that the whole set of elements is enclosed in parentheses instead of square brackets. The elements of a tuple have a defined order and the indices are zero-based, just like a list.

Tuples are faster than lists. If you're defining a constant set of values and all you're ever going to do with it is iterate through it, use a tuple instead of a list. Tuples can be converted into lists, and vice-versa. The built-in `tuple` function takes a list and returns a tuple with the same elements, and the `list` function takes a tuple and returns a list.

```

1 t = ("a", "b", "mpilgrim", "z", "example")
2 print(t)
3
4 # The first element of a non-empty tuple is always t[0]
5 print(t[0])
6
7 # Negative indices count from the end of the tuple, just as with a list.
8 print(t[-1])
9
10 # Slicing works too, just like a list.
11 print(t[1:3])

('a', 'b', 'mpilgrim', 'z', 'example')
a
example
('b', 'mpilgrim')

```

Keep in mind that tuples have not methods.

```

1 t = ('a', 'b', 'mpilgrim', 'z', 'example')
2

```

```

3 # all following examples cause errors
4 # t.append("new")
5 # t.remove("z")
6 # t.index("example")
7
8 # You can use "in" to see if an element exists in the tuple.
9 "z" in t

```

True

ï»¿.. include:: ../../include/global.inc

If and else statements

if Statements

Perhaps the most well-known statement type is the *if* statement. For example:

```

1 x = int(input("Please enter an integer: "))
2
3 if x < 0:
4     print('Negative')
5 elif x == 0:
6     print('Zero')
7 else:
8     print('Positive')

```

There can be zero or more *elif* parts, and the *else* part is optional. The keyword ‘*elif*’ is short for ‘else if’, and is useful to avoid excessive indentation. An *if ... elif ... elif ...* sequence is a substitute for the *switch* or *case* statements found in other languages.

ï»¿.. include:: ../../include/global.inc

Loops and Lists

for Statements

The *for* statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers, or giving the user the ability to define both the iteration step and halting condition, Python’s *for* statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```

1 # Measure some strings:
2 words = ['cat', 'window', 'defenestrate']
3
4 for w in words:
5     print(w, len(w))

```

```

cat 3
window 6
defenestrate 12

```

If you need to modify the sequence you are iterating over while inside the loop (for example to duplicate selected items), it is recommended that you first make a copy. Iterating over a sequence does not implicitly make a copy. The slice notation makes this especially convenient:

```

1 for w in words[:]:          # Loop over a slice copy of the entire list.
2     if len(w) > 6:
3         words.insert(0, w)
4 words

```



```
['defenestrate', 'cat', 'window', 'defenestrate']
```

The range Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```
1 for i in range(5):
2     print(i)

0
1
2
3
4
```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```
1 range(5, 10)           # 5 through 9
2 range(0, 10, 3)        # 0, 3, 6, 9
3 range(-10, -100, -30)   # -10, -40, -70
```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```
1 a = ['Mary', 'had', 'a', 'little', 'lamb']
2 for i in range(len(a)):
3     print(i, a[i])

0 Mary
1 had
2 a
3 little
4 lamb
```

In most such cases, however, it is convenient to use the `enumerate()` function.

break and *continue* Statements, and *else* Clauses on Loops

The *break* statement, like in C, breaks out of the smallest enclosing *for* or *while* loop.

Loop statements may have an *else* clause; it is executed when the loop terminates through exhaustion of the list (with *for*) or when the condition becomes false (with *while*), but not when the loop is terminated by a *break* statement. This is exemplified by the following loop, which searches for prime numbers:

```
1 for n in range(2, 10):
2     for x in range(2, n):
3         if n % x == 0:
4             print(n, 'equals', x, '*', n//x)
5             break
6         else:
7             # loop fell through without finding a factor
8             print(n, 'is a prime number')

2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
```

```
8 equals 2 * 4
9 equals 3 * 3
```

When used with a loop, the `else` clause has more in common with the `else` clause of a *try* statement than it does that of *if* statements: a *try* statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the *try* statement and exceptions.

The *continue* statement continues with the next iteration of the loop:

```
1 for num in range(2, 10):
2     if num % 2 == 0:
3         print("Found an even number", num)
4         continue
5     print("Found a number", num)
```

```
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

```
ï»¿.. include:: ../include/global.inc
```

Classes and Objects

Classes

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below *Private Variables*), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)

A Word About Names and Objects Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

Python Scopes and Namespaces Before introducing classes, I first have to tell you something about Python’s scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what’s going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let’s begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that’s normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as `abs()`, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module’s attributes and the global names defined in the module: they share the same namespace!¹

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names
- the next-to-last scope contains the current module’s global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the middle scope containing the module’s global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variable are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module’s namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module’s namespace, no matter from where or by what alias the function is called. On the other hand, the

¹ Except for one thing. Module objects have a secret read-only attribute called `__dict__` which returns the dictionary used to implement the module’s namespace; the name `__dict__` is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers.

actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don’t rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no `global` statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should be rebound there.

Scopes and Namespaces Example

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Note how the *local* assignment (which is default) didn’t change `scope_test`’s binding of `spam`. The `nonlocal` assignment changed `scope_test`’s binding of `spam`, and the `global` assignment changed the module-level binding.

You can also see that there was no previous binding for `spam` before the `global` assignment. Classes introduce a little bit of new syntax, three new object types, and some new semantics.

Class Definition Syntax

The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    .
```

```
.
<statement-N>
```

Class definitions, like function definitions (`def` statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an `if` statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we’ll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the `end`), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we’ll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (`ClassName` in the example).

Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class’s namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
```

```
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Instance Objects

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

data attributes correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value 16, without leaving a trace:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

The other kind of instance attribute reference is a *method*. A method is a function that “belongs to” an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called `append`, `insert`, `remove`, `sort`, and so on. However, in the following discussion, we’ll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a *method object*, not a function object.

Method Objects

Usually, a method is called right after it is bound:

```
x.f()
```

In the `MyClass` example, this will return the string ‘hello world’. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
    print(xf())
```

will continue to print `hello world` until the end of time.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn’t actually used...

Actually, you may have guessed the answer: the special thing about methods is that the object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method’s object before the first argument.

If you still don’t understand how methods work, a look at the implementation can perhaps clarify matters. When an instance attribute is referenced that isn’t a data attribute, its class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the

function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list. Data attributes override method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts. Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore), or using verbs for methods and nouns for data attributes.

Data attributes may be referenced by methods as well as by ordinary users (“clients”) of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care — clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided — again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a *class browser* program might be written that relies upon such a convention.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

Now `f`, `g` and `h` are all attributes of class `C` that refer to function objects, and consequently they are all methods of instances of `C` — `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the `self` argument:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing its definition. (A class is never used as a global scope.) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we’ll find some good reasons why a method would want to reference its own class.

Each value is an object, and therefore has a *class* (also called its *type*). It is stored as `object.__class__`. Of course, a language feature would not be worthy of the name “class” without supporting inheritance. The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

The name `BaseClassName` must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There’s nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively *virtual*.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`. This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as `BaseClassName` in the global scope.)

Python has two built-in functions that work with inheritance:

- Use `isinstance()` to check an instance’s type: `isinstance(obj, int)` will be `True` only if `obj.__class__` is `int` or some class derived from `int`.
- Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is `True` since `bool` is a subclass of `int`. However, `issubclass(float, int)` is `False` since `float` is not a subclass of `int`.

Multiple Inheritance

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in `DerivedClassName`, it is searched for in `Base1`, then (recursively) in the base classes of `Base1`, and if it was not found there, it was searched for in `Base2`, and so on.

In fact, it is slightly more complex than that; the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the `super` call found in single-inheritance languages.

Dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see <http://www.python.org/download/releases/2.3/mro/>. “Private” instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. `_spam`) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called *name mangling*. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `__classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update    # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

Notice that code passed to `exec()` or `eval()` does not consider the `classname` of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing `__dict__` directly. Sometimes it is useful to have a data type similar to the Pascal “record” or C “struct”, bundling together a few named data items. An empty class definition will do nicely:

```
class Employee:
    pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
```

```
john.salary = 1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

Instance method objects have attributes, too: `m.__self__` is the instance object with the method `m()`, and `m.__func__` is the function object corresponding to the method. User-defined exceptions are identified by classes as well. Using this mechanism it is possible to create extensible hierarchies of exceptions.

There are two new valid (semantic) forms for the `raise` statement:

```
raise Class
```

```
raise Instance
```

In the first form, `Class` must be an instance of `type` or of a class derived from it. The first form is a shorthand for:

```
raise Class()
```

A class in an `except` clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an `except` clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```
class B(Exception):
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Note that if the `except` clauses were reversed (with `except B` first), it would have printed B, B, B — the first matching `except` clause is triggered.

When an error message is printed for an unhandled exception, the exception's class name is printed, then a colon and a space, and finally the instance converted to a string using the built-in function `str()`. By now you have probably noticed that most container objects can be looped over using a `for` statement:

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line)
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `__next__()` which accesses elements in the container one at a time. When there are no

more elements, `__next__()` raises a `StopIteration` exception which tells the `for` loop to terminate. You can call the `__next__()` method using the `next()` built-in function; this example shows how it all works:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    next(it)
StopIteration
```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__()` method which returns an object with a `__next__()` method. If the class defines `__next__()`, then `__iter__()` can just return `self`:

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the `yield` statement whenever they want to return data. Each time `next()` is called on it, the generator resumes where it left-off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Anything that can be done with generators can also be done with class based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `__next__()` methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`. In combination, these features make it easy to create iterators with no more effort than writing a regular function. Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

Examples:

```
>>> sum(i*i for i in range(10))           # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))   # dot product
260

>>> from math import pi, sin
>>> sine_table = {x: sin(x*pi/180) for x in range(0, 91)}

>>> unique_words = set(word for line in page for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

```
ï»¿.. include:: ../../include/global.inc
```

Functions

Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*. (More about docstrings can be found in the section [Documentation Strings](#).) There

are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a *global* statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object). [#] When a function calls another function, a new local symbol table is created for that call.

A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function. This serves as a general renaming mechanism:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a *return* statement do return a value, albeit a rather boring one. This value is called `None` (it's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using `print()`:

```
>>> fib(0)
>>> print(fib(0))
None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

- The *return* statement returns with a value from a function. *return* without an expression argument returns `None`. Falling off the end of a function also returns `None`.
- The statement `result.append(a)` calls a *method* of the list object `result`. A method is a function that 'belongs' to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using *classes*, see [Classes](#)) The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `result = result + [a]`, but more efficient.

More on Defining Functions It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
        print(complaint)
```

This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

This example also introduces the `in` keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the *defining* scope, so that

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

will print 5.

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

This will print

```
[1]
[1, 2]
[1, 2, 3]
```

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

Keyword Arguments

Functions can also be called using *keyword arguments* of the form `kwarg=value`. For instance, the following function:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

```
parrot(1000)                                # 1 positional argument
parrot(voltage=1000)                        # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')  # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)  # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

but all the following calls would be invalid:

```
parrot()                                # required argument missing
parrot(voltage=5.0, 'dead')             # non-keyword argument after a keyword argument
parrot(110, voltage=220)                 # duplicate value for the same argument
parrot(actor='John Cleese')              # unknown keyword argument
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. `actor` is not a valid argument for the `parrot` function), and their order is not important. This also includes non-optional arguments (e.g. `parrot(voltage=1000)` is valid too). No argument may receive a value more than once. Here's an example that fails due to this restriction:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see *Mapping Types — dict*) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`.) For example, if we define a function like this:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    keys = sorted(keywords.keys())
    for kw in keys:
        print(kw, ":", keywords[kw])
```

It could be called like this:

```
cheeseshop("Limburger", "It's very runny, sir.",
            "It's really very, VERY runny, sir.",
            shopkeeper="Michael Palin",
```

```
client="John Cleese",
sketch="Cheese Shop Sketch")
```

and of course it would print:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Note that the list of keyword argument names is created by sorting the result of the keywords dictionary's `keys()` method before printing its contents; if this is not done, the order in which the arguments are printed is undefined.

Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see *Tuples and Sequences*). Before the variable number of arguments, zero or more normal arguments may occur.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Normally, these variadic arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function. Any formal parameters which occur after the `*args` parameter are 'keyword-only' arguments, meaning that they can only be used as keywords rather than positional arguments.

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in `range()` function expects separate *start* and *stop* arguments. If they are not available separately, write the function call with the `*`-operator to unpack the arguments out of a list or tuple:

```
>>> list(range(3, 6))           # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))         # call with arguments unpacked from a list
[3, 4, 5]
```

In the same fashion, dictionaries can deliver keyword arguments with the `**`-operator:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
```



```
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

Lambda Forms

By popular demand, a few features commonly found in functional programming languages like Lisp have been added to Python. With the `lambda` keyword, small anonymous functions can be created. Here's a function that returns the sum of its two arguments: `lambda a, b: a+b`. Lambda forms can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda forms can reference variables from the containing scope:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

Documentation Strings

Here are some conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line *after* the first line of the string determines the amount of indentation for the entire documentation string. (We can't use the first line since it is generally adjacent to the string's opening quotes so its indentation is not apparent in the string literal.) Whitespace "equivalent" to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Here is an example of a multi-line docstring:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
>>> print(my_function.__doc__)
Do nothing, but document it.

    No, really, it doesn't do anything.
```

```
ï»¿.. include:: ../../include/global.inc
```

Modules

Import Modules

Reload Modules in ITOM

path Variablen

itom-packages Folder

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibonacci.py` in the current directory with the following contents:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibonacci
```

This does not enter the names of the functions defined in `fibonacci` directly in the current symbol table; it only enters the module name `fibonacci` there. Using the module name you can access the functions:

```
>>> fibonacci.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibonacci.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibonacci.__name__
'fibonacci'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

More on Modules A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module is imported somewhere.²

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, `modname.itemname`.

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.

There is a variant of the `import` statement that imports names from a module directly into the importing module's symbol table. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

Note: For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it's just one module you want to test interactively, use `imp.reload()`, e.g. `import imp; imp.reload(modulename)`.

Executing modules as scripts

When you run a Python module with

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

² In fact function definitions are also ‘statements’ that are ‘executed’; the execution of a module-level function enters the function name in the module's global symbol table.

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

The Module Search Path

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- the directory containing the input script (or the current directory).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- the installation-dependent default.

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See section *Standard Modules* for more information.

“Compiled” Python files

As an important speed-up of the start-up time for short programs that use a lot of standard modules, if a file called `spam.pyc` exists in the directory where `spam.py` is found, this is assumed to contain an already-“byte-compiled” version of the module `spam`. The modification time of the version of `spam.py` used to create `spam.pyc` is recorded in `spam.pyc`, and the `.pyc` file is ignored if these don’t match.

Normally, you don’t need to do anything to create the `spam.pyc` file. Whenever `spam.py` is successfully compiled, an attempt is made to write the compiled version to `spam.pyc`. It is not an error if this attempt fails; if for any reason the file is not written completely, the resulting `spam.pyc` file will be recognized as invalid and thus ignored later. The contents of the `spam.pyc` file are platform independent, so a Python module directory can be shared by machines of different architectures.

Some tips for experts:

- When the Python interpreter is invoked with the `-O` flag, optimized code is generated and stored in `.pyo` files. The optimizer currently doesn’t help much; it only removes `assert` statements. When `-O` is used, *all bytecode* is optimized; `.pyc` files are ignored and `.py` files are compiled to optimized bytecode.
- Passing two `-O` flags to the Python interpreter (`-OO`) will cause the bytecode compiler to perform optimizations that could in some rare cases result in malfunctioning programs. Currently only `__doc__` strings are removed from the bytecode, resulting in more compact `.pyo` files. Since some programs may rely on having these available, you should only use this option if you know what you’re doing.
- A program doesn’t run any faster when it is read from a `.pyc` or `.pyo` file than when it is read from a `.py` file; the only thing that’s faster about `.pyc` or `.pyo` files is the speed with which they are loaded.
- When a script is run by giving its name on the command line, the bytecode for the script is never written to a `.pyc` or `.pyo` file. Thus, the startup time of a script may be reduced by moving most of its code to a module and having a small bootstrap script that imports that module. It is also possible to name a `.pyc` or `.pyo` file directly on the command line.
- It is possible to have a file called `spam.pyc` (or `spam.pyo` when `-O` is used) without a file `spam.py` for the same module. This can be used to distribute a library of Python code in a form that is moderately hard to reverse engineer.

- The module `compileall` can create `.pyc` files (or `.pyo` files when `-O` is used) for all modules in a directory.

Python comes with a library of standard modules, described in a separate document, the Python Library Reference (“Library Reference” hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the `winreg` module is only provided on Windows systems. One particular module deserves some attention: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determines the interpreter’s search path for modules. It is initialized to a default path taken from the environment variable `PYTHONPATH`, or from a built-in default if `PYTHONPATH` is not set. You can modify it using standard list operations:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
 '__stdin__', '__stdout__', '__getframe__', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'callstats', 'copyright',
 'displayhook', 'exc_info', 'excepthook',
 'exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getdlopenflags',
 'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',
 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
 'version', 'version_info', 'warnoptions']
```

Without arguments, `dir()` lists the names you have defined currently:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `builtins`:

```
>>> import builtins
>>> dir(builtins)

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'Buffer
Error', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'Environme
```

```
ntError', 'Exception', 'False', 'FloatingPointError', 'FutureWarning', 'Generato
rExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexErr
or', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError',
'None', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'P
endingDeprecationWarning', 'ReferenceError', 'RuntimeError', 'RuntimeWarning', '
StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'Ta
bError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'Unicod
eEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserW
arning', 'ValueError', 'Warning', 'ZeroDivisionError', '__build_class__', '__deb
ug__', '__doc__', '__import__', '__name__', '__package__', 'abs', 'all', 'any',
'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'chr', 'classmethod', 'compile', '
complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate
', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr',
'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance',
'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memory
view', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property'
, 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sort
ed', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

Packages are a way of structuring Python’s module namespace by using “dotted module names”. For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other’s global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other’s module names.

Suppose you want to design a collection of modules (a “package”) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here’s a possible structure for your package (expressed in terms of a hierarchical filesystem):

```
sound/                                Top-level package
  __init__.py                         Initialize the sound package
  formats/                           Subpackage for file format conversions
    __init__.py
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
  effects/                           Subpackage for sound effects
    __init__.py
    echo.py
    surround.py
    reverse.py
    ...
  filters/                           Subpackage for filters
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...
```

When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also

execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

An alternative way of importing the submodule is:

```
from sound.effects import echo
```

This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Yet another variation is to import the desired function or variable directly:

```
from sound.effects.echo import echofilter
```

Again, this loads the submodule `echo`, but this makes its function `echofilter()` directly available:

```
echofilter(input, output, delay=0.7, atten=4)
```

Note that when using `from package import item`, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an `ImportError` exception is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

Importing * From a Package

Now what happens when the user writes `from sound.effects import *`? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The `import` statement uses the following convention: if a package's `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when `from package import *` is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing `*` from their package. For example, the file `sounds/effects/__init__.py` could contain the following code:

```
__all__ = ["echo", "surround", "reverse"]
```

This would mean that `from sound.effects import *` would import the three named submodules of the `sound` package.

If `__all__` is not defined, the statement `from sound.effects import *` does *not* import all sub-modules from the package `sound.effects` into the current namespace; it only ensures that the package `sound.effects` has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by `__init__.py`. It also includes any submodules of the package that were explicitly loaded by previous `import` statements. Consider this code:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In this example, the `echo` and `surround` modules are imported in the current namespace because they are defined in the `sound.effects` package when the `from . . import` statement is executed. (This also works when `__all__` is defined.)

Although certain modules are designed to export only names that follow certain patterns when you use `import *`, it is still considered bad practise in production code.

Remember, there is nothing wrong with using `from Package import specific_submodule!` In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

Intra-package References

When packages are structured into subpackages (as with the `sound` package in the example), you can use absolute imports to refer to submodules of siblings packages. For example, if the module `sound.filters.vocoder` needs to use the `echo` module in the `sound.effects` package, it can use `from sound.effects import echo`.

You can also write relative imports, with the `from module import name` form of import statement. These imports use leading dots to indicate the current and parent packages involved in the relative import. From the `surround` module for example, you might use:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Note that relative imports are based on the name of the current module. Since the name of the main module is always `"__main__"`, modules intended for use as the main module of a Python application must always use absolute imports.

Packages in Multiple Directories

Packages support one more special attribute, `__path__`. This is initialized to be a list containing the name of the directory holding the package's `__init__.py` before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

While this feature is not often needed, it can be used to extend the set of modules found in a package.

```
ï»¿>.. include:: ../../include/global.inc
```

Prompting and passing

```
ï»¿>.. include:: ../../include/global.inc
```

Reading and writing files

Pickle -> load / dump

The following resources have been used:

- [Python-Kurs \(v3\) \(German\)](#)
- [Python Course \(v3\) \(English\)](#)
- [Official Python v3.3.0 documentation](#)
- [Moving from Python 2 to Python 3 \(Cheatsheet\)](#)

- Dive into Python (v2)
- Learn Python The Hard Way (be careful: Python 2)
- Python Tutorial on javas2s.com (v2)
- Python examples (example source code) on javas2s.com (v2)
- Python Tutorial on tutorialspoint.com (v2)

If you use **Python** in **itom** you should consider some tiny limitations, stated in this document:

8.1.2 Python Limitations in itom

You can use almost all methods, classes or modules provided by the core of **Python** or any external modules, if you consider the following hints or rules:

- The embedded python interpreter in **itom** is based on **Python 3.2**.
- **Python** is executed in its own thread in **itom**, therefore you should not use any modules that create some form of GUI. Usually the creation of any GUI element needs to be executed in the main thread of the calling application. Therefore don't use the python package **PyQt**. Instead use the GUI-functionality which is directly provided by **itom** and allows to extend the GUI of **itom**.
- If you were used to use to python methods, that asked the user for some input in the original python command window, you cannot use these commands in **itom**. Instead use message or input boxes that are provided by the class **ui** of the module **itom** in order to get some basic information from the user.
- It can not be guaranteed that the use of the **threading** module of **Python** is fully usable together with **itom**.

8.2 Python module itom

The main purpose of the embedded **Python** interpreter in **itom** is to access the specific functionalities provided by **itom**. This is done by the **Python**-module `itom`, that is importable only in the embedded **Python** interpreter in **itom**. This module includes interfaces for hardware and algorithm plugins of **itom** as well as classes that wrap the most important internal data structures of **itom**, like matrices (class `dataObject`), point clouds (class `pointCloud`) or polygon meshes (class `polygonMesh`). Additionally the module provides functions to manipulate or extend the graphical user interface of **itom** as well as to create own dialogs or windows (provided by the class `ui` and `uiItem`).

More information about the module `itom` can be found under:

8.2.1 Python-Module itom

The module `itom` is a python module, that builds the connecting part between the embedded python script language and the overall **itom** application. This module is only available in the context of a running **itom** software.

Import itom

Like any other python module, you need to import the module `itom` in your script. Usually, the import is done at the beginning of a script. In the following example, it is shown how the method **filterHelp** of the module `itom` can be called with respect to different ways of import `itom`:

```

1  #1. import the whole module. Any method is then directly accessible
2  from itom import *
3  filterHelp("...")
4
5  #2. import the itom module without the global access to its methods
6  import itom
7  itom.filterHelp("...")

```

```
8
9 #3. only import certain methods or classes of the module itom
10 from itom import filterHelp as anyAlias
11 anyAlias("...")
```

If you simply type any commands in the **itom** command line or if you directly execute any script at the top level, you don't need import the module `itom`, since this module already has been imported at startup of **itom** with

```
from itom import *
```

However, if you import another script file in your main script file and you want to access any methods of `itom` in this secondary script, then you need to import `itom` in this script using one of the methods shown above.

Content of `itom`

- class **dataObject**. This is the **itom** internal matrix class, compatible to *Numpy*, and also used by any connected grabber or camera. Every matrix is extended by fixed and user-defined tags and keywords. For an introduction to the data object, see *DataObject*, a full reference is available under `dataObject`. The data object is compatible to any numpy array, however the tags and keywords will get lost.
- class **npDataObject**. This class is inherited from **numpy.array** and extends it by the tags and keywords, also available in the data object. For more information see *itomNpDataObject* or the script reference `npDataObject`.
- classes **ui** and **uiItem** are the main classes for creating user defined dialogs and windows in **itom** and show them using some lines of script code. For more information about their use, see *Creating advanced dialogs and windows* or their definitions in the script reference `ui` and `uiItem`.
- class **dataIO** is the class in order to access any plugin instance of type **dataIO** (cameras, grabbers, AD-converter...). The full reference can be found under `dataIO`.
- class **actuator** is the class in order to access any plugin instance of type **actuator**, like motor stages... The full reference can be found under `actuator`.
- **other class free methods**. `itom` also directly contains a lot of methods, that makes features of `itom` accessible by a Python script:
 - add or remove buttons or items to the **itom** menu and toolbar (see *Customize the menu and toolbars of itom*)
 - get help about plugins and their functionality
 - call any algorithm or filter, provided by a plugin of type **algo**
 - directly plot matrices like `dataObjects`.

Contents:

DataObject

Introduction In **itom**, the class `dataObject` is the main array object. Arrays in **itom** can have the following properties:

- unlimited number of dimensions
- each dimension can have an arbitrary size
- **possible data types:**

```
"uint8"      #unsigned integer, 8 bit [0,255]
"int8"       #signed integer, 8 bit [-128,127]
"uint16"     #unsigned integer, 16 bit [0,65536]
"int16"      #signed integer, 16 bit [-32768,32767]
"uint32"     #unsigned integer, 32 bit
```

```

"int32"          #signed integer, 32 bit
"float32"        #floating point, 32 bit single precision
"float64"        #floating point, 64 bit double precision
"complex64"      #complex number with two float32 components
"complex128"     #complex number with two float64 components

```

Before giving a short tutorial about how to use the class `dataObject`, the base idea and concept of the array structure should be explained. If you already know the huge **Python** module **Numpy** with its base array class **numpy.array**, one will ask why another similar array class is provided by **itom**. The reasons for this are as follows:

- The python class `dataObject` is just a wrapper for the **itom** internal class **DataObject**, written in C++. This array structure is used all over **itom** and also passed to any plugin instances of **itom**. Internally, the C++ class **DataObject** is based on OpenCV-matrices, such that functionalities provided by the open-source Computer-Vision Library (OpenCV) can be used by **itom**.
- The class **dataObject** should also be used to store real measurement data. Therefore it is possible to add tags and other meta information to every `dataObject` (like axes descriptions, scale and offset values, protocol entries...).
- Usually, array classes (like the class **Numpy.array**) store the whole matrix in one non-interrupting block in memory. Due to the working principle of every operating system, it is sometimes difficult to allocate a huge block in memory. Therefore, **dataObject** only stores the sub-matrices of the last two-dimensions in single blocks in memory, while the first **n-2** dimensions of the array are represented by one vector in memory, where every cell is pointing to the corresponding sub-matrix (called plane). Using this concept, huge arrays can be allocated without causing a memory error.

Creating a dataObject In general, a `dataObject` is created like any other class instance in **Python**, hence the constructor of class `dataObject` is called. For a full reference of the constructor of class **dataObject**, type

```
help(dataObject)
```

In the following example, some `dataObjects` of different size and types are created. Using these constructors, the content of the created array is arbitrary at initialization:

```

1  #1. empty dataObject, dimensions: 0, size: []
2  a = dataObject()
3
4  #2. one dimensional dataObject
5  # a one dimensional dataObject already is
6  # allocated as an array of size [1 x n]
7  b = dataObject([5], "float32") #size [1x5]
8
9  #3. 5 x 3 array, type: int8
10 c = dataObject([5,3], "int8")
11
12 #4. 2 x 5 x 10 array, type: complex128
13 # here two planes of size [5x10] are created and a vector with two items points to them
14 d = dataObject([2,5,10], "complex128")
15
16 #5. 2 x 5 x 10 array, type: complex128, continuous
17 # This matrix has the same size and type than matrix
18 # 'd' above. However, the continuous keyword indicates,
19 # that python should already allocate all planes in
20 # one block. Then the data object can be converted in
21 # a numpy.array without the need of copying the data block
22 # in memory. It is useful to use this keyword, if you
23 # often want to switch between dataObject and numpy.arrays.
24 # However consider that this is not recommended for huge
25 # matrices.
26 e = dataObject([2,5,10], "complex128", continuous = True)

```

You can also use the copy constructor of class **dataObject** in order to create a **dataObject** from another array-like object or a sequence of numbers (tuple, list...). In **Python** it is usual, that different objects share their memory (for arrays the memory is mainly the data block(s)) as long as possible, such that memory and execution time is saved. This is also the case when using the copy constructor. See the **Numpy** documentation for more information about this. The main thing you should know is, that if you change the value of any cell of an array, the corresponding value is also changed in all arrays, that share their memory with the **dataObject**.

```
1  #1. create dataObject from any array-like object (e.g. Numpy array)
2  import numpy as np
3  a = np.ndarray([5,7])
4  b = dataObject(a) #b has the continuous flag set
5
6  #2. create dataObject from a tuple of values
7  # any object, that python can interpret as sequence can be used
8  # in order to initialize the data object. The dataObject can have
9  # an arbitrary size or number of dimensions, if the total number
10 # of elements fits to the length of the given input sequence.
11 # In this case, the sequence is totally copied into the data object.
12 # The values are filled row-by-row into the array, also called as
13 # c-continuous creation.
14 c = (2,7,4,3,8,9,6,2) #8 values
15 d = dataObject([2,4], data = c)
16
17 #3. create a dataObject as shallow copy of another dataObject
18 e = dataObject(d)
```

Some text about the basic idea of the **dataObject** and how it works. Some additional pictures.

For a detailed methods-summary of the *dataObject* see *itom Script Reference*.

npDataObject

Some text about the basic idea of the **npDataObject** and how it works. Some additional pictures.

For a detailed methods-summary of the **npDataObject** see *itom Script Reference*.

Plugin-Interface and Plugins

The basic plugin idea again, some more details,

a description of type, initialisation, ... how to work with, how to delete them, how to get help, ...

For a detailed methods-summary of **dataIO**, **actuator**, **filter** and **widgets** see *itom Script Reference*.

8.2.2 Glossary

(non-)continuous data object

per default, matrixes in the data object are stored as follows: the last two dimensions are stored continuously in a two dimensional matrix by the help of openCV-Mat-structures. All these 2dim-matrixes are then stored in a one-dimensional vector, however the relation between the position in this vector and the first ($n-2$) dimensions, where n is the number of total dimensions, is given by a simple equation, which can for instance be found in the *openCV* documentation. Since the data blocks for each two-dimensional matrix must not be stored continuously in memory, this method is called *non-continuous data object*.

In order to realize a compatible version with respect to *numpy*, *matlab*... the data can also be stored *continuously*. The basic structure for the data object is the same than in the *non-continuous* (default) version, but the data of each 2dim-matrix lies continuously in memory and each data-pointer of each matrix just points to the first element of the corresponding matrix in this big data block in memory.

Data objects of class *ndDataObject* are always organized as continuous data object, while ordinary data objects (class *dataObject*) can be continuous or non-continuous.

The non-continuous representation has advantages especially with respect to huge data sets, since it is more difficult to get a big continuous block in memory without reorganizing it than multiple smaller blocks of memory, which can be distributed randomly in memory.

Matrixes with only one or two dimension are automatically stored continuously.

Deep Copies

Are copies

Shallow Copies

are copied references

Python Version

The full script reference of the module `itom` can be found under :ref:`itom-Script-Reference`.

8.3 Further python packages

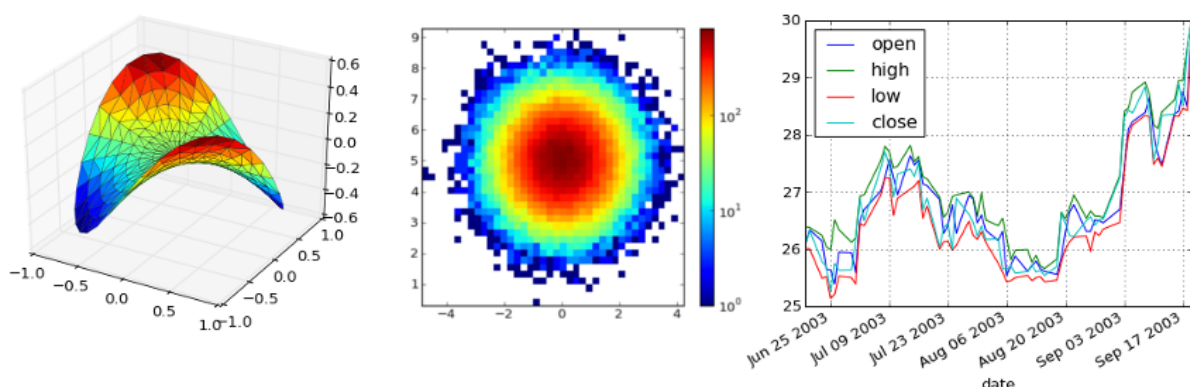
Beside the module `itom`, it is recommended to use the functionalities provided by the **Python** packages **Numpy**, **Scipy** and **Matplotlib**. During the development of **itom** a high compatibility especially to those modules has been taken into account. For instance it is possible to render the **Matplotlib** figures in user defined windows, created by the class `ui` of the module `itom` (see *qtcreator*). Additionally, the **Numpy** array is compatible to the **itom** internal `dataObject` or `npDataObject`.

8.3.1 Python-Module numpy

8.3.2 Python-Module scipy

8.3.3 Python-Module matplotlib

The module **matplotlib** can be used in order to create two or three dimensional plots like in these examples:



Since the internal plotting methods of **itom** mainly provide display widgets for plotting matrix contents and images, **matplotlib** can be used for plotting other types of graphics like graphs, line plots, bars... A huge list of examples can be found under <http://matplotlib.org/gallery.html>.

In order to render the output of **matplotlib** into an **itom** window or a user interface generated by **itom** (see *qtcreator*), you need to write the following command before importing any module of the package **matplotlib**:

```
import matplotlib
matplotlib.use('module://mpl_itom.backend_itomagg', False)
```

Simple Matplotlib example

This example shows you that is possible to use any arbitrary matplotlib python script and execute it in **itom**. Therefore, the example `hist2d_log_demo.py` from the pylab examples on <http://matplotlib.org> is taken.

Its source code is:

```
import matplotlib
matplotlib.use('module://mpl_itom.backend_itomagg', False)
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.collections import EllipseCollection

x = np.arange(10)
y = np.arange(15)
X, Y = np.meshgrid(x, y)

XY = np.hstack((X.ravel()[:, np.newaxis], Y.ravel()[:, np.newaxis]))

ww = X/10.0
hh = Y/15.0
aa = X*9

ax = plt.subplot(1,1,1)

ec = EllipseCollection(
    ww,
    hh,
    aa,
    units='x',
    offsets=XY,
    transOffset=ax.transData)

ec.set_array((X+Y).ravel())
ax.add_collection(ec)
ax.autoscale_view()
ax.set_xlabel('X')
ax.set_ylabel('Y')
cbar = plt.colorbar(ec)
cbar.set_label('X+Y')
title("ellipse collection")
plt.show()
```

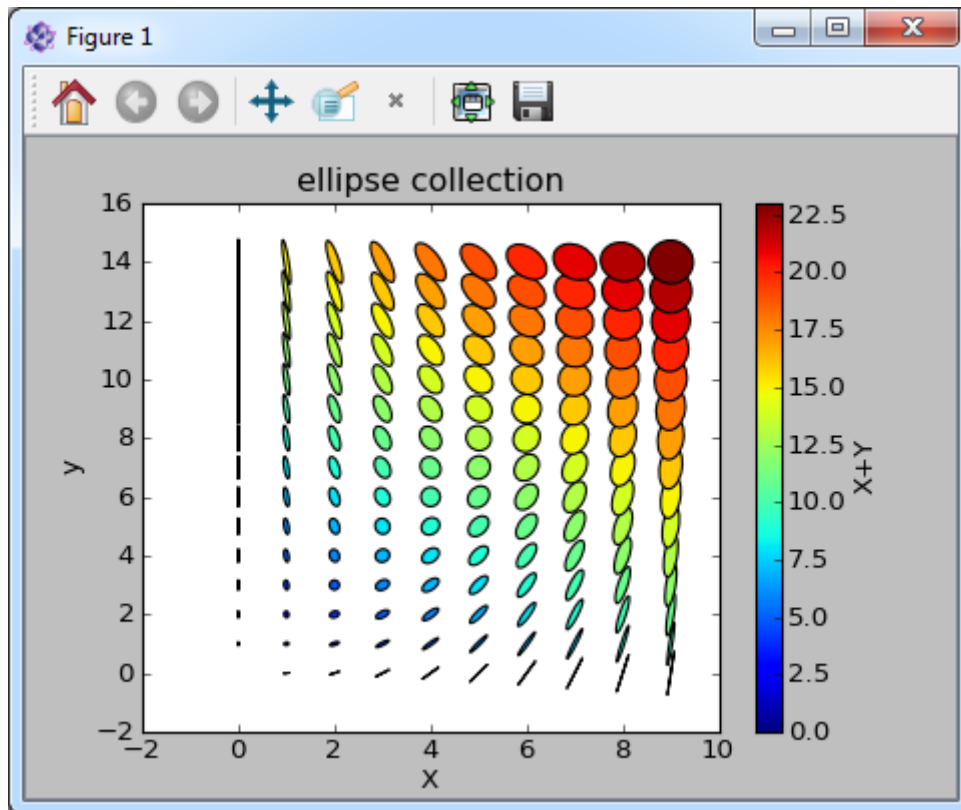
Please consider that the original source code has been changed such that the first two lines are prepended. After executing this script, the following figure is displayed in **itom**:

Note: If the figure does not appear, the matplotlib designer widget for **itom** is not available. This means, the library **matplotlibFigure** in the **designer** folder of **itom** is missing.

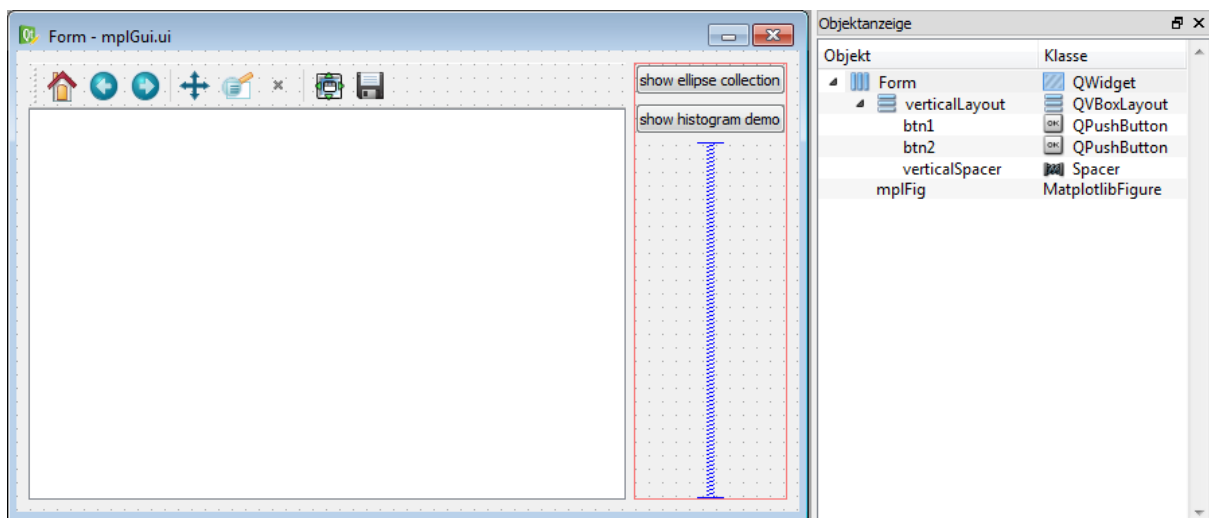
Embedding a matplotlib figure in your own user interface

itom not only provides stand-alone windows for showing the result of the matplotlib, but it is also possible to integrate a matplotlib canvas into own user interfaces created by the QtDesigner and scripted with **Python**. For more information how to do this, see *Creating advanced dialogs and windows*.

If you open the QtDesigner, you will see a widget **MatplotlibFigure** in the section **ITOM Plugins**. Then you can drag&drop one instance of this widget onto your user interface. In our case we opened a new main window and



placed on the left side a MatplotlibFigure (name: mplFig) and on the right side, there are two buttons (name: btn1 and btn2):



When any of the both buttons are pressed, the corresponding example should be displayed in the figure **mplFig** on the left side.

Rest TODO

Other recommended packages are:

- **scikit-image** for image processing
- **PIL** is the python image library
- **sphinx** for creating this documentation

ITOM SCRIPT REFERENCE

This reference gives a full reference for all script commands that are contained in the python module `itom`.

Content:

9.1 itom methods

9.1.1 Plotting and camera

`itom.liveImage (cam[, className])` → shows a camera live image in a newly created figure
Creates a plot-image (2D) and automatically grabs images into this window. This function is not blocking.

Parameters `cam` : {dataIO-Instance}

Camera grabber device from which images are acquired.

`className` : {str}, optional

class name of desired plot (if not indicated default plot will be used (see application settings))

`itom.plot (data[, className])` → plots a dataObject in a newly created figure
Plot an existing dataObject in not dockable, not blocking window. The style of the plot will depend on the object dimensions. If x-dim or y-dim are equal to 1, plot will be a lineplot else a 2D-plot.

Parameters `data` : {DataObject}

Is the data object whose region of interest will be plotted.

`className` : {str}, optional

class name of desired plot (if not indicated default plot will be used (see application settings))

9.1.2 Using algorithms and filters

`itom.filter (name[, furtherParameters, ...])` → invoke a filter (or algorithm) function within an algorithm-plugin.

Parameters `name` : {str}

The name of the filter

`furtherParameters` : {variant}

Further parameters depend on the filter-methods itself (give the mandatory and then optional parameters in their defined order).

Returns `output parameters` : {variant}

The returned values depend on the definition of each filter. In general it is a tuple of all output parameters that are defined by the filter function.

Notes

This function is used to invoke itom filter-functions or algorithms, declared within itom-algorithm plugins. The parameters (arguments) depends on the specific filter function (see filterHelp(name)), By filterHelp() a list of available filter functions is retrieved.

9.1.3 Methods for getting help and information about filters, plugins, ui-elements, ...

`itom.pluginHelp(pluginName[, dictionary = False])` → generates an online help for the specified plugin.

Gets (also print to console) the initialisation parameters of the plugin specified pluginName (str, as specified in the plugin window). If dictionary is True, a dict with all plugin parameters is returned.

Parameters `pluginName` : {str}

is the fullname of a plugin as specified in the plugin window.

dictionary : {bool}, optional

if dictionary == True, function returns an dict with plugin parameters (default: False)

Returns Returns None or a dict depending on the value of parameter dictionary. :

`itom.widgetHelp([widgetName, dictionary = 0, furtherInfos = 0])` → generates an online help for the given widget(s).

Generates an online help for the given widget or returns a list of available widgets.

Parameters `widgetName` : {str}, optional

is the fullname or a part of any widget-name which should be displayed. If widgetName is none or no widget matches widgetName casesensitiv a list with all suitable widgets is given.

dictionary : {dict}, optional

if dictionary == 1, function returns an Py_Dictionary with parameters Default value is 0.

furtherInfos : {int}, optional

defines if a complete parameter-list of name-related widgets to the given widgetName should be displayed (1)

Returns Returns none or a PyDictionary depending on the value of dictionary. :

`itom.filterHelp([filterName, dictionary = 0, furtherInfos = 0])` → generates an online help for the given filter(s).

Generates an online help for the given widget or returns a list of available filter.

Parameters `filterName` : {str}, optional

is the fullname or a part of any filter-name which should be displayed. If filterName is none or no filter matches filterName casesensitiv a list with all suitable filters is given.

dictionary : {dict}, optional

if dictionary == 1, function returns an Py_Dictionary with parameters Default value is 0.

furtherInfos : {int}, optional

defines if a complete parameter-list of name-related filters to the given filterName should be displayed (1)

Returns Returns none or a PyDictionary depending on the value of dictionary. :

`itom.pluginLoaded(pluginname)` → checks if a certain plugin was loaded.

Checks if a specified plugin is loaded and returns the result as a boolean expression.

Parameters `pluginname` : {str}

The name of a specified plugin as usually displayed in the plugin window.

Returns **True, if the plugin has been loaded and can be used, else False. :**

`itom.version([toggle-output[, include-plugins]])` → retrieve complete information about itom version numbers

Parameters **toggle-output** : {bool}, optional

default = false if true, output will be written to a dictionary else to console.

dictionary : {bool}, optional

default = false if true, add informations about plugIn versions.

Returns **None (display outPut) or PyDictionary with version information. :**

Notes

Retrieve complete version information of ITOM and if specified version information of loaded plugins and print it either to the console or to a PyDictionary.

9.1.4 Adding elements to the GUI

`itom.addButton(toolbarName, buttonName, code[, icon, argtuple])` → adds a button to a toolbar in the main window

Parameters **toolbarName** : {str}

The name of the toolbar.

buttonName : {str}

The name (str, identifier) of the button to create.

code : {str, Method, Function}

The code to be executed if button is pressed.

icon : {str}, optional

The filename of an icon-file. This can also be relative to the application directory of 'itom'.

argtuple : {tuple}, optional

Arguments, which will be passed to method (in order to avoid cyclic references try to only use basic element types).

Notes

This function adds a button to a toolbar in the main window. If the button is pressed the simple python command specified by python-code is executed. If the toolbar specified by toolbarName do not exist, the toolbar is created. The button representation will be the optional icon or if icon is not loadable 'buttonName' will be displayed.

ITOM comes with basic icons addressable by './.../iconname.png', e.g. './gui/icons/close.png'. Find available via iconbrower under 'Editor-Menu/Edit/iconbrower' or ctrl-b

`itom.removeButton(toolbarName, buttonName)` → removes a button from a given toolbar.

Parameters **toolbarName** : {str}

The name of the toolbar.

buttonName : {str}

The name (str, identifier) of the button to remove.

Notes

This function removes a button from a toolbar in the main window. If the toolbar is empty after removal, it is deleted.

`itom.addMenu (type, key[, name, code, icon, argtuple])` → adds an element to the menu bar.

Parameters **type** : {Int}

The type of the menu-element (BUTTON:0 [default], SEPARATOR:1, MENU:2). Use the corresponding constants in module 'itom'.

key : {str}

A slash-separated string where every sub-element is the key-name for the menu-element in the specific level.

name : {str}, optional

The text of the menu-element. If not indicated, the last sub-element of key is taken.

code : {str, Method, Function}, optional

The code to be executed if menu element is pressed.

icon : {str}, optional

The filename of an icon-file. This can also be relative to the application directory of 'itom'.

argtuple : {tuple}, optional

Arguments, which will be passed to method (in order to avoid cyclic references try to only use basic element types).

Notes

This function adds an element to the main window menu bar. The root element of every menu-list must be a MENU-element. Such a MENU-element can contain sub-elements. The following sub-elements can be either another MENU, a SEPARATOR or a BUTTON. Only the BUTTON itself triggers a signal, which then executes the code, given by a string or a reference to a callable python method or function. Remember, that this reference is only stored as a weak pointer. If you want to directly add a sub-element, you can give a slash-separated string in the key-parameter. Every sub-component of this string then represents the menu-element in its specific level. Only the element in the last can be something else than MENU.

ITOM comes with basic icons addressable by './.../iconname.png', e.g. './gui/icons/close.png'. Find available via iconbrowser under 'Editor-Menu/Edit/iconbrowser' or ctrl-b

`itom.removeMenu (key)` → remove a menu element with the given key.

Parameters **key** : {str}

The name (str, identifier) of the menu entry to remove.

Notes

This function remove a menu element with the given key. key is a slash separated list. The sub-components then lead the way to the final element, which should be removed.

For more information about using these methods, see *Customize the menu and toolbars of itom*.

9.1.5 Disk-IO

`itom.loadDataObject (filename, dataObject[, doNotAppendIDO])` → load a dataObject from the harddrive to an existing dataObject.

Parameters `filename` : {str}

Filename and Path of the destination (.ido will be added if not available)

dataObject : {DataObject}

A pre-allocated dataObject (empty Objects are allowed).

doNotAppendIDO : {bool}, optional

Optional tag to avoid adding -ido-Tag, default is False.

Notes

This function reads an itom::dataObject from the file specified by filename. MetaData saveType (string, binary) are extracted from the file and restored within the object. The value of string-Tags must be encoded to avoid XML-conflicts. Tagnames which contains special characters leads to XML-conflicts.

`itom.saveDataObject(filename, dataObject[, tagsAsBinary])` → save a dataObject to harddrive.

Parameters `filename` : {str}

Filename and Path of the destination (.ido will be added if no .*-ending is available)

dataObject : {DataObject}

An allocated dataObject of n-Dimensions.

tagsAsBinary : {bool}, optional

Optional tag to toggle if numeric-tags should be saved (metaData) as binary or by default as string.

Notes

This function writes an itom::dataObject to the file specified by filename. The data is stored as binary. The value of string-Tags is encoded to avoid XML-conflicts. The value of numerical-Tags are saved as string with 15 significant digits (>32bit, tagsAsBinary == False, default) or as binary (tagsAsBinary == True). Tagnames which contains special characters leads to XML-conflicts.

`itom.loadMatlabMat(filename)` → loads matlab mat-file by using scipy methods and returns the loaded dictionary.

Parameters `filename` : {str}

Filename from which the data will be imported (.mat will be added if not available)

Returns PyDictionary with content of the Matlab-file :

Notes

This function loads matlab mat-file by using scipy methods and returns the loaded dictionary.

`itom.saveMatlabMat(filename, dictionary[, matrixName])` → saves strings, numbers, arrays or combinations to a Matlab matrix.

Parameters `filename` : {str}

Filename to which the data will be written (.mat will be added if not available)

dictionary : {dictionary, list, tuple}

dictionary, list or tuple with elements of type number, string, array (dataObject, numpy.ndarray, npDataObject...)

matrix-name : {string or list or tuple of strings}, optional

string or list or tuple of string (same length than object-sequence)

Notes

This function saves strings, numbers, arrays or combinations to a Matlab matrix (file). List or Tuples will be included to new dictionary (element-wise entry with name matrix1,...,matrixN or names given by last optional matrix-name sequence).

`itom.loadIDC(filename)` → loads a pickled idc-file and returns the content as dictionary

This methods loads the given idc-file using the method load from the python-buildin module pickle and returns the loaded dictionary.

Parameters `filename` : {String}

absolute filename or filename relative to the current directory.

See Also:

`pickle.load`

`itom.saveIDC(filename, dict[, overwriteIfExists = True])` → saves the given dictionary as pickled idc-file.

This method saves the given dictionary as pickled idc-file using the method dump from the builtin module pickle.

Parameters `filename` : {string}

absolute filename or filename relative to the current directory.

dict : {dict}

dictionary which should be pickled.

overwriteIfExists : {bool}, default: True

if True, an existing file will be overwritten.

See Also:

`pickle.dump`

9.1.6 Debug-Tools

`itom.gcEndTracking()` → compares the current object list of the garbage collector with the recently saved list.

`itom.gcStartTracking()` → stores the current object list of the garbage collector.

`itom.getDebugger()` → returns new reference to debugger instance

9.1.7 Further commands

`itom.scriptEditor()` -> opens new, empty script editor window (undocked)

Notes

Opens a new and empty itom script editor window. The window is undocked and non blocking.

`itom.openScript(filename)` → opens the given script in current script window.

Parameters `filename` : {str}

Path and File of the file to open. Unter windows not case sensitiv.

Notes

Open an existing itom script from the harddrive into the latest opened editor window.

`itom.newScript()` → opens an empty, new script in the current script window.

Notes

Creates a new itom script in the latest opened editor window.

`itom.setCurrentPath(newPath)` → sets absolute path of current working directory

Parameters `newPath` : {str}

The new working path of this application

Returns `Success` : {bool}

True in case of success else False

Notes

sets absolute path of current working directory returns True if currentPath could be changed, else False.

`itom.getAppPath()` → returns absolute path of application base directory.

Returns `Path` : {str}

string with absolute path of this application

Notes

This function returns the absolute path of application base directory. The return value is independent of the current working directory

`itom.getCurrentPath()` → returns absolute path of current working directory.

Returns `Path` : {str}

string with current working path

Notes

This function returns the current working path of the application.

`itom.getScreenInfo()` → returns dictionary with information about all available screens.

Returns `ScreenInfo` : {PyDict}

Returns a PyDictionary containing:

screenCount : {int}

number of available screens

primaryScreen : {int}

index (0-based) of primary screen

geometry : {tuple}

tuple with dictionaries for each screen containing data for width (w), height (h) and its top-left-position (x, y)

Notes

This function returns a PyDictionary which contains informations about the current screen configuration of this PC.

`itom.getDefaultScaleableUnits()`
getDefaultScaleAbleUnits() -> Get a PythonList with standard scaleable units.

Returns List with strings containing all scaleable units : {PyList}

Notes

Get a PythonList with standard scaleable units. Used together with `itom.ScaleValueAndUnit(...)`

`itom.scaleDoubleUnit()`
ScaleValueAndUnit(ScaleableUnits, value, valueUnit) -> Scale a value and its unit and returns [value, 'Unit']

Parameters ScaleableUnits : {PyList of Strings}

A string list with all scaleable units

value : {double}

The value to be scaled

valueUnit : {str}

The value unit to be scaled

Returns PyTuple with scaled value and scaled unit :

Notes

Rescale a value with SI-unit (e.g. 0.01 mm to 10 micrometer). Used together with `itom.getDefaultScaleAbleUnits()`

Another possibility to add methods to this page is to use the auto-summary function. Since, the default-role property in `conf.py` is set to 'autolink' and the auto-summary module is included, small pages will be automatically created for each method in the following list and a hyperlink to this site is created:

With method signature:

<code>itom.widgetHelp([widgetName, ...])</code>	Generates an online help for the given widget or returns a list of available widgets.
<code>itom.pluginHelp([pluginName [, ...]])</code>	Gets (also print to console) the initialisation parameters of the plugin specified plugin
<code>itom.filterHelp([filterName, ...])</code>	Generates an online help for the given widget or returns a list of available filter.

Without method signature:

<code>itom.liveImage((cam, ...))</code>	Creates a plot-image (2D) and automatically grabs images into this window.
<code>itom.plot((data, ...))</code>	Plot an existing dataObject in not dockable, not blocking window.

9.2 dataIO

class `itom.dataIO`
`dataIO(name[, mandparams, optparams])` -> constructor

Parameters name : {str}

is the fullname (case sensitive) of a 'dataIO'-plugin as specified in the plugin-window. init-

Parameters : {variant}, mandatory & optional Parameters to pass to the plugin, content and type depend on the specific plugin.

Returns Returns none or a PyDictionary depending on the value of detailLevel. :

Notes

This is the constructor for a dataIO-type plugins. It initializes an new instance if the plugin specified by 'name'. The initialisation parameters are parsed and unnamed parameters are used in their incoming order to fill first mandatory parameters and afterwards optional parameters. Parameters may be passed with name as well but after the first named parameter no more unnamed parameters are allowed. See pluginHelp(name) for detail information about the specific initialisation parameters.

acquire (*trigger=dataIO.TRIGGER_SOFTWARE*) → triggers the camera acquisition
Use this command to start the image acquisition depending on the trigger parameter.

Parameters **trigger** : {Integer}, optional

default = 0, dataIO.TRIGGER_SOFTWARE In case of dataIO.TRIGGER_SOFTWARE
(0) the acquisition is immediately started after this command.

copyVal (*dataObject*) → gets deep copy of data of this plugin, stored in the given data object.

Parameters **dataObject** : {doctodo}

Notes

The object is not changed / adepted to the grabber and must be allocated properly before copyVal is called

disableAutoGrabbing () → Disables auto grabbing for the grabber (camera...),

Notes

such that live images only will be updated if a new image is manually grabbed. [Recommended if the measurement routine requires camera images by itself.]

enableAutoGrabbing () → enables auto grabbing for the grabber (camera...),

Notes

such that live images will continuously get new data. [Recommended if the measurement routine does not need any camera image at the moment.]

exec (*funcName*[, *param1*, ...]) → invoke a function 'funcName' within an dataIO-plugin.

Parameters **funcName** : {str}

The name of the filter

paramN : {variant}

Further parameters depend on the function itself.

Returns Variable return values. :

The return values depend on the function itself.

Notes

This function is used to invoke a plugin-Specific `execFunc`, declared within the corresponding plugin. The parameters (arguments), output parameters / return values depends on the function (see `plugin.getExecFuncsInfo()` or `plugin.getExecFuncsInfo(funcName)`).

getAutoGrabbing () → returns the status of the auto grabbing flag.

Returns **auto grabbing flag** : {bool}

- false = auto grabbing off
- true = auto grabbing on.

See Also:

`enableAutoGrabbing`, `disableAutoGrabbing`

Notes

See methods `enableAutoGrabbing()` or `disableAutoGrabbing()`.

getExecFuncsInfo ([*funcName* [, *detailLevel*]]) → plots a list of available `execFuncs` or a detailed description to the specified `execFunc`.

Parameters **funcName** : {str}, optional

is the fullname or a part of any `execFunc`-name which should be displayed. If `funcName` is none or no `execFunc` matches `funcName` casesensitiv a list with all suitable `execFuncs` is given.

detailLevel : {dict}, optional

if dictionary == 1, function returns an `Py_Dictionary` with parameters Default value is 0.

Returns **None or Dict** :

depending on the value of *detailLevel*.

Notes

Generates an online help with all `execFuncs` for this plugin or returns a list of available `execFuncs`.

getParam (*name*) → returns the value of the given parameter.

Parameters **name** : {str???

Returns **Value of the given parameter** :

Notes

doctodo

getParamList () → returns list of possible parameters.

Returns **List of possible Parameters** :

Notes

doctodo

getParamListInfo ([*detailLevel*]) → plots informations about plugin parameters.

Parameters **detailLevel** : {dict}, optional

if dictionary == 1, function returns an Py_Dictionary with parameters Default value is 0.

Returns Returns none or a PyDictionary depending on the value of detailLevel. :

Notes

Generates an online help for available parameters and additional informations of the plugin.

getType () → returns dataIO type

Returns dataIO type :

Notes

doctodo

getVal (*buffer=dataObject|byteArray|bytes* [, *length=maxlength*]) → returns shallow copy of internal camera image if dataObject-buffer is provided. Else values from plugin are copied to given byte or byte-array buffer.

Parameters doctodo :

Returns shallow copy of internal camera image :

Notes

Cameras, Grabber: - buffer (dataObject), no length value: The image in dataObject is only a shallow copy of the camera internal memory. Therefore this content

may change if a new image has been acquired by the camera. Therefore consider to make a deep copy of this image or use the method copyVal.

further IO-devices: - buffer (allocated byteArray, bytes...) and optional a length with the maximum number of characters which should be requested by the plugin. If length is not provided it is set to the length of the given buffer. Finally the number of effectively set characters is returned.

hideToolbox () → hides toolbox of the plugin

name () → returns name of plugin.

Returns Name of the Plugin : {str}

Notes

doctodo

setAutoGrabbing (*on*) → Sets auto grabbing of the grabber device on or off

Parameters on : {bool}

- TRUE = on
- FALSE = off

Notes

such that live images only will be updated if a new image is manually grabbed (on).

setParam (*name, value*) → sets value of parameter, given by name.

Parameters name : {str??}

value : {str, int, double, ...}

Notes

doctodo

setVal (*dataObject*) → transfers given dataObject to dataIO-plugin.

Parameters **dataObject** : {dataObject??}

Notes

doctodo

showConfiguration () → open configuration dialog of the plugin

showToolbox () → open toolbox of the plugin

startDevice ([*count=1*]) → starts the given dataIO-plugin.

If you call startDevice multiple times, the device is only started at the first call, the next calls only increment a internal counter. This is necessary, since every connected live image needs to start the device without knowledge about any previous start. A call to stopDevice decrements this counter and closes the hardware device if that counter drops to 0 again. No acquisition is possible, if the device has not been started, hence the counter is 0.

Parameters **count** : {unsigned integer}, optional

default = 1

Notes

if count > 1, startDevice is executed 'count' times, in order to increment the grabber internal start counter.

stopDevice ([*count=1*]) → stops the given dataIO-plugin.

Usually no acquisition is possible, if the device is not started.

Parameters **count** : {Integer > 0}, optional

default = 1 if count > 1, stopDevice is executed 'count' times, in order to decrement the grabber internal start counter. You can also use -1 as count argument, then stopDevice is repeated until the internal start counter is 0. The number of effective counts is then returned

Returns **None** or the number of cycles that have been necessary to finally decrement the grabber's internal start counter to 0 (only if count==-1) :

9.3 actuator

class **itom.actuator**

actuator(name[, mandparams, optparams]) -> constructor

Parameters **name** : {str}

is the fullname (case sensitive) of a 'actuator'-plugin as specified in the plugin-window.

initParameters : {variant}, mandatory & optional

Parameters to pass to the plugin, content and type depend on the specific plugin.

Returns **Returns none** or a **PyDictionary** depending on the value of **detailLevel**. :

Notes

This is the constructor for a actuator-type plugins. It initializes an new instance if the plugin specified by 'name'. The initialisation parameters are parsed and unnamed parameters are used in their incoming order to fill first mandatory parameters and afterwards optional parameters. Parameters may be passed with name as well but after the first named parameter no more unnamed parameters are allowed. See `pluginHelp(name)` for detail information about the specific initialisation parameters.

calib (*axis*[, *axis1*, ...]) → starts calibration of given axes (0-based).

Parameters **axis** : {axis??}

axis that should be calibrated

Notes

doctodo

exec (*funcName*[, *param1*, ...]) → invoke a function 'funcName' within an actuator-plugin.

Parameters **funcName** : {str}

The name of the filter

param1 : {variant}

Further parameters depend on the function itself.

Returns **Variable return values.** :

The return values depend on the function itself.

Notes

This function is used to invoke a plugIn-Specific `execFunc`, declared within the corresponding plugin. The parameters (arguments), output parameters / return values depends on the function (see `plugin.getExecFuncsInfo()` or `plugin.getExecFuncsInfo(funcName)`).

getExecFuncsInfo ([*funcName*[, *detailLevel*]]) → plots a list of available `execFuncs` or a detailed description to the specified `execFunc`.

Parameters **funcName** : {str}, optional

is the fullname or a part of any `execFunc`-name which should be displayed. If `funcName` is none or no `execFunc` matches `funcName` casesensitiv a list with all suitable `execFuncs` is given.

detailLevel : {dict}, optional

if dictionary == 1, function returns an `Py_Dictionary` with parameters Default value is 0.

Returns **None or Dict** :

depending on the value of *detailLevel*.

Notes

Generates an online help with all `execFuncs` for this plugIn or returns a list of available `execFuncs`.

getParam (*name*) → value of the parameter 'name'.

Parameters **name** : {str??}

name of the parameter to get value for

Returns **doctodo** :

Notes

doctodo

getParamList () → returns the list of available parameters of the plugin

Returns list of available parameters of the plugin :

Notes

doctodo

getParamListInfo ([*detailLevel*]) → plots informations about plugin parameters.

Parameters *detailLevel* : {dict}, optional

if dictionary == 1, function returns an Py_Dictionary with parameters Default value is 0.

Returns Returns none or a PyDictionary depending on the value of *detailLevel* .

Notes

Generates an online help for available parameters and additional informations of the plugin.

getPos (*axis*[, *axisI*, ...]) → returns the actual positions of the given axes.

Parameters *axis* : {axis??}

axis for which the position should be returned

Returns doctodo :

Notes

doctodo

getStatus () → retrieve the actuator status.

Returns doctodo :

Notes

doctodo

getType () → returns actuator type

Returns actuator type :

Notes

doctodo

hideToolbox () → hides toolbox of the plugin

name () → returns the plugin name

Returns name of the Plugin :

Notes

doctodo

setOrigin (*axis* [, *axis1*, ...]) → defines the actual position of the given axes to value 0.

Parameters **axis** : {axis}

axis for which the origin should be set

Notes

doctodo

setParam (*name*, *value*) → sets parameter 'name' to the given value.

Parameters **name** : {str??}

name of the parameter which value is set

value : {str, int, double, ...}

value that will be set. The value is checked against the param's parameter definition before invoking the setParam method

Notes

doctodo

setPosAbs (*axis0*, *pos0* [, *axis1*, *pos1*, ...]) → moves axis to given absolute value (in mm).

Parameters **axis** : {axis??}, optional

axis that should be moved absolute

pos : {??}

new position for axis

Notes

doctodo

setPosRel (*axis*, *pos* [, *axis1*, *pos1*, ...]) → relatively moves given axes by the given distances [in mm].

Parameters **axis** : {axis??}

axis that should be moved relative

pos : {??}

position increment/decrement for axis

Notes

doctodo

showConfiguration () → open configuration dialog of the plugin

showToolbox () → open toolbox of the plugin

9.4 Algorithms, Widgets and Filters

The algorithm-plugins are container for **filters** and **widgets**. They do not have an own instance.

Filters are called by the **filter(...)**-Methods. To get an online help use **filterHelp(...)**. Widgets are used with **ui**-dialogs. To get an online help use **widgetHelp(...)**.

`itom.filter(name[, furtherParameters, ...])` → invoke a filter (or algorithm) function within an algorithm-plugin.

Parameters **name** : {str}

The name of the filter

furtherParameters : {variant}

Further parameters depend on the filter-methods itself (give the mandatory and then optional parameters in their defined order).

Returns **output parameters** : {variant}

The returned values depend on the definition of each filter. In general it is a tuple of all output parameters that are defined by the filter function.

Notes

This function is used to invoke itom filter-functions or algorithms, declared within itom-algorithm plugins. The parameters (arguments) depends on the specific filter function (see filterHelp(name)), By filterHelp() a list of available filter functions is retrieved.

`itom.filterHelp([filterName, dictionary = 0, furtherInfos = 0])` → generates an online help for the given filter(s).

Generates an online help for the given widget or returns a list of available filter.

Parameters **filterName** : {str}, optional

is the fullname or a part of any filter-name which should be displayed. If filterName is none or no filter matches filterName casesensitiv a list with all suitable filters is given.

dictionary : {dict}, optional

if dictionary == 1, function returns an Py_Dictionary with parameters Default value is 0.

furtherInfos : {int}, optional

defines if a complete parameter-list of name-related filters to the given filterName should be displayed (1)

Returns Returns none or a PyDictionary depending on the value of dictionary. :

`itom.widgetHelp([widgetName, dictionary = 0, furtherInfos = 0])` → generates an online help for the given widget(s).

Generates an online help for the given widget or returns a list of available widgets.

Parameters **widgetName** : {str}, optional

is the fullname or a part of any widget-name which should be displayed. If widgetName is none or no widget matches widgetName casesensitiv a list with all suitable widgets is given.

dictionary : {dict}, optional

if dictionary == 1, function returns an Py_Dictionary with parameters Default value is 0.

furtherInfos : {int}, optional

defines if a complete parameter-list of name-related widgets to the given widgetName should be displayed (1)

Returns Returns none or a PyDictionary depending on the value of dictionary. :

9.5 figure

class `itom.figure`

Bases: `itom.uiItem`

`figure([handle, [rows = 1, cols = 1]])` -> creates figure window.

The class `itom.figure` represents a standalone figure window, that can have various subplots. If an instance of this class is created without further parameters a new figure is created and opened having one subplot area (currently empty) and the numeric handle to this figure is returned:

```
h = figure()
```

Subplots are arranged in a regular grid whose size is defined by the optional parameters 'rows' and 'cols'. If you create a figure instance with a given handle, the instance is either a reference to an existing figure that has got this handle or if it does not exist, a new figure with the desired handle is opened and the handle is returned, too.

Parameters `handle` : {int}

numeric handle of the desired figure.

rows : {int, default: 1}

number of rows this figure should have (defines the size of the subplot-grid)

cols : {int, default: 1}

number of columns this figure should have (defines the size of the subplot-grid)

static `close (handle|'all')` -> static method to close any specific or all open figures (unless any figure-instance still keeps track of them)

This method closes and deletes any specific figure (given by handle) or all opened figures.

Parameters `handle` : {dataIO-Instance}

any figure handle (>0) or 'all' in order to close all opened figures

Notes

If any instance of class 'figure' still keeps a reference to any figure, it is only closed and deleted if the last instance is deleted, too.

hide () → hides figure without deleting it

liveImage (`cam` [, `areaIndex`, `className`]) → shows a camera live image in the current or given area of this figure

Creates a plot-image (2D) and automatically grabs images into this window. This function is not blocking.

Parameters `cam` : {dataIO-Instance}

Camera grabber device from which images are acquired.

areaIndex: {int}, optional :

className : {str}, optional

class name of desired plot (if not indicated default plot will be used (see application settings))

plot (`data` [, `areaIndex`, `className`]) → plots a dataObject in the current or given area of this figure

Plot an existing dataObject in not dockable, not blocking window. The style of the plot will depend on the object dimensions. If x-dim or y-dim are equal to 1, plot will be a lineplot else a 2D-plot.

Parameters `data` : {DataObject}

Is the data object whose region of interest will be plotted.

areaIndex: {int}, optional :

className : {str}, optional

class name of desired plot (if not indicated default plot will be used (see application settings))

show() → shows figure

subplot (index) → returns plotItem of desired subplot

This method closes and deletes any specific figure (given by handle) or all opened figures.

Parameters index : {unsigned int}

index to desired subplot. The subplot at the top, left position has the index 0 whereas the index is incremented row-wise.

docked

dock status of figure (True/False)

this attribute controls the dock appearance of this figure. If it is docked, the figure is integrated into the main window of itom, else it is a independent window.

handle

returns handle of figure

9.6 plotItem

class itom.plotItem

Bases: `itom.uiItem`

`plotItem(figure[, subplotIdx])` -> instance of the plot or subplot of a figure.

Use can use this constructor to access any plot or subplot (if more than one plot) of a figure. The subplotIndex row-wisely addresses the subplots, beginning with 0.

Parameters doc :

pickPoints (points [,maxNrPoints]) -> *method to let the user pick points on a plot (only if plot supports this)*

This method lets the user select one or multiple points (up to maxNrPoints) at the current plot (if the plot supports this).

Parameters points : {DataObject}

resulting data object containing the 2D positions of the selected points [2 x nrOfSelectedPoints].

maxNrPoints: {int}, optional :

let the user select up to this number of points [default: infinity]. Selection can be stopped pressing Space or Esc.

9.7 ui-elements (GUI)

9.7.1 uitem-Class

class itom.uiItem

call (slotOrPublicMethod[, argument1, argument2, ...]) → calls any public slot of this widget or any accessible public method.

Parameters slotOrPublicMethod : {str}

name of the slot or method

arguments : {various types}, optional

Here you must indicate every argument, that the definition of the slot indicates. The type must be convertible into the requested C++ based argument type.

Notes

Use this method, to invoke any public slot or wrapped method of the underlying *uiltem*. For instance, see the Qt-help for slots of the widget of element you are wrapping by this instance of *uiltem*.

connect (*signalSignature*, *callableMethod*) → connects the signal of the widget with the given callable python method

Parameters **signalSignature** : {str}

This must be the valid signature, known from the Qt-method *connect* (e.g. 'clicked(bool)')

callableMethod : {python method or function}

valid method or function that is called if the signal is emitted.

See Also:

`disconnect`, `invokeKeyboardInterrupt`

Notes

This instance of *uiltem* wraps a widget, that is defined by a C++-class, that is finally derived from *QWidget*. See Qt-help for more information about the capabilities of every specific widget. Every widget can send various signals. Use this method to connect any signal to any callable python method (bounded or unbounded). This method must have the same number of arguments than the signal and the types of the signal definition must be convertible into a python object.

disconnect (*signalSignature*, *callableMethod*) → disconnects a connection which must have been established with exactly the same parameters.

Parameters **signalSignature** : {str}

callableMethod : {python method or function}

Notes

doctodo

getAttribute (*attributeNumber*) → returns specified attribute of corresponding widget.

Parameters **attributeNumber** : {int}

Notes

doctodo

getProperty (*propertyName* | *listOfPropertyNames*) -> returns tuple of requested properties (single property or tuple of properties)

Use this method or the operator [] in order to get the value of one specific property of this widget or of multiple properties. Multiple properties are given by a tuple or list of property names. For one single property, its value is returned as it is. If the property names are passed as sequence, a sequence of same size is returned with the corresponding values.

Parameters **property** : {string, string-list}

Name of one property or sequence (tuple,list...) of property names

Returns returns the value of one single property or a list of values, if a sequence of names is given as parameter. :

See Also:

`setProperty`

Notes

doctodo

getPropertyInfo (*[propertyName]*) → returns information about the property ‘propertyName’ of this widget or all properties, if no name indicated.

Parameters *propertyName* : {tuple}, optional

Notes

doctodo

getWindowFlags (*flags*) → gets window flags of corresponding widget.

The flags-value is an or-combination of the enumeration Qt::WindowFlag. See Qt documentation for more information.

invokeKeyboardInterrupt (*signalSignature*) → connects the given signal with a slot immediately invoking a python interrupt signal.

Parameters *signalSignature* : {str}

This must be the valid signature, known from the Qt-method *connect* (e.g. ‘clicked(bool)’)

See Also:

`connect`

Notes

If you use the connect method to link a signal with a python method or function, this method can only be executed if python is in an idle status. However, if you want raise the python interrupt signal if a specific signal is emitted, this interruption should be immediately invoked. Therefore

setAttribute (*attributeNumber, value*) → sets attribute of corresponding widget.

Parameters *attributeNumber* : {int}

value : {bool}

Notes

doctodo

setProperty (*propertyDict*) → each property in the parameter dictionary is set to the dictionaries value.

Parameters *propertyDict* : {dict}

Dictionary with properties (keyword) and the values that should be set.

See Also:

`getProperty`

setWindowFlags (*flags*) → sets window flags of corresponding widget.

Parameters **flags** : {int}

window flags to set (or-combination, see Qt::WindowFlags) :

9.7.2 ui-Class

class `itom.ui`

Bases: `itom.uiItem`

`ui(filename, [type, dialogButtonBar, dialogButtons, childOfMainWindow, deleteOnClose])` → instance of user interface

The class **ui** wraps a user interface, externally designed and given by a ui-file. If your user interface is a dialog or window, chose `ui.TYPEWINDOW` as type, if the user interface is a widget (simplest case), chose `ui.TYPEDIALOG` and your widget will be embedded in a dialog, provided by *itom*. This dialog can be equipped with a button bar, whose buttons are already connected to *itom* internal methods. If you then show your dialog in a modal mode, *itom* knows which button has been clicked in order to accept or reject the dialog.

Parameters **filename** : {str}

path to user interface file (**.ui*), absolute or relative to current directory

type : {int}, optional

display type:

- 0 (`ui.TYPEDIALOG`): ui-file is embedded in auto-created dialog (default),
- 1 (`ui.TYPEWINDOW`): ui-file is handled as main window,
- 2 (`ui.TYPEDOCKWIDGET`): ui-file is handled as dock-widget and appended to the main-window dock area

dialogButtonBar : {int}, optional

Only for type `ui.TYPEDIALOG` (0). Indicates whether buttons should automatically be added to the dialog:

- 0 (`ui.BUTTONBAR_NO`): do not add any buttons (default)
- 1 (`ui.BUTTONBAR_HORIZONTAL`): add horizontal button bar
- 2 (`ui.BUTTONBAR_VERTICAL`): add vertical button bar

dialogButtons : {dict}, optional

every dictionary-entry is one button. key is the role, value is the button text

childOfMainWindow : {bool}, optional

for type `TYPEDIALOG` and `TYPEWINDOW` only. Indicates whether window should be a child of itom main window (default: True)

deleteOnClose : {bool}, optional

Indicates whether window should be deleted if user closes it or if it is hidden (default: Hidden, False)

static **createNewPluginWidget** (*widgetName* [, *mandparams*, *optparams*]) → creates widget defined by any algorithm plugin and returns the instance of type 'ui'

Parameters **widgetName** : { }

name algorithm widget parameters to pass to the plugin. The parameters are parsed and unnamed parameters are used in their incoming order to fill first mandatory parameters and afterwards optional parameters. Parameters may be passed with name as well but after the first named parameter no more unnamed parameters are allowed.

Notes

doctodo

static **getDouble** (*title, label, defaultValue*[, *min, max, decimals=3*]) → shows a dialog to get a double value from the user

Parameters **title** : {str}

is the dialog title

label : {str}

is the label above the spin box

defaultValue : {double}, optional

is the default value in the spin box

min : {double}, optional

default = -2147483647.0 is the allowed minimal value

max : {double}, optional

default = 2147483647.0 is the allowed maximal value

decimals : {int}, optional

the maximum number of decimal places (default: 1)

Returns A tuple where the first value contains the current double value. The second value is True if the dialog has been accepted, else False. :

See Also:

[getInt](#), [getText](#), [getItem](#)

static **getExistingDirectory** (*caption, startDirectory*[, *options, parent*]) → opens a dialog to choose an existing directory

Parameters **caption** : {str}

is the caption of this dialog

startDirectory : {str}

is the start directory

options : {int}, optional

is an or-combination of the following options (see ‘QFileDialog::Option’):

- 1: ShowDirsOnly [default]
- 2: DontResolveSymlinks
- ... (for others see Qt-Help)

parent : {ui}, optional

is a parent dialog or window, this dialog becomes modal.

Returns The selected directory is returned as absolute path or None if the dialog has been rejected. :

static **getInt** (*title, label, defaultValue*[, *min, max, step=1*]) → shows a dialog to get an integer value from the user

Parameters **title** : {str}

is the dialog title

label : {str}

is the label above the spinbox

defaultValue : {int}, optional

is the default value in the spinbox

min : {int}, optional

is the allowed minimal value (default: -2147483647)

max : {int}, optional

is the allowed maximal value (default: 2147483647)

step : {int}, optional

is the step size if user presses the up/down arrow (default: 1)

Returns A tuple where the first value contains the current integer value. The second value is True if the dialog has been accepted, else False. :

See Also:

`getDouble`, `getText`, `getItem`

static getItem (*title*, *label*, *stringList*[, *currentIndex*=0, *editable*=True]) → shows a dialog to let the user select an item from a string list

Parameters **title** : {str}

is the dialog title

label : {str}

is the label above the text box

stringList : {tuple or list}, optional

is a list or tuple of possible string values

currentIndex : {int}, optional

defines the preselected value index (default: 0)

editable : {bool}, optional

defines whether new entries can be added (True) or not (False, default)

Returns A tuple where the first value contains the current active or typed string value. The second value is True if the dialog has been accepted, else False. :

See Also:

`getInt`, `getDouble`, `getText`

static getOpenFileName ([*caption*, *startDirectory*, *filters*, *selectedFilterIndex*, *options*, *parent*]) → opens dialog for choosing an existing file.

Parameters **caption** : {str}, optional

This is the optional title of the dialog, default: no title

startDirectory {str}, optional :

optional, if not indicated currentDirectory will be taken

filters : {str}, optional

default = 0 possible filter list, entries should be separated by ;; , e.g. 'Images (.png *.jpg);;Text files (.txt)'

selectedFilterIndex : {int}, optional

is the index of filters which is set by default (0 is first entry)

options : {int}, optional

default = 0 or-combination of enum values `QFileDialog::Options`

parent : {ui}, optional

is the parent widget of this dialog

Returns filename as string or None if dialog has been aborted. :

See Also:

`getSaveFileName`

static getSaveFileName (*[caption, startDirectory, filters, selectedFilterIndex, options, parent]*)
→ opens dialog for choosing a file to save.

Parameters **caption** : {str}, optional

This is the title of the dialog

startDirectory : {String}, optional

if not indicated, the current working directory will be taken

filters : {str}, optional

possible filter list, entries should be separated by ;, e.g. 'Images (.png *.jpg);;Text files (.txt)'

selectedFilterIndex : {int}, optional

default = 0 is the index of filters which is set by default (0 is first entry)

options : {int}, optional

default = 0 or-combination of enum values `QFileDialog::Options`

parent : {ui}, optional

is the parent widget of this dialog

Returns filename as string or None if dialog has been aborted. :

See Also:

`getOpenFileName`

static getText (*title, label, defaultString*) → opens a dialog in order to ask the user for a string

Parameters **title** : {str}

is the dialog title

label : {str}

is the label above the text box

defaultString : {str}

is the default string in the text box

Returns A tuple where the first value contains the current string value. The second value is True if the dialog has been accepted, else False. :

See Also:

`getInt`, `getDouble`, `getItem`

hide () → hides initialized UI-Dialog

Notes

doctodo

isVisible () → returns true if dialog is still visible

Returns **dialog visible** : {bool}

Notes

doctodo

static msgCritical (*title, text* [, *buttons, defaultButton, parent*]) → opens a critical message box

Parameters **title** : {str}

is the message box title

text : {str}

is the message text

buttons : {int}, optional

is an or-combination of ui.MsgBox[...] -constants indicating the buttons to display. Use | for the or-combination.

defaultButton : {int}, optional

is a value of ui.MsgBox[...] which indicates the default button

parent : {ui}, optional

is the parent dialog of the message box.

Notes

doctodo

static msgInformation (*title, text* [, *buttons, defaultButton, parent*]) → opens an information message box

Parameters **title** : {str}

is the message box title

text : {str}

is the message text

buttons : {int}, optional

is an or-combination of ui.MsgBox[...] -constants indicating the buttons to display. Use | for the or-combination.

defaultButton : {int}, optional

is a value of ui.MsgBox[...] which indicates the default button

parent : {ui}, optional

is the parent dialog of the message box.

Notes

doctodo

static msgQuestion (*title*, *text*[, *buttons*, *defaultButton*, *parent*]) → opens a question message box

Parameters **title** : {str}

is the message box title

text : {str}

is the message text

buttons : {int}, optional

is an or-combination of ui.MsgBox[...] -constants indicating the buttons to display. Use | for the or-combination.

defaultButton : {int}, optional

is a value of ui.MsgBox[...] which indicates the default button

parent : {ui}, optional

is the parent dialog of the message box.

static msgWarning (*title*, *text*[, *buttons*, *defaultButton*, *parent*]) → opens a warning message box

Parameters **title** : {str}

is the message box title

text : {str}

is the message text

buttons : {int}, optional

is an or-combination of ui.MsgBox[...] -constants indicating the buttons to display. Use | for the or-combination.

defaultButton : {int}, optional

is a value of ui.MsgBox[...] which indicates the default button

parent : {ui}, optional

is the parent dialog of the message box.

Notes

doctodo

show (*modal*) → shows initialized UI-Dialog

Parameters **modal** : {int}

- 0: non-modal (default)
- 1: modal (python waits until dialog is hidden)
- 2: modal (python returns immediately)

9.7.3 timer-Class

class `itom.timer`

`timer(interval, callbackFunc, [argTuple])` -> new callback timer

Creates a timer object that continuously calls a python callback function or method with a certain interval. The timer is active after construction and stops when this instance is destroyed or `stop()` is called.

Parameters `interval` : {double}

time out interval in ms

callbackFunc: {function or method} :

Python function that should be called when timer event raises

argTuple: {tuple}, optional :

tuple of parameters passed as arguments to the callback function

isActive () → returns timer status

Returns `status` : {bool}

True if the timer is running, otherwise False.

setInterval (*interval*) → sets timer interval in [ms]

This method sets the timeout interval in milliseconds. The timer calls the callback function continuously after this interval (if started)

start () → starts timer

Starts or restarts the timer with its timeout interval. If the timer is already running, it will be stopped and restarted.

stop () → stops timer

9.8 dataObject

class `itom.dataObject`

`dataObject([dims [, dtype='uint8'[, continuous = 0][, data = valueOrSequence]])` -> constructor to get a new dataObject.

Parameters `dims` : {List of Integer}, optional

'dims' is list indicating the size of each dimension, e.g. [2,3] is a matrix with 2 rows and 3 columns. If not given, an empty data object is created.

dtype : {str}, optional

'dtype' is the data type of each element, possible values: 'int8', 'uint8', ..., 'int32', 'uint32', 'float32', 'float64', 'complex64', 'complex128'

continuous : {str}, optional

'continuous' [0|1] defines whether the data block should be continuously allocated in memory [1] or in different smaller blocks [0] (recommended for huge matrices).

data : {str}, optional

'data' is a single value or a sequence with the same amount of values than the data object. The values from data will be assigned to the new data object (filled row by row).

Returns `dataObject {dataObject}` :

Notes

The `itom.dataObject` contains a n-dimensional matrix and corresponding meta-data. It has a direct c++-representation for the python-c++ interface. The n-dimensional matrix can have different element types. Recently the following types are supported:

- Integer-type (`int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`),
- Floating-type (`float32`, `float64` ($\Rightarrow$ `double`)),
- Complex-type (`complex64` (`2x float32`), `complex128` (`2x float64`)).

Warning ‘`uint32`’ is not fully openCV-compatible and hence causes instability!

In order to handle huge matrices, the data object can divide one matrix into subparts in memory. Each subpart (called matrix-plane) is two-dimensional and covers data of the last two dimensions. In c++-context each of these matrix-planes is of type `cv::Mat_<type>` and can be used with every operator given by the openCV-framework (version 2.3.1 or higher).

The dimensions of the matrix are structured descending. So if we assume to have a n-dimensional matrix A, where each dimension has its size `s_i`, the dimensions order is n, ..., z, y, x and the corresponding sizes of A are `[s_n, s_(n-1), s_(n-2), ..., s_y, s_x]`.

In order to make the data object compatible to continuously organized data structures, like numpy-arrays, it is also possible to have all matrix-planes in one data-block in memory (not recommended for huge matrices). Nevertheless, the indicated data structure with the two-dimensional sub-matrix-planes is still existing. The data organization is equal to the one of open-cv, hence, two-dimensional matrices are stored row-by-row (C-style)...

Deep Copy, Shallow Copy and ROI

It is possible to set a n-dimensional region of interest (ROI) to each matrix, the virtual dimensions, which will be delivered if the user asks for the matrix size. To avoid copy operations where possible a simple `=_Operator` will also make a shallow copy of the object. Shallow copies share the same data (elements and meta data) with the original object, hence manipulations of one object will affect the original object and all shallow copies.

The opposite a deep copy of a `dataObject` (by `sourceObject.copy()`) creates a complete new matrix with own meta data object.

Example:

```
#Create an object
dObj = dataObject([5, 10, 10], 'int8')

# Make a shallow copy
dObjShallow = dObj

# Make a shallow copy on ROI
dObjROI = dObj[1, :, :]

# Set the value of element [1, 0, 0] to 0
dObj[1, 0, 0] = 0

# Make a deep copy of the dObjROI
dObjROICopy = dObjROI.copy()

# Set the value of dObjROICopy element [0, 0, 0] to 127 without effecting other objects
dObjROICopy[0, 0, 0] = 127
```

Constructor The function `dataObject([dims [, dtype='uint8'[, continuous = 0][, data = valueOrSequence]])` creates a new itom-dataObject filled with undefined data. If no parameters are given, an uninitialized DataObject (`dims = 0`, no sizes) is created.

As second possibility you can also use the copy-constructor ‘`dataObject(AnyArray)`’, where `AnyArray` must be any array-like structure which is parsable by the numpy-interface.

Further static methods for object construction are `dataObject.zeros(..)`, `dataObject.ones(..)`, `dataObject.rand(..)`, `dataObject.randN(..)` and `dataObject.eye(..)`.

addToProtocol (`[newLine]`) → Appends a protocol line to the protocol. ROIs are added automatically.

Parameters `newLine` : {str}

The text to be added to the protocol.

Notes

Appends a protocol line to the protocol. The current ROIs of the `dataObject` are added automatically. Also a new line is added automatically to the protocol. The content of the protocol can be addressed by `'dataObject.tags['protocol']'`

adj () -> *Adjugate all elements (inline)*

Notes

doctodo

adjugate () → Return the adjugate, element-wise

Notes

doctodo

adjustROI ()

Notes

doctodo

astype () → todo

Notes

doctodo

conj () → complex-conjugates all elements of this `dataObject` (inline).

Returns doctodo :

Notes

Every value of this `dataObject` is replaced by its complex-conjugate value. If the data type of this `dataObject` is no complex data type, a `TypeError` is raised.

conjugate () → returns a copy of this `dataObject` where every element is complex-conjugated.

Returns copy of this `dataObject` :

Notes

If the data type of this dataObject is no complex data type, a TypeError is raised.

copy (*region_only=0*) → todo

Parameters **regionOnly** : { }, optional

Notes

doctodo

data () → prints the content of the dataObject in a readable form.

Notes

When calling this method, the complete content of the dataObject is printed to the standard output stream.

deleteTag (*key*) → Delete the tag specified by key.

Parameters **key** : {str}

the name of the tag to be deleted

Notes

Remove a tag from the tag space the tag space.

div () → a.div(b) returns element wise division of a./b and returns result

Parameters **doctodo** :

Returns **doctodo** :

Notes

doctodo

existTag (*key*) → return 1 if tag exists, else return 0.

Parameters **key** : {str}

the name of the tag

Returns **checkExists** : {bool}

Notes

Checks if a specific tag already exists and return the result as a boolean value.

static eye ([*dims* [, *dtype='uint8'* [, *continuous = 0*]]]) → creates a 2D, square, eye-matrix.

Parameters **size** : {int},

the size of the square matrix (single value)

dtype : {str}, optional

'dtype' is the data type of each element, possible values:
'int8', 'uint8', ..., 'int32', 'uint32', 'float32', 'float64', 'complex64', 'complex128'

Returns **The dataObject** : {dataObject}

Notes

Static method for creating a two-dimensional, square, eye-matrix of type itom-dataObject.

getTagListSize () → returns the number of tags in tag map

Notes

doctodo

locateROI () → todo

Notes

locateROI returns a tuple with two elements. The first is a list with the original sizes of this matrix, the second is a list with the offset for each axe in order to get from the original first element to the first element in the given ROI

makeContinuous () → return continuous representation of dataObject (if not continuous yet, else returns shallow copy of original data object)

Returns doctodo :

Notes

doctodo

mul () → a.mul(b) returns element wise multiplication of a*b and returns result

Parameters doctodo :

Notes

doctodo

name () → prints the content of the dataObject in a readable form.

Notes

doctodo

static ones ([*dims* [, *dtype*='uint8' [, *continuous* = 0]]]) → creates new dataObject filled with ones.

Parameters *dims* : {List of Integer}

'dims' is list indicating the size of each dimension, e.g. [2,3] is a matrix with 2 rows and 3 columns

dtype : {str}, optional

'dtype' is the data type of each element, possible values: 'int8', 'uint8', ..., 'int32', 'float32', 'float64', 'complex64', 'complex128'

continuous : {str}, optional

'continuous' [0|1] defines whether the data block should be continuously allocated in memory [1] or in different smaller blocks [0] (recommended for huge matrices).

Returns **The dataObject** : {dataObject}

Notes

Creates a new itom-dataObject filled with ones.

static rand (`[dims[, dtype='uint8'[, continuous = 0]]]`) → creates new dataObject filled with uniform distributed random values.

Parameters **dims** : {List of Integer}

'dims' is list indicating the size of each dimension, e.g. [2,3] is a matrix with 2 rows and 3 columns.

dtype : {str}, optional

'dtype' is the data type of each element, possible values: 'int8', 'uint8', ..., 'int32', 'float32', 'float64', 'complex64', 'complex128'

continuous : {str}, optional

'continuous' [0|1] defines whether the data block should be continuously allocated in memory [1] or in different smaller blocks [0] (recommended for huge matrices).

Returns **The dataObject** : {dataObject}

Notes

Creates a new itom-dataObject filled with uniform distributed random numbers. In case of an integer type, the uniform noise is from min<ObjectType>(inclusiv) to max<ObjectType>(inclusiv). For floating point types, the noise is between 0(inclusiv) and 1(exclusiv).

static randN (`[dims[, dtype='uint8'[, continuous = 0]]]`) → creates new dataObject filled with gaussian distributed random values.

Parameters **dims** : {List of Integer}

'dims' is list indicating the size of each dimension, e.g. [2,3] is a matrix with 2 rows and 3 columns.

dtype : {str}, optional

'dtype' is the data type of each element, possible values: 'int8', 'uint8', ..., 'int32', 'float32', 'float64', 'complex64', 'complex128'

continuous : {str}, optional

'continuous' [0|1] defines whether the data block should be continuously allocated in memory [1] or in different smaller blocks [0] (recommended for huge matrices).

Returns **The dataObject** : {dataObject}

Notes

Creates a new itom-dataObject filled with gaussian distributed random numbers. In case of an integer type, the gaussian noise mean value is (max+min)/2.0 and the standard deviation is (max-min)/6.0 to max. For floating point types, the noise mean value is 0 and the standard deviation is 1.0/3.0.

reshape (`newSizes`) → Returns reshaped shallow copy of data object

Returns **reshaped shallow copy of data object** :

Notes

Not implemented yet.

setAxisDescription ($[axisNum[, axisDescription]]$) → Set the description of the specified axis.

Parameters **axisNum** : {int}

The addressed axis

axisDescription : {str}

New axis description

Notes

Set the axis description (e.g. intensity) of axisNum

setAxisOffset ($[axisNum[, axisOffset]]$) → Set the offset of the specified axis.

Parameters **axisNum** : {int}

The addressed axis

axisOffset : {double}

New axis offset in [px]

Notes

Set the axis offset in [px] of axisNum to axisOffset, Unit-Coordinate = (px-Coordinate - Offset)*
Scale

setAxisScale ($[axisNum[, axisScale]]$) → Set the scale of the specified axis.

Parameters **axisNum** : {int}

The addressed axis

axisScale : {double}

New axis scale in [unit / px]

Notes

Set the axis scale in [unit / px] of axisNum to axisScale, Unit-Coordinate = (px-Coordinate - Offset)*
Scale

setAxisUnit ($[axisNum[, axisUnit]]$) → Set the unit of the specified axis.

Parameters **axisNum** : {int}

The addressed axis

axisUnit : {str}

New axis unit

Notes

Set the axis unit (e.g. mm) of axisNum

setTag (*key*, *tagvalue*) → Set the value of tag specified by key.

Parameters **key** : {str}

the name of the tag to set

tagvalue : {str or double}

the new value of the tag, either string or double value

Notes

Set the value of a tag within the meta data tagspace. If tag does not exist, it is created. Do NOT use 'special character' within the tag key because they are not XML-save.

size (*[index]*) → returns the size of this dataObject (tuple of the sizes in all dimensions or size in dimension indicated by optional value index).

Parameters *index* : {PyDataObject}, optional

Notes

doctodo

squeeze () → returns a squeezed shallow copy (if possible) of this data object.

Returns **ShallowCopy** : {dataObject}

At least 2D or more object with all dimensions bigger than 1 except the x/y-Dimension.

Notes

This method removes every dimension with size equal to 1. Take care, that none of the last two dimensions is considered by this squeeze-command. The squeezed return value is a shallow copy of the original dataObject and hence changes in its values will also change the original data set. (This command is equal to numpy.squeeze)

tolist () → returns nested list of content of data object

Notes

doctodo

trans () → returns transposed matrix

Returns **doctodo** :

Notes

doctodo

static zeros (*[dims[, dtype='uint8'[, continuous = 0]]]*) → creates new dataObject filled with zeros.

Parameters *dims* : {List of Integer}

'dims' is list indicating the size of each dimension, e.g. [2,3] is a matrix with 2 rows and 3 columns

dtype : {str}, optional

'dtype' is the data type of each element, possible values:
'int8', 'uint8', ..., 'int32', 'uint32', 'float32', 'float64', 'complex64', 'complex128'

continuous : {str}, optional

'continuous' [0|1] defines whether the data block should be continuously allocated in memory [1] or in different smaller blocks [0] (recommended for huge matrices).

Returns **The dataObject** : {dataObject}

Notes

Creates a new itom-dataObject filled with zeros.

axisDescriptions

tuple containing the axis descriptions.

This attribute gives access to the internal axis descriptions expressed as a tuple strings. One entry for each object dimensions is due. This tuple is readonly, but you can either set an entire new tuple or change specific elements using the method `setAxisDescription(index,value)`

setter: Tuple of double value with size equal object dims.

getter: Readonly tuple of double value with size equal object dims.

Notes

{tuple of str} : ReadWrite

axisOffsets

tuple containing the axis offsets [px].

This attribute gives access to the internal axis offsets [px] expressed as a tuple double values. One entry for each object dimensions is due. This tuple is readonly, but you can either set an entire new tuple or change specific elements using the method `setAxisOffset(index,value)`.

Definition: $\text{Unit-Coordinate} = (\text{px-Coordinate} - \text{Offset}) * \text{Scale}$ setter: Tuple of double value with size equal object dims.

getter: Readonly tuple of double value with size equal object dims.

Notes

{tuple of double} : ReadWrite

axisScales

tuple containing the axis scales [unit/px].

This attribute gives access to the internal axis scales [unit/px] expressed as a tuple double values. One entry for each object dimensions is due. This tuple is readonly, but you can either set an entire new tuple or change specific elements using the method `setAxisScale(index,value)`

Definition: $\text{Unit-Coordinate} = (\text{px-Coordinate} - \text{Offset}) * \text{Scale}$ setter: Tuple of double value with size equal object dims. getter: Readonly tuple of double value with size equal object dims.

Notes

{tuple of double} : ReadWrite

axisUnits

tuple containing the axis units.

This attribute gives access to the internal axis units expressed as a tuple strings. One entry for each object dimensions is due. This tuple is readonly, but you can either set an entire new tuple or change specific elements using the method `setAxisUnit(index,value)`

setter: Tuple of double value with size equal object dims.

getter: Readonly tuple of double value with size equal object dims.

Notes

{tuple of str} : ReadWrite

base

base object

continuous

true if matrix is continuously organized, else false. If matrix is continuously organized this function returns true, else false. If true, the whole matrix is allocated in one huge block in memory, hence, this data object can be transformed into a numpy representation (npDataObject) without reallocating memory. Notes — {bool} : ReadOnly

dims

number of dimensions of this data object Notes — {int} : ReadOnly

dtype

get type string of data in this data object This attribute returns the dataObject-Type as a string. This type string has one of these values: 'uint8', 'int8', 'uint16', 'int16', 'uint32', 'int32', 'float32', 'float64', 'complex64', 'complex128'

Notes

{str} : ReadOnly

metaDict

new dictionary with all tags inside

Notes

{dict} : ReadOnly

tags

tag dictionary for this data object.

By this attribute you get access to the read-only tag dictionary or you can set an entire new dictionary. You can add single elements using the method setTag(key,value) or you can delete tags using deleteTag(key). The value of the tags can be either numeric or string values. Do NOT use 'special character' within the tag key because they are not XML-save. setter: The dataObject dictionary is filled with the entries in 'newDictionary'. getter: A readonly dictionary with all Metadata-Tags of the dataObject.

Notes

{bool} : ReadWrite

value

get/set the values within the ROI as a one-dimensional tuple.

This method gets or sets the values within the ROI. If this attribute is called by means of a getter, a tuple is returned which is created by iterating through the values of the data object (row-wise). In the same way of iterating, the values are set to the data object if you provide a tuple of the size of the data object or its ROI, respectively.

Example

```
b = dataObject[1,1:10,1,1].value
# or for the first value
b = dataObject[1,1:10,1,1].value[0]
# The elements of the tuple are adressed with b[idx].
```

valueDescription
value unit description.

Notes

{str} : ReadWrite

valueOffset
value offset [default: 0.0].

Notes

{float???} : ReadOnly

valueScale
value scale [default: 1.0].

Notes

{float???} : ReadWrite

valueUnit
value unit string.

Notes

{str} : ReadWrite

xyRotationalMatrix

Access the 3x3 rotational matrix in the dataObject tag space

This attribute gives access to the xyRotationalMatrix in the metaData-Tag space. The getter method returns a 3x3-Array deepcopied from the internal matrix, Implemented to offer compability to x3p format.

Notes

{3x3 array of doubles} : ReadWrite

9.9 npDataObject

class `itom.npDataObject`
npDataObject objects

all (*axis=None, out=None*)
Returns True if all elements evaluate to True.
Refer to `numpy.all` for full documentation.

See Also:

numpy.all equivalent function

any (*axis=None, out=None*)

Returns True if any of the elements of *a* evaluate to True.

Refer to `numpy.any` for full documentation.

See Also:

`numpy.any` equivalent function

argmax (*axis=None, out=None*)

Return indices of the maximum values along the given axis.

Refer to `numpy.argmax` for full documentation.

See Also:

`numpy.argmax` equivalent function

argmin (*axis=None, out=None*)

Return indices of the minimum values along the given axis of *a*.

Refer to `numpy.argmin` for detailed documentation.

See Also:

`numpy.argmin` equivalent function

argsort (*axis=-1, kind='quicksort', order=None*)

Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

See Also:

`numpy.argsort` equivalent function

astype (*dtype, order='K', casting='unsafe', subok=True, copy=True*)

Copy of the array, cast to a specified type.

Parameters **dtype** : str or dtype

Typecode or data-type to which the array is cast.

order : { 'C', 'F', 'A', or 'K' }, optional

Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

casting : { 'no', 'equiv', 'safe', 'same_kind', 'unsafe' }, optional

Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

subok : bool, optional

If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy : bool, optional

By default, `astype` always returns a newly allocated array. If this is set to false, and the `dtype`, `order`, and `subok` requirements are satisfied, the input array is returned instead of a copy.

Raises **ComplexWarning** :

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([ 1. ,  2. ,  2.5])

>>> x.astype(int)
array([1, 2, 2])
```

byteswap (*inplace*)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place.

Parameters **inplace**: bool, optional :

If True, swap bytes in-place, default is False.

Returns **out**: ndarray :

The byteswapped array. If *inplace* is True, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([ 256,      1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']
```

Arrays of strings are not swapped

```
>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
      dtype='<S3')
```

choose (*choices*, *out=None*, *mode='raise'*)

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

See Also:

numpy.choose equivalent function

clip (*a_min, a_max, out=None*)

Return an array whose values are limited to [*a_min*, *a_max*].

Refer to `numpy.clip` for full documentation.

See Also:

numpy.clip equivalent function

compress (*condition, axis=None, out=None*)

Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

See Also:

numpy.compress equivalent function

conj ()

Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

See Also:

numpy.conjugate equivalent function

conjugate ()

Return the complex conjugate, element-wise.

Refer to `numpy.conjugate` for full documentation.

See Also:

numpy.conjugate equivalent function

copy (*order='C'*)

Return a copy of the array.

Parameters **order** : {'C', 'F', 'A', 'K'}, optional

Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `numpy.copy` are very similar, but have different default values for their `order=` arguments.)

See Also:

`numpy.copy`, `numpy.copyto`

Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```



```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])

>>> y.flags['C_CONTIGUOUS']
True
```

cumprod (*axis=None, dtype=None, out=None*)

Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

See Also:

numpy.cumprod equivalent function

cumsum (*axis=None, dtype=None, out=None*)

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See Also:

numpy.cumsum equivalent function

diagonal (*offset=0, axis1=0, axis2=1*)

Return specified diagonals.

Refer to `numpy.diagonal()` for full documentation.

See Also:

numpy.diagonal equivalent function

dot (*b, out=None*)

Dot product of two arrays.

Refer to `numpy.dot` for full documentation.

See Also:

numpy.dot equivalent function

Examples

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

dump (*file*)

Dump a pickle of the array to the specified file. The array can be read back with `pickle.load` or `numpy.load`.

Parameters **file** : str

A string naming the dump file.

dumps()

Returns the pickle of the array as a string. `pickle.loads` or `numpy.loads` will convert the string back to an array.

Parameters **None** :

fill(value)

Fill the array with a scalar value.

Parameters **value** : scalar

All elements of *a* will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1.,  1.]
```

flatten(order='C')

Return a copy of the array collapsed into one dimension.

Parameters **order** : {'C', 'F', 'A'}, optional

Whether to flatten in C (row-major), Fortran (column-major) order, or preserve the C/Fortran ordering from *a*. The default is 'C'.

Returns **y** : ndarray

A copy of the input array, flattened to one dimension.

See Also:

ravel Return a flattened array.

flat A 1-D flat iterator over the array.

Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

getfield(dtype, offset=0)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype `complex128` has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters **dtype** : str or dtype

The data type of the view. The dtype size of the view can not be larger than that of the array itself.

offset : int

Number of bytes to skip before beginning the element view.

Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[ 1.+1.j,  0.+0.j],
       [ 0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[ 1.,  0.],
       [ 0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[ 1.,  0.],
       [ 0.,  4.]])
```

item(*args)

Copy an element of an array to a standard Python scalar and return it.

Parameters **args*: Arguments (variable number and type)

- *none*: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- *int_type*: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- *tuple of int_types*: functions as does a single *int_type* argument, except that the argument is interpreted as an nd-index into the array.

Returns *z*: Standard Python scalar object

A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of *a* is longdouble or clongdouble, *item()* returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for *item()*, unless fields are defined, in which case a tuple is returned.

item is very similar to *a[args]*, except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
```

```
>>> x.item((2, 2))
3
```

itemset (*args)

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array *a*.

Parameters *args : Arguments

If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Notes

Compared to indexing syntax, `itemset` provides some speed increase for placing a scalar into a particular location in an *ndarray*, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using `itemset` (and `item`) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])
```

max (axis=None, out=None)

Return the maximum along a given axis.

Refer to `numpy.amax` for full documentation.

See Also:

numpy.amax equivalent function

mean (axis=None, dtype=None, out=None)

Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

See Also:

numpy.mean equivalent function

min (axis=None, out=None)

Return the minimum along a given axis.

Refer to `numpy.amin` for full documentation.

See Also:

numpy.amin equivalent function

name()

newbyteorder (*new_order='S'*)

Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbytorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

Parameters *new_order* : string, optional

Byte order to force; a value from the byte order specifications above. *new_order* codes can be any of:

```
* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)
```

The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of 'B' or 'b' or 'bigish' are valid to specify big-endian.

Returns *new_arr* : array

New array object with the dtype reflecting given change to the byte order.

nonzero()

Return the indices of the elements that are non-zero.

Refer to `numpy.nonzero` for full documentation.

See Also:

numpy.nonzero equivalent function

prod (*axis=None, dtype=None, out=None*)

Return the product of the array elements over the given axis

Refer to `numpy.prod` for full documentation.

See Also:

numpy.prod equivalent function

ptp (*axis=None, out=None*)

Peak to peak (maximum - minimum) value along a given axis.

Refer to `numpy.ptp` for full documentation.

See Also:

numpy.ptp equivalent function

put (*indices, values, mode='raise'*)

Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to `numpy.put` for full documentation.

See Also:

numpy.put equivalent function

ravel (*[order]*)

Return a flattened array.

Refer to `numpy.ravel` for full documentation.

See Also:

numpy.ravel equivalent function

ndarray.flat a flat iterator on the array.

repeat (*repeats, axis=None*)

Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

See Also:

numpy.repeat equivalent function

reshape (*shape, order='C'*)

Returns an array containing the same data with a new shape.

Refer to `numpy.reshape` for full documentation.

See Also:

numpy.reshape equivalent function

resize (*new_shape, refcheck=True*)

Change shape and size of array in-place.

Parameters **new_shape** : tuple of ints, or *n* ints

Shape of resized array.

refcheck : bool, optional

If False, reference count will not be checked. Default is True.

Returns **None** :

Raises **ValueError** :

If *a* does not own its own data or references or views to it exist, and the data memory must be changed.

SystemError :

If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

See Also:

resize Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])

>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that has been referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

round (*decimals=0, out=None*)

Return *a* with each element rounded to the given number of decimals.

Refer to `numpy.around` for full documentation.

See Also:

numpy.around equivalent function

searchsorted (*v, side='left', sorter=None*)

Find indices where elements of *v* should be inserted in *a* to maintain order.

For full documentation, see `numpy.searchsorted`

See Also:

numpy.searchsorted equivalent function

setfield (*val, dtype, offset=0*)

Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by *dtype* and beginning *offset* bytes into the field.

Parameters *val* : object

Value to be placed in field.

dtype : dtype object

Data-type of the field in which to place *val*.

offset : int, optional

The number of bytes into the field at which to place *val*.

Returns **None** :

See Also:

`getfield`

Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]])
>>> x
array([[ 1.00000000e+000,  1.48219694e-323,  1.48219694e-323],
       [ 1.48219694e-323,  1.00000000e+000,  1.48219694e-323],
       [ 1.48219694e-323,  1.48219694e-323,  1.00000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

setflags (*write=None, align=None, uic=None*)

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters **write** : bool, optional

Describes whether or not *a* can be written to.

align : bool, optional

Describes whether or not *a* is aligned properly for its type.

uic : bool, optional

Describes whether or not *a* is a copy of another “base” array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by `.base`). When this array is deallocated, the base array will be updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well as the full name.

Examples

```
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

sort (*axis=-1*, *kind='quicksort'*, *order=None*)

Sort an array, in-place.

Parameters *axis* : int, optional

Axis along which to sort. Default is -1, which means sort along the last axis.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional

Sorting algorithm. Default is 'quicksort'.

order : list, optional

When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

See Also:

numpy.sort Return a sorted copy of an array.

argsort Indirect sort.

lexsort Indirect stable sort on multiple keys.

searchsorted Find elements in sorted array.

Notes

See `sort` for notes on the different sorting algorithms.

Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', '<S1'), ('y', '<i4')])
```

squeeze (*axis=None*)

Remove single-dimensional entries from the shape of *a*.

Refer to `numpy.squeeze` for full documentation.

See Also:

numpy.squeeze equivalent function

std (*axis=None, dtype=None, out=None, ddof=0*)

Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

See Also:

numpy.std equivalent function

sum (*axis=None, dtype=None, out=None*)

Return the sum of the array elements over the given axis.

Refer to `numpy.sum` for full documentation.

See Also:

numpy.sum equivalent function

swapaxes (*axis1, axis2*)

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to `numpy.swapaxes` for full documentation.

See Also:

numpy.swapaxes equivalent function

take (*indices, axis=None, out=None, mode='raise'*)

Return an array formed from the elements of *a* at the given indices.

Refer to `numpy.take` for full documentation.

See Also:

numpy.take equivalent function

tofile (*fid*, *sep*="", *format*="%s")

Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of *a*. The data produced by this method can be recovered using the function `fromfile()`.

Parameters *fid* : file or str

An open file object, or a string containing a filename.

sep : str

Separator between array items for text output. If "" (empty), a binary file is written, equivalent to `file.write(a.toststring())`.

format : str

Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using "format" % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

tolist()

Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

Parameters *none* :

Returns *y* : list

The possibly nested list of array elements.

Notes

The array may be recreated, `a = np.array(a.tolist())`.

Examples

```
>>> a = np.array([1, 2])
>>> a.tolist()
[1, 2]
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

toststring (*order*='C')

Construct a Python string containing the raw data bytes in the array.

Constructs a Python string showing a copy of the raw contents of data memory. The string can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the `F_CONTIGUOUS` flag in the array is set, in which case it means 'Fortran' order.

Parameters *order* : {'C', 'F', None}, optional

Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

Returns `s` : str

A Python string exhibiting a copy of *a*'s raw data.

Examples

```
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tostring()
'\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> x.tostring('C') == x.tostring()
True
>>> x.tostring('F')
'\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'
```

trace (*offset=0, axis1=0, axis2=1, dtype=None, out=None*)

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See Also:

`numpy.trace` equivalent function

transpose (**axes*)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an *n*-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ..., i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ..., i[1], i[0])`.

Parameters `axes` : None, tuple of ints, or *n* ints

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an *n*-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form)

Returns `out` : ndarray

View of *a*, with axes suitably permuted.

See Also:

`ndarray.T` Array property returning the array transposed.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
```

```
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

var (*axis=None, dtype=None, out=None, ddof=0*)

Returns the variance of the array elements, along given axis.

Refer to `numpy.var` for full documentation.

See Also:

numpy.var equivalent function

view (*dtype=None, type=None*)

New view of array with the same data.

Parameters **dtype** : data-type, optional

Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, results in the view having the same data-type as *a*.

type : Python type, optional

Type of the returned view, e.g., ndarray or matrix. Again, the default None results in type preservation.

Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print type(y)
<class 'numpy.matrixlib.defmatrix.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2), (3, 4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1, 2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([ 2.,  3.]
```

Making changes to the view changes the underlying array

```
>>> xv[0,1] = 20
>>> print x
[(1, 20) (3, 4)]
```

Using a view to convert an array to a record array:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

T

Same as `self.transpose()`, except that `self` is returned if `self.ndim < 2`.

Examples

```
>>> x = np.array([[1.,2.],[3.,4.]])
>>> x
array([[ 1.,  2.],
       [ 3.,  4.]])
>>> x.T
array([[ 1.,  3.],
       [ 2.,  4.]])
>>> x = np.array([1.,2.,3.,4.])
>>> x
array([ 1.,  2.,  3.,  4.])
>>> x.T
array([ 1.,  2.,  3.,  4.])
```

axisDescriptions

list with descriptions for each axis.

axisOffsets

list with offset values for each axis [px].

axisScales

list with scale values for each axis [unit/px].

axisUnits

list containing the units for each axis.

base

Base object if memory is from some other object.

Examples

The base of an array that owns its memory is `None`:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with `x`:

```
>>> y = x[2:]
>>> y.base is x
True
```

ctypes

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters None :

Returns **c** : Python object

Possessing attributes data, shape, strides, etc.

See Also:

`numpy.ctypeslib`

Notes

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

- data**: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self._array_interface_['data'][0]`.
- shape** (`c_intp*self.ndim`): A ctypes array of length `self.ndim` where the basetype is the C-integer corresponding to `dtype('p')` on this platform. This base-type could be `c_int`, `c_long`, or `c_longlong` depending on the platform. The `c_intp` type is defined accordingly in `numpy.ctypeslib`. The ctypes array contains the shape of the underlying array.
- strides** (`c_intp*self.ndim`): A ctypes array of length `self.ndim` where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.
- data_as(obj)**: Return the data pointer cast to a particular c-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: `self.data_as(ctypes.POINTER(ctypes.c_double))`.
- shape_as(obj)**: Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.
- strides_as(obj)**: Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(ctypes.c_void_p)` returns a pointer to memory that is invalid because the array created as `(a+b)` is deallocated before the next Python statement. You can avoid this problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, `ct` will hold a reference to the array until `ct` is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the `as_parameter` attribute which will return an integer equal to the data attribute.

Examples

```
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

data

Python buffer object pointing to the start of the array's data.

dtype

Data-type of the array's elements.

Parameters **None** :

Returns **d** : numpy dtype object

See Also:

`numpy.dtype`

Examples

```
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

flags

Information about the memory layout of the array.

Attributes **C_CONTIGUOUS (C)** :

The data is in a single, C-style contiguous segment.

F_CONTIGUOUS (F) :

The data is in a single, Fortran-style contiguous segment.

OWNDATA (O) :

The array owns the memory it uses or borrows it from another object.

WRITEABLE (W) :

The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable.)

However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a `RuntimeError` exception.

ALIGNED (A) :

The data and strides are aligned appropriately for the hardware.

UPDATEIFCOPY (U) :

This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.

FNC :

`F_CONTIGUOUS` and not `C_CONTIGUOUS`.

FORC :

`F_CONTIGUOUS` or `C_CONTIGUOUS` (one-segment test).

BEHAVED (B) :

`ALIGNED` and `WRITEABLE`.

CARRAY (CA) :

`BEHAVED` and `C_CONTIGUOUS`.

FARRAY (FA) :

`BEHAVED` and `F_CONTIGUOUS` and not `C_CONTIGUOUS`.

Notes

The `flags` object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the `UPDATEIFCOPY`, `WRITEABLE`, and `ALIGNED` flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling `ndarray.setflags`.

The array flags cannot be set arbitrarily:

- `UPDATEIFCOPY` can only be set `False`.
- `ALIGNED` can only be set `True` if the data is truly aligned.
- `WRITEABLE` can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

flat

A 1-D iterator over the array.

This is a `numpy.flatiter` instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See Also:

flatten Return a copy of the array collapsed into one dimension.

`flatiter`

Examples

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

imag

The imaginary part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0.          ,  0.70710678])
>>> x.imag.dtype
dtype('float64')
```

itemsize

Length of one array element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

metaDict

returns new dictionary with all tags inside

nbytes

Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

ndim

Number of array dimensions.

Examples

```
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

real

The real part of the array.

See Also:

numpy.real equivalent function

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.          ,  0.70710678])
>>> x.real.dtype
dtype('float64')
```

shape

Tuple of array dimensions.

Notes

May be used to “reshape” the array, as long as this would not require a change in the total number of elements

Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

size

Number of elements in the array.

Equivalent to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

strides

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array *a* is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

See Also:

`numpy.lib.stride_tricks.as_strided`

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array *x* will be `(20, 4)`.

Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
```

```

>>> offset/y.itemsize
17

>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813

```

tags

tag dictionary for this data object.

valueDescription

value unit description.

valueOffset

value offset [default: 0.0].

valueScale

value scale [default: 1.0].

valueUnit

value unit string.

9.10 point

class itom.point

point([type, [xyz, [intensity,][rgba,][normal,][curvature]]) -> creates new point used for class 'pointCloud'.

Parameters **type** : {int}

the desired type of this point (default: point.PointInvalid). Depending on the type, some of the following parameters must be given:

xyz : {seq}, all types besides PointInvalid

sequence with three floating point elements (x,y,z)

intensity : {float}, only PointXYZI or PointXYZINormal

is a floating point value for the intensity

rgba, {seq. of uint8, three or four values}, only PointXYZRGBA or PointXYZRGBNormal :

a uint8-sequence with either three or four values (r,g,b,a). If alpha value is not given, 255 is assumed

normal : {seq}, only PointXYZNormal, PointXYZINormal and PointXYZRGBNormal

is a sequence with three floating point elements (nx, ny, nz)

curvature : {float}, only PointXYZNormal, PointXYZINormal and PointXYZRGBNormal

is the curvature value for the normal (float)

name ()

PointInvalid = 0

PointXYZ = 1

PointXYZI = 2

PointXYZINormal = 16

PointXYZNormal = 8

PointXYZRGBA = 4

PointXYZRGBNormal = 32

curvature

gets or sets curvature value

Notes

{float} : ReadWrite

intensity

gets or sets intensity

Notes

{float} : ReadWrite

normal

gets or sets normal vector (nx,ny,nz)

Notes

{float-list} : ReadWrite

rgb

gets or sets rgb-values (r,g,b)

Notes

{uint8-list} : ReadWrite

rgba

gets or sets rgba-values (r,g,b,a)

Notes

{uint8-list} : ReadWrite

type

returns type-object for this point

xyz

gets or sets x,y,z-values (x,y,z)

Notes

{float-list} : ReadWrite

9.11 pointCloud

class `itom.pointCloud`

`pointCloud([type] | pointCloud [,indices] | width, height [,point] | point) -> creates new point cloud.`

Parameters `type` : point-type (e.g. `point.PointXYZ`, see Notes)

pointCloud : {pointCloud}

another pointCloud instance which is appended at this point cloud

indices : {sequence}, optional

only the indices of the given pointCloud will be appended to this point cloud

width : {int}

width of the new point cloud

height : {int}

height of the new point cloud (the point cloud is dense if height > 1)

point : {point}

single point instance. This point cloud is filled up with elements of this point. If width and height are given, every element is set to this point, else the point cloud only consists of this single point.

Notes

Possible types:

- `point.PointXYZ`
- `point.PointXYZI`
- `point.PointXYZRGBA`
- `point.PointXYZNormal`
- `point.PointXYZINormal`
- `point.PointXYZRGBNormal`

append (*point*) → appends point at the end of the point cloud.

Parameters `point` : {point???}, optional

Notes

The type of point must fit to the type of the point cloud. If the point cloud is invalid, its type is set to the type of the point.

clear () → clears the whole point cloud

dense

specifies if all the data in points is finite (true), or whether it might contain Inf/NaN values (false).

Notes

{bool} : ReadOnly ReadWrite

empty

returns whether this point cloud is empty, hence size == 1

Notes

{ } : ReadOnly ReadWrite

erase (*indices*) -> *erases the points in point clouds indicated by indices (single number of slice)*

Parameters **indices** : { }

Notes

This method is the same than command 'del pointCloudVariable[indices]'

fields

get available field names of point cloud

Notes

{ } : ReadOnly ReadWrite

static fromDisparity (*disparity* [, *intensity*] [, *deleteNaN*]) → creates a point cloud from a given disparity dataObject.

Parameters **disparity** : {MxN data object, float32}

The values of this dataObject represent the disparity values.

intensity : {MxN data object, float32}, optional

If given, an XYZI-point cloud is created whose intensity values are determined by this dataObject deleteNaN : {bool}, optional If true (default: false), all NaN-values in the disparity map will not be copied into the point cloud.

Returns **PointCloud.** :

static fromXYZ (*X*, *Y*, *Z* | *XYZ*) → creates a point cloud from three X,Y,Z data objects or from one 3xMxN data object

Parameters **X,Y,Z** : {MxN data objects}

Three 2D data objects with the same size.

XYZ : {3xMxN data object}

OR: 3xMxN data object, such that the first plane is X, the second is Y and the third is Z

deleteNaN : {bool}

default = false if True all NaN values are skipped, hence, the resulting point cloud is not dense (organized) any more

Returns **PointCloud.** :

static fromXYZI (*X*, *Y*, *Z*, *I* | *XYZ*, *I*) → creates a point cloud from four X,Y,Z,I data objects or from one 3xMxN data object and one intensity data object

Parameters **X,Y,Z,I** : {MxN data objects}

Four 2D data objects with the same size.

OR :

XYZ : {3xMxN data object}

3xMxN data object, such that the first plane is X, the second is Y and the third is Z I : {MxN data object} MxN data object with the same size than the single X,Y or Z planes

deleteNaN : {bool}

default = false if True all NaN values are skipped, hence, the resulting point cloud is not dense (organized) any more

Returns **PointCloud.** :

height

returns height of point cloud if organized as regular grid (organized == true), else 1 specifies the height of the point cloud dataset in the number of points. HEIGHT has two meanings:

- it can specify the height (total number of rows) of an organized point cloud dataset;
- it is set to 1 for unorganized datasets (thus used to check whether a dataset is organized or not).

Notes

{ } : ReadOnly ReadWrite

insert (*index, values*) → inserts a single point or a sequence of points before position given by index

Parameters **index** : { }

values : { }

Notes

doctodo

name ()

organized

returns whether this point cloud is organized as regular grid, hence height != 1

Notes

{ } : ReadOnly ReadWrite

size

returns number of points in point cloud

Notes

{ } : ReadOnly ReadWrite

toDataObject () → returns a PxN data object, where P is determined by the point type in the point cloud. N is the number of points.

Returns **PxN** : {float}

Notes

doctodo

type

returns point type of point cloud

Notes

{ } : ReadOnly ReadWrite

width

returns width of point cloud if organized as regular grid (organized == true), else equal to size specifies the width of the point cloud dataset in the number of points. WIDTH has two meanings:

- it can specify the total number of points in the cloud (equal with POINTS see below) for unorganized datasets;
- it can specify the width (total number of points in a row) of an organized point cloud dataset.

Notes

{ } : ReadOnly ReadWrite

9.12 polygonMesh

class itom.polygonMesh

`polygonMesh([mesh, polygons])` -> creates a polygon mesh.

This constructor either creates an empty polygon mesh, a shallow copy of another polygon mesh (mesh parameter only) or a deep copy of another polygon mesh where only the polygons, given by the list of indices in the parameter 'polygons', are taken. In this case, the containing cloud is reduced and no longer organized (height=1, dense=false)

Parameters **mesh** : {polygonMesh}, optional

another polygon mesh instance (shallow or deep copy depending on polygons-parameter)

polygons : {sequence or array-like}, optional

If given, polygons must be a sequence or one-dimensional array-like structure, where all values can be transformed into unsigned integer values. Polygons must contain a list of indices pointing to all polygon from the given mesh that should be copied to this new instance.

data()

prints content of polygon mesh

static fromCloudAndPolygons (*cloud, polygons*) → creates a polygon mesh from cloud and polygons.

Parameters **cloud** : {pointCloud}

the input point cloud

polygons : {array-like, MxN}

an array-like matrix with the indices of the polygons. The array contains M polygons and every row gives the indices of the vertices of the cloud belonging to the polygon.

getCloud (*pointType = point.PointInvalid*) → returns the point cloud of this polygon mesh converted to the desired type.

If the pointType is not given or point.PointInvalid, the type of the internal pointCloud is guessed with respect to available types.

Parameters **pointType** : {int, enum point.PointXXX}, optional

the point type value of the desired type, the point cloud should be converted too (default: point.PointInvalid)

getPolygons()

name()

nrOfPolygons

returns the number of polygons in this mesh

9.13 region

class `itom.region`

Bases: `builtins.object`

`region([x, y, w, h [,type=region.RECTANGLE]])` -> creates a rectangular or elliptical region.

This class is a wrapper for the class `QRegion` of Qt. It provides possibilities for creating pixel-based regions. Furthermore you can calculate new regions based on the intersection, union or subtraction of other regions. Based on the region it is possible to get a `uint8` masked `dataObject`, where every point within the entire region has the value 255 and all other values 0

Parameters `x` : {int}

x-coordinate of the reference corner of the region

`y` : {int}

y-coordinate of the reference corner of the region

`w` : {int}

width of the region

`h` : {int}

height of the region

type : {int}, optional

`region.RECTANGLE` creates a rectangular region (default). `region.ELLIPSE` creates an elliptical region, which is placed inside of the given boundaries.

Notes

It is also possible to create an empty instance of the region.

For an alphabetic index see:

- *genindex*

If you need help about other important **Python** modules, like **Numpy**, **Scipy**, **Matplotlib**, see the corresponding references in the internet.

MISCELLANEOUS

10.1 How to use the help

In general you can use this help as any other help.

A problem with function references is, it is hard to keep up to date with the manual. In addition the help can never have all informations about any possible filter- or hardware-plugin delivered from a third party. Therefore some tools for online help were implemented into python and itom.

With `help(method)` you will get an online help for the method or module. To get something similar for the plugin-system, the functions *filterHelp(...)*, *widgetHelp(...)* and *pluginHelp(...)* can be used. If you already have a plugin of type *actuator* or *dataIO* you can get an online-help for possible *parameters* via the member method *getParamListInfo()* and for the exec-functions use *getExecFuncInfo()*.

10.1.1 Rebuild the Help

If you think your help is not up to date and you are using the itom development environment you can rebuilt your help. Therefore you need the up-to-date-version of sphinx for python.

10.2 Units and Conventions

The Units and PlugIn-Parameters are defined according to the SI-Units. Parameters have to be expressed in their unified base unit. (E.g. 1 micron for an actuator setPos-command means 0.001 [mm])

10.2.1 General

- Indexing starts with 0
- The names of PlugIns starts with a capital letter

10.2.2 Physical Units

- Physical Base Units have to be in [mm; s; kg]
- Electrical Base Units have to be in [V; A; W]

DEMO SCRIPTS

There are several python demo scripts available which demonstrate the use of **itom**. All these files are in the directory **demo** The following list gives a short description of each demo.

11.1 itom Basics

- **demoDataObject.py**

Description: Here you can learn the basic function of the dataObject.

Keywords: creating a dataObject; plot; shallow copy; deep copy; meta data

- **demoToolBar.py**

Description: Creating your own toolbar and buttons.

Keywords: create a new class; add new functions; add button

11.2 Plugins

- **demoDummyGrabber.py**

Description: Usage of a camera plugin.

Keywords: dataIO; start device (camera); snapshot (getVal); live image

- **demoDummyMotor.py**

Description: Usage of a motor plugin.

Keywords: set position; get position

- **demoCMU1394.py**

Description: Firewire grabber for different cameras. PointGray Firefly, Sony SX 900, Sony XCD-X700...

Keywords: dataIO; start device (camera); Snapshot (getVal); Live image

11.3 Algorithm / Filter

- **demoOpenCVFilter.py**

Description: Median filtering of a randomly filled image.

- **demoNumpy.py**

Description: Short demonstration of some linear algebra functions provided by Numpy (numeric package of python).

- **demoScipy.py**

Description: Using scipy and matplotlib to calculate the cross-correlation between two images. Scipy is a python package that contains more scientific algorithms.

Keywords: scipy, matplotlib

- **demoSignalSmooth.py**

Description: Further example on how to use matplotlib (plotting package of python) in itom.

11.4 ui

This folder contains some examples on how to create customized user interfaces in itom (see *Creating advanced dialogs and windows*).

PYTHON TUTORIALS

12.1 determine lateral image shift and show images by using itom figure plots

A lateral image shift can be determined by cross correlation. To minimize the calculation time, the inverse fourier transformed product of the fourier transformed lateral shifted images, whereas one image is complex conjugated, is calculated

```
#-----  
#  
# determine the amount of a lateral image shift  
#  
#-----  
  
# import necessary modules  
#  
import numpy as np  
import scipy.misc  
  
#--- create test data (lateral shift an image of Lena) -----  
#  
  
# load an image of Lena  
#  
imageLena = scipy.misc.lena()  
  
# plot lena  
#  
plot(np.flipud(imageLena), 'itom2DQwtFigure')  
  
# amount of pixel shift in x- and y-direction  
#  
xPixelShift = 16  
yPixelShift = -7  
  
# determine the ROI size: relative (centered) size of original image (relativeSize=1: original si  
#  
row, col      = imageLena.shape  
relativeSize = np.floor( min( 1-abs(xPixelShift)/col, 1-abs(yPixelShift)/row ) * 10 ) /10  
  
x0 = int( (col - col*relativeSize)/2 )  
x1 = col-x0 + 1  
y0 = int( (row - row*relativeSize)/2 )
```

```
y1 = row-y0 + 1

# not shifted ROI
image1 = imageLena[y0:y1,x0:x1].copy()
plot(np.flipud(image1),'itom2DQwtFigure')

# shifted ROI
image2 = imageLena[y0+yPixelShift:y1+yPixelShift,x0+xPixelShift:x1+xPixelShift].copy()
plot(np.flipud(image2),'itom2DQwtFigure')

#
#-----

#--- determine the pixel shift -----
#

# discrete fast fourier transformation and complex conjugation of image 2
#
image1FFT = np.fft.fft2(image1)
image2FFT = np.conjugate( np.fft.fft2(image2) )

# inverse fourier transformation of product -> equal to cross correlation
#
imageCCor = np.real( np.fft.ifft2( (image1FFT*image2FFT) ) )

# Shift the zero-frequency component to the center of the spectrum
#
imageCCorShift = np.fft.fftshift(imageCCor)
plot(imageCCorShift,'itom2DQwtFigure')

# determine the distance of the maximum from the center
#
row, col = image1.shape

yShift, xShift = np.unravel_index( np.argmax(imageCCorShift), (row,col) )

yShift -= int(row/2)
xShift -= int(col/2)

print("shift of image1 in x-direction [pixel]: " + str(xShift))
print("shift of image1 in y-direction [pixel]: " + str(yShift))

#
#-----
```

First the necessary modules have to be imported:

```
# import necessary modules
#
import numpy as np
import scipy.misc
```

Test data is created by lateral shifting a region of interest (ROI) of the Lena image. First the image of Lena is loaded:

```
# load an image of Lena
#
imageLena = scipy.misc.lena()
```

The image of Lena is plotted by using the itom figure plot 'itom2DQwtFigure', which is optimized for 2D static images. Since the row index for images starts at the top of an image and not at the bottom like for matrixes, the image as to be flipped up side down before plotting.

```
# plot lena
#
plot(np.flipud(imageLena),'itom2DQwtFigure')
```

The ROI size is determined by the amount of lateral shift in x- and y-direction. One ROI is selected from the center of the original image. Another ROI with the same size is shifted from the center about the defined amount.

```
# amount of pixel shift in x- and y-direction
#
xPixelShift = 16
yPixelShift = -7

# determine the ROI size: relative (centered) size of original image (relativeSize=1: original si
#
row, col      = imageLena.shape
relativeSize = np.floor( min( 1-abs(xPixelShift)/col, 1-abs(yPixelShift)/row ) * 10 ) /10

x0 = int( (col - col*relativeSize)/2 )
x1 = col-x0 + 1
y0 = int( (row - row*relativeSize)/2 )
y1 = row-y0 + 1

# not shifted ROI
image1 = imageLena[y0:y1,x0:x1].copy()
plot(np.flipud(image1),'itom2DQwtFigure')

# shifted ROI
image2 = imageLena[y0+yPixelShift:y1+yPixelShift,x0+xPixelShift:x1+xPixelShift].copy()
plot(np.flipud(image2),'itom2DQwtFigure')
```

Now the lateral shift is determined by calculating the inverse fourier transformed of the product of the fourier transformed ROIs and evaluating the distance from the center of the position of its maximum.

```
# discrete fast fourier transformation and complex conjugation of image 2
#
image1FFT = np.fft.fft2(image1)
image2FFT = np.conjugate( np.fft.fft2(image2) )

# inverse fourier transformation of product -> equal to cross correlation
#
imageCCor = np.real( np.fft.ifft2( (image1FFT*image2FFT) ) )

# Shift the zero-frequency component to the center of the spectrum
#
imageCCorShift = np.fft.fftshift(imageCCor)
plot(imageCCorShift,'itom2DQwtFigure')

# determine the distance of the maximum from the center
#
row, col = image1.shape

yShift, xShift = np.unravel_index( np.argmax(imageCCorShift), (row,col) )

yShift -= int(row/2)
xShift -= int(col/2)
```

```
print("shift of image1 in x-direction [pixel]: " + str(xShift))
print("shift of image1 in y-direction [pixel]: " + str(yShift))
```

..toctree::

maxdepth 1

todo.rst

INDICES AND TABLES

- *genindex*
- *modindex*

PYTHON MODULE INDEX

i

itom, [228](#)