Using SmarAct Python SDKs

www.smaract.com

The information given in this document was carefully checked by our team and is constantly updated. Nevertheless, it is not possible to fully exclude the presence of errors. In order to always get the latest information, please contact our technical sales team.

SmarAct GmbH, Schuette-Lanz-Strasse 9, D-26135 Oldenburg
Phone: +49 (0) 441 - 800879-0, Telefax: +49 (0) 441 – 800879-21
Internet: www.smaract.com, E-Mail: info@smaract.com

This document describes the installation and use of a SmarAct Python SDK (software development kit). All Python SDKs for SmarAct products are very similar in structure. This document therefore describes their use in a product-independent manner.

A **SmarAct Python SDK** typically contains

- a **Python package**, which contains the **Python API** for the respective SmarAct product,
- **programming examples**.

The Python SDK is installed by the software installer which is part of the SmarAct product software.

> In this document, the Linux path separator / is used. Please replace it by \ *on* Windows.

INSTALLATION

The software installer stores a pip-compatible Python package ZIP file in the software installation folder under *SDK/Python/packages*.

The Windows installer also allows to install the Python package directly into your local Python installation using *pip*, if you select this option in the installer. This may not work, if you have Python installed only for your user account, not for all users. (In that case the installer will report that no Python installation could be found).

Requirements

In order to install and use a SmarAct Python package, some conditions must be fulfilled:

- Python 3.4 or later must be installed. (For all steps described in this document it is assumed that Python 3.4 or later is used)
- The Python package installer *pip* must be installed
- The pip tool needs network access to the PyPi package repository for downloading dependencies of the SmarAct Python packages during installation. If it is not possible for pip to access the PyPi repository, ensure that the Python Packages *cffi* and *setuptools* are installed before starting the installation. (Otherwise the installer might display error messages).

Installing a SmarAct Python package manually

In a command shell (Windows: cmd.exe or the PowerShell, Linux: any terminal) locate the Python package ZIP file in the subfolder *SDK/Python/packages* in the product software installation folder. From there you can install the package with pip. (replace <*productname*> and <*version*> by the product name and version of the file in the installation folder):

```
pip install smaract.<productname>-<version>.zip

# or (Linux)
python3 -m pip install smaract.<productname>-<version>.zip

# or (Windows)
python -m pip install smaract.<productname>-<version.zip
```

This should be sufficient to add the module to your local Python installation. If pip is not available, you can unzip the archive and add the directory to the *PYTHONPATH* environment variable.

Before you start using the Python API make sure that you have

- (Windows): the corresponding SmarAct DLLs in the search path for executables so that your Python interpreter can find them.
- (Linux): the corresponding shared objects installed in one of the Linux distributions standard search paths or adjusted the *LD_LIBRARY_PATH* variable.

The DLLs or shared objects are usually installed by the SmarAct software installer. However, if you want to install the Python package on a different computer (where the software installer was not executed) you need to install the libraries from the *SDK/Redistributable* folder manually.

The success of the installation may be verified by importing the module in the Python shell. If no error is displayed, the installation should have been successful. For example, for the *smaract.smarpod* package it should look like this:

```
>>> import smaract.smarpod
>>>
```

UNINSTALLING

The uninstaller does not remove the installed python package from the local Python installation. It has to be removed manually, for example, for "smaract.apiname":

```
pip uninstall smaract.apiname
```

FINDING API DOCUMENTATION

Docstrings

The Python modules have docstrings that can be displayed with the *help()* function. For example, for "smaract.apiname":

```
>>> import smaract.apiname
>>> help(smaract.apiname)
```

and for a single functions:

```
>>> help(smaract.apiname.Function)
```

Programming Examples

The Python SDK includes programming examples which demonstrate using the API. You find the example files in the *SDK/Python/examples* subfolder in the product software installation folder.

Programmer's Guides

Most *Programmer's Guide* PDFs included in SmarAct product software bundles focus on the C API. Since the relationship between a Python function or constant and the C counterpart is simple, you can search for Python function and constant names in the guide. See also *Differences to the C API* below.

DIFFERENCES TO THE C API

This section describes the differences between the C and the Python API and is intended to help you reading the *Programmer's Guide* document (or the C header files), which focuses on the C API, from the Python programmer's point of view.

Naming of Modules, Functions and Constants

SmarAct C API function and constant names are usually prefixed (because C knows no namespaces, classes or modules). The C name prefix is removed in Python. In C, functions may be named

```
SA_CTL_FindDevices()
```

where *SA_CTL_* is a name prefix that is not part of the corresponding Python function name. The documentation can be found under its C name *SA_CTL_FindDevices()* in the *Programmer's Guide*.

Error Handling

In SmarAct C APIs most functions return an error code that indicates the success or failure of the called function. In the Python API errors are not returned as function results but raised as exceptions if the function execution fails. The error code is stored in the exception object.

Since there is no error code returned, some Python API functions don't return anything at all.
A C API function that is declared:

```
ErrorCode SA_APINAME_Function1(Type1 inParameter1, Type2 inParameter2);
```

and that would be called in C or C++ as follows:

```
ErrorCode err;
err = SA_APINAME_Function1(p1,p2);
```

has a different signature in Python. The error code is not returned and cannot be assigned to a variable. The Python program should catch the exception that may be raised by *Function1()* somewhere:

```
try:
    # ...
    smaract.apiname.Function1(inParameter1,inParameter2)
    # ...
except smaract.apiname.Error as err:
    # handle exception
```

In order to free resources correctly, like closing a connection to a SmarAct controller, you should write a *finally* clause after a *try/catch* block in which the connection is closed. (Please refer to the Python programming example files in the respective product Python SDK for a demonstration).

Output Parameters

In SmarAct C APIs, output parameters are implemented as pointers. Output parameters in Python are actual return values. For example, a C function that is declared:

```
ErrorCode SA_APINAME_Function2(Type1 inParameter, Type2 *outParameter);
```

returns *outParameter* as a function return value in Python which can be assigned to a variable:

```
out = smaract.apiname.Function2(inParameter)
```

Multiple output parameters of a C function

```
ErrorCode SA_APINAME_Function3(Type1 inParameter, Type2 *outParam1, Type3 *outParam2);
```

are multiple return values of the Python function (in the same order as the C parameters):

```
out1, out2 = smaract.apiname.Function3(inParameter)
```

Buffer Parameters

There are special input/output parameter combinations in the C APIs which are handled differently in the Python API. Some functions return lists of values a buffer. On the C side this is implemented with a combination of a pointer and a size parameter. The size parameter is an input and an output parameter: the caller specifies the capacity of the buffer he provides in the call. On return, the function writes the number of elements into the size parameter that it has stored in the buffer or the required size, if the provided buffer was too small. The corresponding Python function has just a size parameter in which the caller can specify the array size to be allocated before executing the function. The buffer data is returned as a Python array. C function:

```
ErrorCode SA_APINAME_FindSystems(const char *options,
                                 char *outBuffer,
                                 unsigned int *ioBufferSize);
```

Python function call with a buffer size of 1024 elements:

```
outArray = smaract.apiname.FindSystems("", 1024)
```

Default Parameter Values

Some parameters have default values in Python. The default values are described in the function docstrings. For example:

```
>>> help(smaract.apiname.FindSystems)
Help on function FindSystems in module smaract.apiname:

FindSystems(options='', ioBufferSize=256)
    Searches for controllers.

    Parameters:
    - options = "": Reserved for now.
    - ioBufferSize = 256: Pass the size of the provided buffer in here.

    Return value(s):
    - outBuffer: The buffer for writing the result into. the found items
    are separated by a newline character.
```

Parameters for which default values are defined can be omitted when calling the Python function (which is not possible in C). This means that the function from the above example may be called:

```
# with empty options and default buffer size (=256)
outArray = smaract.apiname.FindSystems()

# with a user-defined value for options and default buffer size
outArray = smaract.apiname.FindSystems("x=3")

# with all arguments specified by the caller
outArray = smaract.apiname.FindSystems("", 1024)
```

Constants Grouping

In SmarAct C API header files constants are defined using C preprocessor macros, e.g.:

```
#define SA_APINAME_VERSION_MAJOR    1
#define SA_APINAME_VERSION_MINOR    4
#define SA_APINAME_VERSION_UPDATE   2
```

In Python the constants are grouped in enumerations:

```
class ApiVersion(enum.IntEnum):
    MAJOR  = 0x1
    MINOR  = 0x4
    UPDATE = 0x2
```

and can be accessed with

```
ApiVersion.MAJOR
ApiVersion.MINOR
ApiVersion.UPDATE
```

Constants in the *Global* group are also defined at the module level. For example, a constant *HOLDTIME_INFINITE* from the *Global* group can be accessed in two ways:

```
smaract.apiname.Global.HOLDTIME_INFINITE
smaract.apiname.HOLDTIME_INFINITE
```

Boolean Parameters and TRUE and FALSE Values

Some SmarAct C APIs define *TRUE* and *FALSE* constants, typically as preprocessor macros:

```
#define SA_APINAME_FALSE 0
#define SA_APINAME_TRUE  1
```

The Python API provides these constants in the *Global* constants group:

```
smaract.apiname.TRUE
smaract.apiname.FALSE
# or...
smaract.apiname.Global.TRUE
smaract.apiname.Global.FALSE
```

Instead of using these constants, you can pass the Python literals *True* and *False* in boolean parameters, which is much simpler.