TM870

APROL Python Programming

PERFECTION IN AUTOMATION
www.br-automation.com

Requirements	
Training modules	TM830 - APROL Project Engineering
Software	APROL SuSE LINUX
Hardware	1 control computer, 1 controller

Table of Contents

1 Introduction

This training module deals with Python, a free programming language which is continuously winning in popularity and can be deployed in a flexible manner. Python is a scripting language which is easy to learn and allows maximum usage based upon a few basic rules. Furthermore, Python is available on all common operating systems and platforms and offers superb interaction in the corresponding environment.

This is also the reason why APROL took the decision for this additional type of programming, in order to be able to realize special functionalities which are not contained in the standard scope of the system. The Python expansion allows customer-specific visualization requirements (operation) to be realized according to the corresponding operation philosophy.

This programming language is the perfect tool for RAD development (Rapid Application Development).

Figure 1: Python expansion In APROL

1.1 Objectives

After this training unit and studying the documents, you will be in the position to write programs in the Python programing language. And you will have also learnt about the Python API in APROL. You have got to know the different areas of usage for Python in APROL. You have an overview of which hooks (= events) there are and can connect these hooks with the functions you have written.

You are now able to extend your project by using Python. Of course, the standard APROL functionalities are used where possible. You are able to use python for requirements which go beyond these demands.

Figure 2: Objectives

2 Python basics

2.1 Python history

Python was developed by the Dutchman Guido van Rossum at the Centrum voor Wiskunde en Informatica (CWI) in Amsterdam, in the 90s.

The name Python comes from van Rossum's passion for the British comedy group Monty Python.

The noncommercial Python Foundation exists since 2001. It owns the Python code rights. It also hosts the Python website on www.python.org. The web page is also the source for downloading Python (Linux, Windows, etc.) and for the official Python documentation.

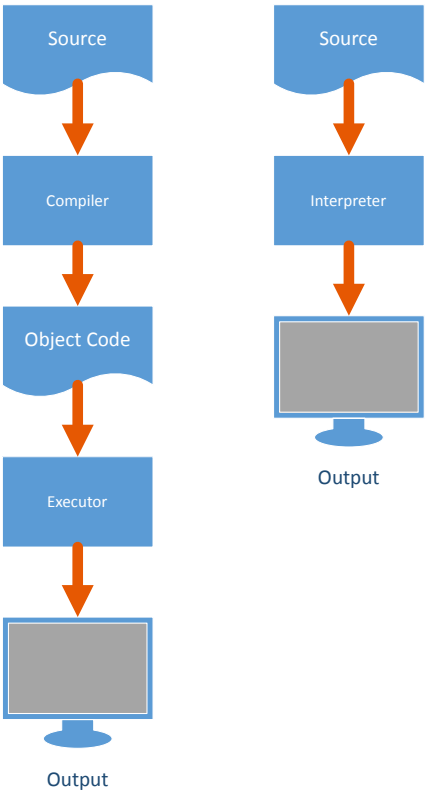
2.2 Python references

- www.python.org
- APROL Product Documentation - X3 - Python Documentation
- APROL Product Documentation - X11 - Python Beginners Course
- German book: Python das umfassende Handbuch, Peter Kaiser, Johannes Ernesti, Galileo Computing publishing
- English book: Learning Python, Mark Lutz, O'Reilly publishing

2.3 Python properties

Python is a universal, high-level, object-oriented, interpretation scripting language.

Explanations of the key words Interpret and Compile:



- A program (compiler) translates the source code during the compilation process into machine code, which is the only dialect which a computer can understand. An executable file is the result of the compilation.
- The interpretation process is when a program is translated and executed directly, line-by-line.

Figure 3: Compiler vs. Interpreter

Because of the declared benefits, Python has become very popular. Here are some of these benefits:

- Python is very easy to learn and powerful at the same time.
- Python code can be read very well.
- Python code can be transferred. The code can be used in UNIX, Widows, and Mac.
- Python is heavily object-oriented.
- Python has a dynamic data type management (variable declarations are not necessary).
- Python contains an automatic memory management (Garbage Collection). The allocation and release of memory must not be carried out manually, but is done automatically.
- Python is gratis. It is available to everyone without limitations, because it is covered by the GPL (General Public License).
- Python offers an extensive collection of libraries (os, sys, math, time, random, httplib, ftplib, re, ...).
- Python can be extended. It can be extended with C modules at any time, for example.
- Python is well documented. There are many books concerning the topic and tutorials in the internet. There are also many web sites where you can get help from the Linux community.

2.4 Python application

This script language is very suitable for resolving the following programming jobs:

- It is very suitable for creating platform-independent system utility tools, because of the many interfaces to the operating system.
- Python can also be used to program graphical user interfaces, because of its interfaces to different tool kits (PyQt , Tkinter, etc.).
- Numerous libraries make it easy to solve numerical or mathematical problems in Python.
- Python has a powerful API for connecting databases
- Python is also used for the scripting in internet pages
- Python is deployed on embedded systems such as RaspberryPi for training purposes. (The "Pi" in RaspberryPi originally stands for Python Interpreter!)

Named references for Python usage are, amongst others, Google, Yahoo, and the NASA.

Python is similar to other interpreter languages in so far that it is slower than the compiler languages such as C. Drivers and such are therefore mostly developed in C or assembly language.

2.5 Python interactive mode

Python programs can be created in various ways. A very effective way of programming in Python is the usage of the interactive mode.

The interactive mode is started by entering 'python' in a Linux console.
The interactive mode is terminated by using the 'Ctrl' + 'd' key combination or by entering the command 'exit()'.

You are greeted by the prompt ('>>>') after having started the interactive mode. You can now directly enter Python commands without having had to create a program file beforehand. The interactive mode is a very good way of trying something out quickly.

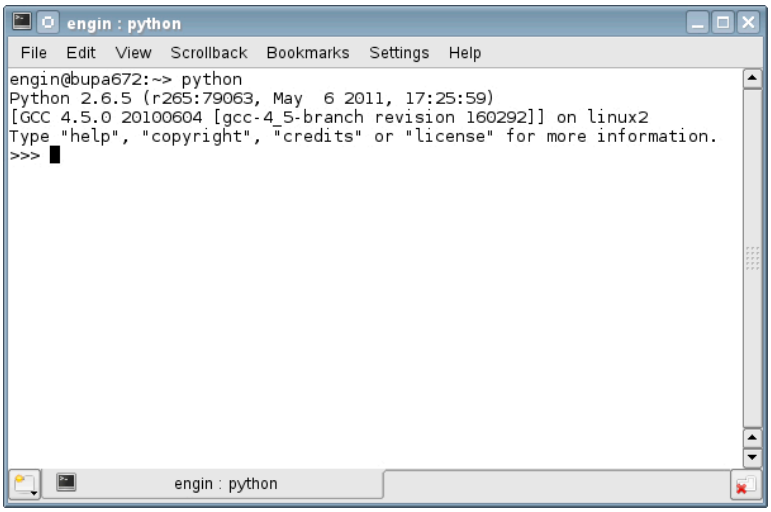


Figure 4: Python interactive mode

The directives are entered one line at a time. The input of one line is terminated with 'Enter'. The directive is executed immediately by the Python interpreter after pressing 'Enter'. If the directive which was entered provokes an output, it will be shown immediately in interactive mode.

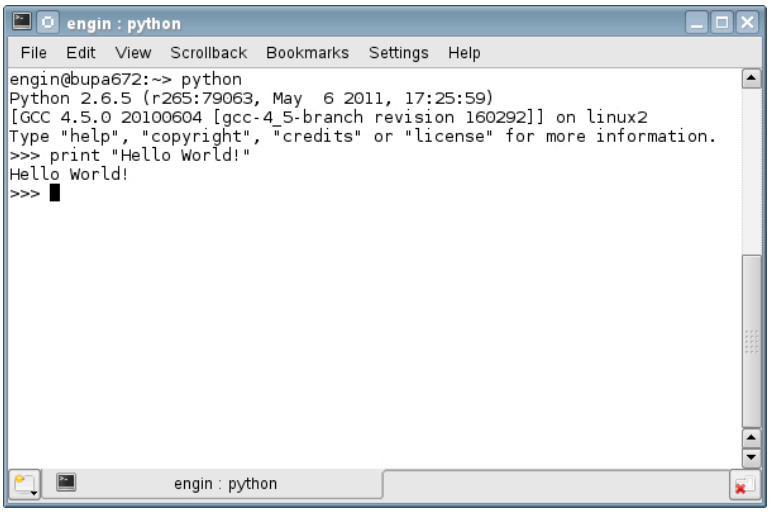
The Python interactive mode has a history function. The entries which were made last can be shown again by pressing the 'Arrow up' and 'Arrow down' keys. The entries can then be edited or sent once again.

It is also possible to enter several directives at once in the interactive mode. It is also possible to create loops, case differentiations, or functions in this case. If a multiple line directive is recognized in the input, the prompt will changes to ('...'). This allows you to see that a directive with several lines is currently being entered.

The drawback of the interactive mode is that directives which have been entered are executed immediately and are not available as a source file. If no changes in the code are necessary, all of the directives must be typed out once again.

Hello World

A Hello World program will serve as the first programming example. We use the built-in 'print' function to output a character string. This appears as follows in the interactive mode:



Task: Use of the interactive Python console

1) Open the interactive python console.
2) Search for the version of the Python interpreter which is installed.
3) Create a 'Hello World' program in the interactive console.

2.6 Python Source File

In contrast to the use of the interactive mode, there is the variation with the source file. The source code is then written in one or more source files.

Editor

A editor is required to edit the source files. The editor should at least be able to carry out **syntax coloring** for Python. The **'kwrite'** editor which is in the Linux system can be used, for example. There would also be more IDEs (**I**ntergrated **D**evelopment **E**nvironment) which contain many more features (automatic code completion, static code analysis, debugging, etc.).

File name extension

The Python source files must be saved with the **'.py'** extension. A valid name would then be **'myTest.py'**. A proper file name extension is required for the case that this Python source file is integrated in another source file.

Path to the Python interpreter (Shebang)

The system path for the Python interpreter must be specified, so that the source file can be executed directly at a later stage. This is defined with the so-called **"Shebang"** (also known as the Magic Line). The line in the source file appears as follows:

```
#!/usr/bin/python
```

This line should be at the beginning of the source code in the file (first line).

A **"Hash" ("#")** normally signalizes the beginning of a comment in Python. The **exclamation mark ("!")** which was added at the beginning signalizes that this is a Shebang line, i.e. the specification of the Python interpreter. The path where the Python interpreter is stored follows after the exclamation mark.

If you do not know in which path the Python interpreter is installed on your system, there are ways of describing or finding out the correct path.

A possible way of specifying the interpreter is not through where it was installed, but an environment variable. It does not depend on the installation and just requires that the **'PATH'** variable was extended respectively during the Python installation, so that the Python interpreter can be found.

```
#!/usr/bin/env python
```

Alternatively, there is also a **'whereis'** Linux shell command which shows where a certain program is stored.

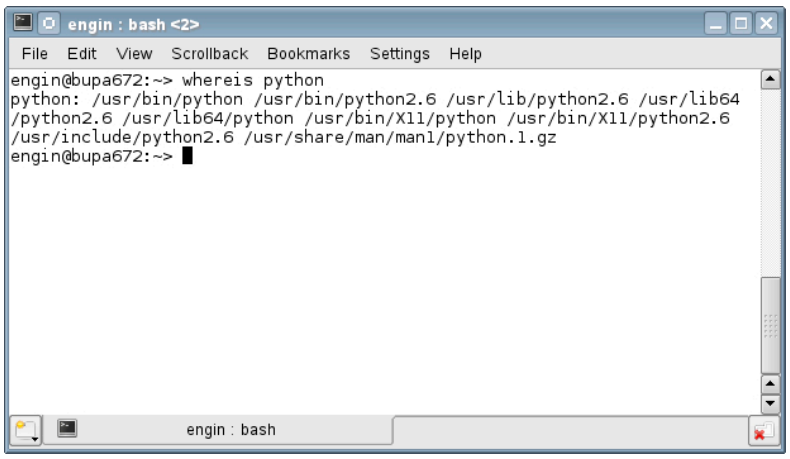


Figure 5: whereis python

The correct path must be picked from the output and inserted in the Python source file as a Shebang line.

Encoding

Encoding is understood as in which character coding the source file was saved. This definition is compulsory, because of different global character codes (latin-1, Latin-2, UTF8, etc.).

This is important for showing special characters in strings properly. Many editors suggest the insertion of an encoding entry while the source file is being saved for the very first time. This entry appears as follows for a file which was saved with a Unicode encoding (UTF-8).

```
# -*- coding: utf-8 -*-
```

The encoding should also be (like the interpreter specification) in the first lines of the source file (the second line is recommended).

The standard encoding for your system can be read in a Linux shell with the 'locale' command.

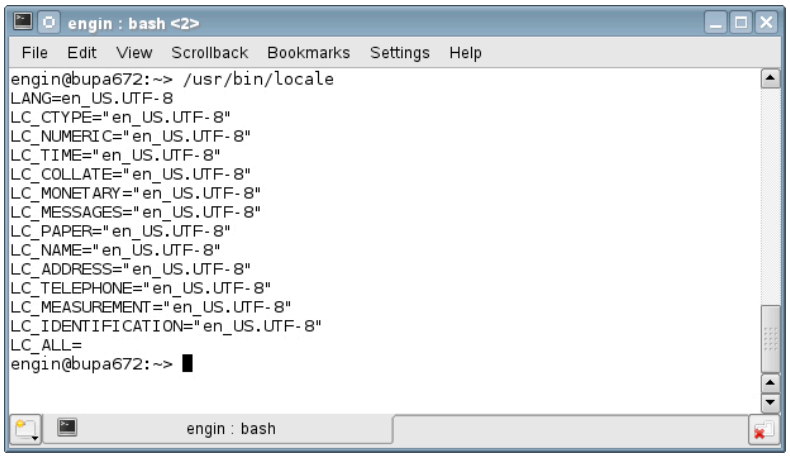


Figure 6: Output of the standard encoding

Run-capable (Executable-Flag)

The **Executable flag** must be set for a program to be executed in Linux. This can be done with the **'chmod'** command. There are different notations for chmod. For example, the command **"chmod a+x <file name>.py"** can be entered in order to grant all Linux users and all groups execution rights for a certain file.

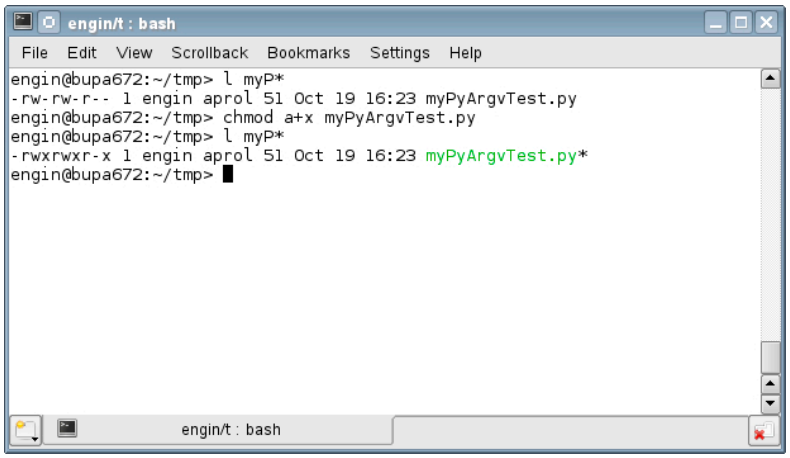


Figure 7: Make file executable

If the file can be executed, it should have a green colored font after listing the directory content with the 'ls-la' command. Details about this can be found in **'TM890 Linux Basics Training manual'** and many other Linux manuals.

Execute file

There are two basic methods to execute a Python program.

The Python source file does not have to be made executable if the first method is used (using the python command previously). The Shebang line can also be omitted, i.e. the specification of the interpreter in the file. They can be omitted from the file, because of the explicit specification of the Python interpreter during the execution of the file name. The syntax for this type of execution is **"python myFilename.py"**.

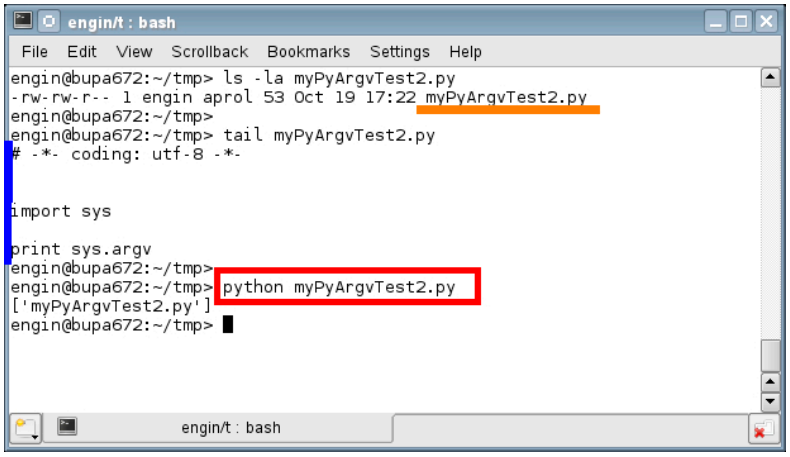


Figure 8: Execution with explicit specification of the interpreter

For the second method, the file must have be made executable beforehand ("chmod a+x myFile-Name.py"). Apart from that, the Shebang line must be contained in the file. It is then possible to run execute the file directly. The syntax for executing in the Linux shell is, for example, **"./myFilename.py"**.

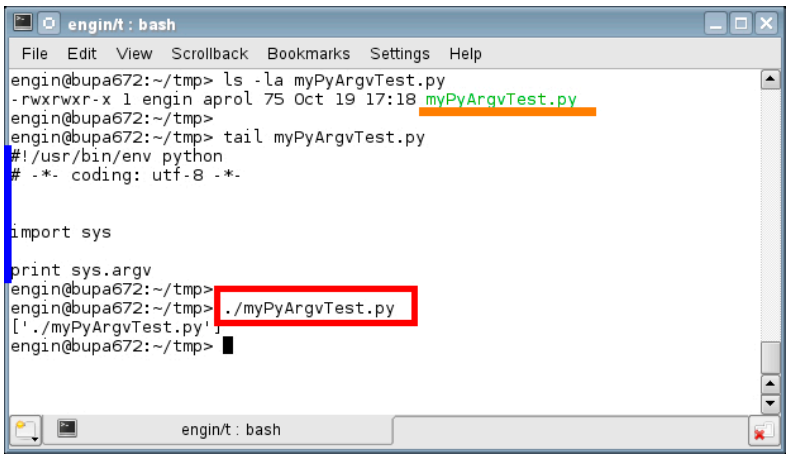


Figure 9: Direct execution of the program

Task: Use of a source file

1) Create a source file with the help of the 'kwrite' editor.
2) Save the file with the file extension '.py'.
3) Enter the Shebang line in the first line of the source file.
4) Enter the encoding in the second line of the source file.
5) Insert the code of the 'Hello word' program.
6) Make the file executable.
7) Execute the file.

2.7 Python structure

Indentation

Each programming language requires a certain syntax to specify related code blocks. Curly brackets {} are used for this in C. The curly bracket opens up at the beginning of an if statement. The curly bracket closes at the end of an if statement. This defines the beginning and end of the logical if block.

```
/*IF-block in C-Language*/

if (var > 100)
{         /* curly braces indicate begin of logical block*/
    cnt++;
}         /* curly braces indicate end of logical block*/
```

No brackets are used for this structure in python. Instead, **logically related blocks** are marked **by the same indentation depth**. **Spaces of tabs** can be used for indentation. You should decide on either tabs or spaces in order to avoid unnecessary errors.

The Python interpreter recalculates all tabs into 8 spaces.
This is also the reason why some Python editors use a tab width of 8 spaces. Most editors use a tab width of 4 spaces.

A **logical block** is normally introduced in python by a **colon (':')**. All of the subsequent commands are shifted one level to the right. As soon as a command which does not belong to that logical block comes about, it should be placed back one level to the left.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#main indentation level
print "Begin of program"
var=1000
cnt=0

#main indentation level
if (var > 100):         #colon indicates begin of a new logical block.
    #Indentaion level of if-block
    cnt = cnt +1
    print cnt

print "End of program"
#main indentation level
```

All whitespace characters (spaces, tabs) which are at the beginning of a code line are used by the Python interpreter for code analysis. The interpreter will create an exception if there is a whitespace character at the beginning of a line and the interpreter cannot recognize a new logical block. The following code would therefore lead to an **'Indentation error'**:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


print "Test1"
 print "Test2"    #Error: Indentation used but no new block
print "Test3"
```

The code of different software developers appears very similar and not different in style, because the indentation is a part of the language. This makes it very much easier to understand a program.

Logical and physical lines

A logical line is what Python sees as a command.

A physical line is what can be seen in a line of code.

A physical line should normally only contain one logical line. There are also examples where this is not the case. An example of one physical line per logical line is as follows:

```
a = 3
print a
```

Several logical lines can also be inserted into one physical line. A **semicolon (";")** is set between the logical commands. We do not recommend using this syntax, because the readability of the program will suffer enormously.

```
a = 3; print a
```

It is also possible to split the content of a logical line to several physical lines. A **backslash ("\")** is used for this.

```
#example for strings

mystring = "this is a \
long string. \
The string is \
written in 4 lines."



#example for code
a =\
3
```

2.8 Python code comments

One-line comment

It is of course possible and encouraged to make comments in your python code. A **hash ("#")** at the beginning of a line is used for this. All characters **between the hash ("#") and the end of the line** are considered comments.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

#this is a comment
print "Test"
print "Hello world!"    #comment next to code
```

Multi-line comment

If the comment goes over several lines, the **hash ("#") can be set at the beginning of each line**.

It is more effective to use a so-called **multi-line string** for comments over several lines. A multi-line string is introduced by **three Quotation marks (""" or ''')** and also closed by three quotation marks. Everything that is between the quotation marks is considered to be a comment. It normally does not matter if either single or double quotation marks are used. But the same sort of quotation mark must be used at the beginning and end.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
print "Begin of program"
"""Begin of
multline comment.
Next line of comment.
And another line of comment.
End of multiline comment."""
print "End of program"
```

2.9 Functions integrated in Python

Python has a series of so-called **'integrated functions' (built-in functions)**. They can be used without having to import a **library (module)**. The following shows some important **'built-in functions'**.

You obtain an overview of all integrated functions by entering the following command in the interactive mode:

```
>>> dir(__builtins__)
```

Standard output of - print

The **"print"** command is actually a key word and not a built-in function. In any case, print is a very important command in Python. The **"print" command** outputs a text to the monitor. Strictly speaking, the text is output on the standard output.

The syntax for that is: Keyword "print" followed by a text which is enclosed in single, double or triple quotation marks. It is also possible to output variable values and text mixed with variable values.

```
>>> print "some text"
some text
>>>
>>> a=10
>>> print a
10
>>> print "Some text " + str(a)
Some text 10
```

Several texts may also be concatenated. The syntax demands that commas are placed between the individual pieces of text.

```
>>> print "valueA:", 100, "valueB:", 200
valueA: 100 valueB: 200
```

A print command normally creates a line feed at the end if its output. The line feed can be suppressed by adding a comma to the end of the print command.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

```
print "some",
print "text"
```

```
some text
```

Input on the standard input - raw_input

The **"raw_input()"** function reads the user's input from the standard input. The input is interpreted as a string. If a number should be read, it must be converted explicitly from a string to a number.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

inputVariable = raw_input("Please enter a number:")
print "type before conversion is: ", type(inputVariable)

inputNumber=int(inputVariable)
print "type after conversion is: ",  type(inputNumber)
```

```
>>> Please enter a number:
300
type before conversion is:  <type 'str'>
type after conversion is:  <type 'int'>
```

Type conversion - int, float, str, etc.

The following functions perform explicit type conversion:

Function	Description
str()	Conversion to STRING
int()	Conversion to whole number (integer)
long()	Conversion to long whole number (long integer)
float()	Conversion to a floating point number
char()	Conversion from number to letter
ord()	Conversion from letter to number

2.10 Python control structures

Control structures are necessary to **control the course of a program**.

There is a differentiation between **loops (for and while)** and control structures for **conditional execution (if, elif, else)**.

Apart from that, there are also commands to **break** loop iteration (repetitions) and **continue** with the next iteration.

2.10.1 Python IF Statement

The **if statement** differentiates between different cases. A check is made to see if a logical condition is true or false. Depending on the result, the **if branch (logical condition is true)** is executed or the **else branch (logical condition is false)**. In this case, only the instructions in the corresponding branch are carried out.

The syntax is composed of the key word **"if"** a **logical condition** and a **colon (":")** in the head of the statement. The statement body follows and contains the instructions to be executed if the condition is met. This code block must be **indented by one level** (indentation)!

The key word **"elif"** can come after the key word **"if"** on the **same indentation level** . The **"elif"** also contains a **logical condition** and is also concluded with a **colon (":")**. The statement body follows and contains the instructions to be executed if the condition is met. This code block must be **indented by one level** (indentation)!

The key word **"else"** can come after the key word **"if"** on the **same indentation level** . The **"else"** is also concluded with a **colon (":")**. The statement body follows and contains the instructions to be executed if the condition is not met. This code block must be **indented by one level** (indentation)!

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

var=6

if(var < 5):
    print "less than 5"
elif(var < 7):
    print "less than 7, but greater-equal 4"
else:
    print "greater-equal 7"
```

The Python language does not contain a **"switch case"** construction as in the language C. One must work with **"if-elif-else"** constructions in Python instead.

2.10.2 Python WHILE loop

The **while statement** executes code blocks **repeatedly** as long as a **logical condition is met**.

The syntax in the loop head is the key word **"while"** followed by the **logical condition**. The loop head is also concluded with a **colon (":")**. The loop body follows one **indentation level deeper**. All instructions in the loop body are executed until the condition in the loop head is no longer met.

An **else branch** can come after the key word **"while"** on the **same indentation level** . The **else branch** will be executed as soon as the **while condition** is **not met** for the first time.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

var=5

while var > 0:
    print var
    var = var -1
else:
    print "while condition is False"
```

The code leads to the following output:

```
>>>
5
4
3
2
1
while condition is False
```

2.10.3 Python FOR loop

The **for loop** iterates (repeats) all elements of an object's sequence.

The basis of an iteration **(=running through step-by-step)** is a sequence (collection) of objects. This may be a list of values, for example, or any collection of objects. It is also possible to run through a string, because it is composed of single letters. In one cycle, the loop body (instruction block) is processed once for each element in the sequence.

The syntax for the statement head is composed of the key word **"for",** followed by a **loop variable,** followed by the key word **"in",** followed by a sequence, and closed with a **colon (":")**. The instruction body follows one indentation level deeper. The instruction body, the loop variable contains the element of the sequence which is being iterated.

The key word **"else"** can be used after the key word **"for"** on the **same indentation level** . The **else block** must then be **indented by one level**. The 'else' branch is then executed after looping over all of the elements in the sequence. If the loop is terminated prematurely with the **break statement,** the **else branch will not be executed**.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

myList=(1,4,7,8,6)

for loopvar in myList:
    print loopvar
else:
    print "finished"
```

A list of values is defined in the example. An iteration is carried out, step-by-step over all values.

In the first run, the **"loopvar"** variable is assigned the first value of the sequence; in the second run the second value of the sequence, and so on. Finally, the else branch is also executed once.

Integrated range() function

The built-in function **range()** is used very often in combination with the **for loop**. The **range(start,stop,step)** function delivers a list of values as a result. The "start" and "step" parameters are optional when calling the function. The first parameter specifies the start value, the second value the final value, and the third parameter the step value (if it is specified).

```
>>> range(10,20)
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> range(10,20,2)
[10, 12, 14, 16, 18]
>>> range(20)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Please note that the **start value** is always contained in the **result, but not the final value**.

A **for loop** which uses **range()** may loop something like this:

```
for x in range(10,20):
    print x
```

The result then looks like this:

```
10
11
12
13
14
15
16
17
18
19
```

A for loop with enumerate()

enumerate() is also an integrated function. It delivers a tuple (data record) which consists of a **consecutive value** and the **actual element** for each element in a sequence. By using **"enumerate"**, a **for loop** can be built in a very similar manner to that in C.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

myList=["a","x","m"]

for count,x in enumerate(myList):
    print count, x
```

The result of this code appears as follows:

```
>>>
0 a
1 x
2 m
```

2.10.4 Python break statement

The **break statement** is used to completely terminate a **while** or **for loop** . The code which follows the loop is then simply executed.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

var=5

while var > 0:
    print var

    if(var < 3):
        break

    var = var -1
else:
    print "while condition is False"
```

This code produces the following output:

```
>>>
5
4
3
2
```

This example will terminate the loop if the value of variable "var" is smaller than 3. Please note that the else branch will not then be executed!

2.10.5 Python continue statement

The **continue statement terminates the current loop cycle**. The loop continues with the next run.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

myString="abcdefghijklmnopqrstuvwxyz"

for letter in myString:
    if letter == "a":
        continue
    if letter == "e":
        continue
    if letter == "i":
        continue
    if letter == "o":
        continue
    if letter == "u":
        continue

    if letter == "z":
        break

    print letter
```

The program should output all of a string's letters which are not a vowel. If a vowel is found, the **"continue"** jumps to the next iteration. If the letter "z" is encountered, the loop will be terminated completely.

The code output appears as follows:

```
>>>
b
c
d
f
g
h
j
k
l
m
n
p
q
r
s
t
v
w
x
y
```

Using the Python control structures

1) Create a source file with the kwrite editor.
2) Insert the Shebang line and the encoding line.
3) Create Python code which uses the "print" and "raw_input" built-in functions.
4) Create Python code which uses control structures for conditional execution ("if", "elif", "else").
5) Create Python code which uses control structures for loops ("while", "for").
6) Also use the key words "break" and "continue" in your code.

2.11 Python operators

Operators have the same meaning as the **"calculation characters"** which can be used on different objects. Most of the instructions in programs contain **"expressions"**. These expressions are composed of operators (calculation characters) and operands (objects). Python supplies a series of operators.

- Arithmetic operators (+,-,*,/,...)
- Comparison operators (==, !=, <, >, ...)
- Assignment operators (=, +=, ...)
- Bit operators (&,|, ^,~)
- Membership operators (in, not in)
- Identity operators (is, is not)

It is important to remember that there are operators with higher and lower priorities. Multiplication comes before addition, etc.

Arithmetic operators

Here is an excerpt from the arithmetic operators.

Operator	Function
+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Floor division (places after the comma are cut off)
**	Exponents
%	Modulo (the remainder of whole number division)

```
>>> 2**8
256
>>> 10/3
3
>>> 10%3
1
```

Comparison operators

A simple check is performed with comparison operators. These are instructions and conditions which can either be **true** or **false (0 or 1)**.

An excerpt from the comparison operators:

Operator	Function
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!= or <>	Not equal to

```
>>> a=10
>>> b=20
>>> a < b
True
>>> b < a
False
>>> a == b
False
>>> a <= b
```

```
True
>>> b >= a
True
```

Assignment operators

The most important assignment operator is definitely the **"=" operator**. A value assignment is carried out with it.

Bitwise operators

This operator is required for **bit connections**:

Operator	Function
&	bit-by-bit AND
|	bit-by-bit OR
^	bit-by-bit exclusive OR
~	Complement
<<	bit shift to the left
>>	bit shift to the right

Logic operators

Several **expressions can be linked logically** and check for truth by using logical operators.

The following logical operators are known to Python:

Operator	Function
not x	Inversion 1 if x=0, otherwise 0
x or y	OR operation 1 if x=1 or y=1
x and y	AND operation 1 if x=1 and y=1

Membership operators

These operators check if a **certain value** is in a sequence.

Operator	Function
in	checks if a value is contained in a sequence
not in	checks if a value is not contained in a sequence

```
>>> myList=["a", "n", "u", "x"]
>>> "n" in myList
True
>>> "b" in myList
False
>>> "y" not in myList
True
```

Identity operators

These operators check if there are references to the same object.

Operator	Function
is	Checks if it is the same reference
is not	Checks if it is not the same reference

```
>>> a=10
>>> b=20
>>> c=20
>>>
>>> a is b
False
>>> b is c
True
```

2.12 Python variables

Variables are placeholders for values. Looking at it in a physical manner, they are memory areas in your computer. The contents of the variables can be very different.

The structure of the saved information changes depending on if a character string or a floating point number should be saved. This structure is described by the **data type used** (string, integer, etc.). The **variable must have a name** assigned in order to be able to access the memory area.

It is not necessary to declare variables in Python. Python has a **dynamic data type management**.

It is not necessary to allocate and release memory in Python. Python contains an **automatic memory management** (Garbage Collection).

Dynamic type management

A variable is created in Python as soon as a **value is assigned** to it. The **type** for the variable is determined by the **value which is assigned**.

If an existing variable is assigned the value of a different type, the type of the variable will change.

```
>>> a = 10  #variable is created
>>> print a
10
>>> type(a)
<type 'int'>
>>> a = "Text"
>>> print a
Text
>>> type(a)
<type 'str'>
```

Binding - What happens internally? (binding)

A variable is created as soon as a value is assigned to it. But, what happens exactly internally?

```
>>> a = 10
```

Figure 10: Binding

First of all, an instance (object) of the integer value "10" is created in memory. A reference is then created from the variable "a" to the instance. This process is called binding. The variable name "a" is bound to the instance "10".

```
>>> a = 10
>>> a = "Text"
```

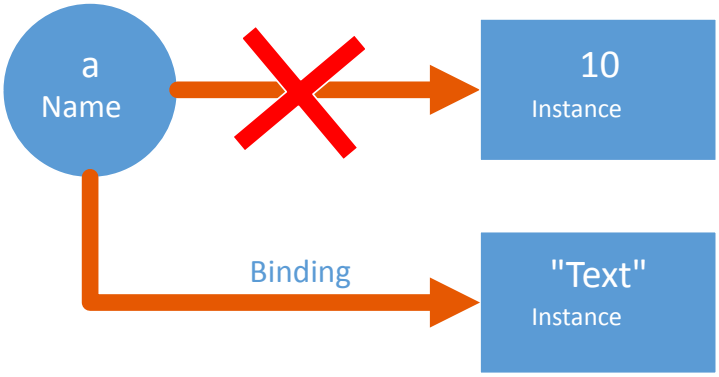


Figure 11: Rebinding

In this example, the instance for the value "10" is created in the memory first and then the name "a" is bound to the instance. Afterwards, the instance for the string "Text" is created and the name "a" is removed from the instance of the value "10" and rebound to the new string instance (rebinding). No variable is bound to the old integer instance. The Garbage Collection (memory cleaning) takes care of the old instance and removes it from the memory at a convenient period in time.

Changeable and unchangeable data types

The following code example shows one of the main properties of variables in Python. A distinction is made between mutable (changeable) and immutable (unchangeable) data types.

```
>>> a = 10
>>> b = 10
>>> id(a)
31043556
>>> id(b)
31043556
```

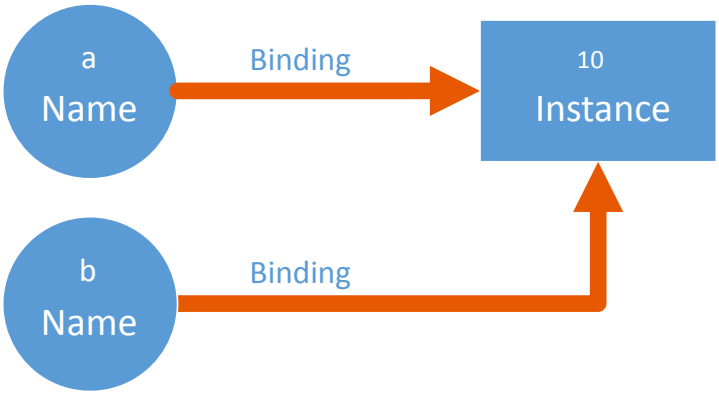


Figure 12: Binding to the same instance

There are immutable (unchangeable) data types in Python. The whole number data type "Integer" is one of them. An immutable data type cannot change its content during runtime and is thus unchangeable. Because the content of the variable cannot change, it does not make sense to create two or more instances for the same integer value in the memory. This is why both variables ("a2 and "b") in the example above reference the same instance. The address of a variable can be read out with the "id" command.

As soon as the variable gets a new value assigned, the variable name references a new instance which is stored in another part of memory. In this example, you can see that the memory address of the variable has changed after the new value assignment.

```
>>> a = 10
>>> id(a)
31043556
>>> a = 11
>>> id(a)
31043544
```

The mutable (changeable) data types are characterized by being able to change their content during runtime. This is like adding an element to a list during runtime, as in the data type "list". Python presumes that variables of this type will be changed anyway during runtime. This is why it does not make sense to refer two different variables to the same instance. Two variables which have the same content will obtain different addresses from the beginning if they are of a mutable type.

```
>>> a = ["x"]
>>> id(a)
43589384
>>> b = ["x"]
>>> id(b)
43611960
```

rules for variable names

- The first character of a variable name must be a capital, a small letter, or an underscore ("_").
- The rest of the variable name must be composed of capital or small letters, numbers (0-9), or underscores ("_").
- Variable names are case-sensitive. There is a differentiation between case.

The following variable names would thus not be allowed:

- 4fun
- variable 1
- test-variable

2.13 Python data types

There are very powerful built-in data types in Python. This chapter should give you an overview of the different types. The types are divided into the following groups:

- Numeric types
- Sequential types
- Mappings (Dictionary)

The type None

The "None" type is special in Python. This type represents "nothing". It can be compared to "NULL" in other languages. None can be used to initialize things in Python. An identity check with None can be performed at best with the "is" operator.

```
>>> a = None
>>> print a
None
>>> type(a)
<type 'NoneType'>
>>> a is None
True
```

2.13.1 Python numeric types

Python has the following numeric data types:

Data type	Description
int	Integer, whole numbers, up to 2^64 on 64-bit machines
long	Long integer, whole numbers, infinitely long, limited by the memory
float	Floating point numbers
complex	Complex numbers
bool	Boolean values (True or False)

All numeric data types are immutable (unchangeable).

The numeric data types can be used as follows:

```
>>> a = 10
>>> type(a)
<type 'int'>
>>> b = 12L
>>> type(b)
<type 'long'>
>>> c = 3.4
>>> type(c)
```

```
<type 'float'>
>>> d = 2 + 3j
>>> type(d)
<type 'complex'>
>>> e = True
>>> type(e)
<type 'bool'>
```

2.13.2 Python sequential data types

Python differentiates between the following sequential data types:

- Strings (immutable - unchangeable)
- Lists (mutable - changeable)
- Tuple (immutable - unchangeable)

2.13.2.1 Python type String

Strings are character chains. They are sequences of single characters. It is therefore possible to perform all operations which can be used on sequences.

Create strings

Strings are created by using quotation marks. Either single, double quotation marks, or triple single or double quotation marks can be used. It is important that the same type of quotation marks are used at both ends of the string.

```
>>> a = "this is a string"
>>> print a
this is a string
>>> b = ' another string'
>>> print b
 another string
>>> c = """This is a multiline
string.

Next line."""
>>> print c
This is a multiline
string.

Next line.
```

Escape sequences

These are special control characters which influence the display of the text on the monitor, but are not displayed themselves. Escape sequences are introduced with a backslash ("\").

Escape sequence	Description
\n	Linefeed. Linefeed.
\t	Tabulator

Table 1: Escape sequences

Escape sequence	Description
\"	Double quote
\\	Backslash

Table 1: Escape sequences

```
>>> a = " string containing \"quotes\" in python"
>>> print a
 string containing "quotes" in python
>>>
>>> b = " string with linefeed.\n Next Line."
>>> print b
 string with linefeed.
 Next Line.
```

Raw Strings

If you want to create a string in which control characters are not only handled as such, but also output as their raw value, it is possible to use a raw string. Raw strings are marked with an "r" prefix.

```
>>> a = r"string with linefeed. \n Next Line."
>>> print a
string with linefeed. \n Next Line.
```

Unicode Strings

Unicode strings can also contain special characters from different character sets. A Unicode string is marked with a "u" prefix. You can see in the example that "Unicode" has its own data type in Python.

```
>>> a = u"string with special characters like ä"
>>> print a
string with special characters like ä
>>> type(a)
<type 'unicode'>
```

Operations

Operation	Result	Description
a + b	"Yeah now "	Fit together
a * 3	"Yeah Yeah Yeah"	Review
a[1]	"e"	Access via index
a[2:4]	"ah"	Section , Part
len(a)	5	Length
"Y" in a	True	Membership

Table 2: String - operations; a="Yeah "; b="now "

Methods

The "string" and "Unicode" data types have useful methods.

Method	Result	Description
a.find("e")	4	Detects the first occurrence of the search string

Table 3: String methods; a="My test string."

Method	Result	Description
a.count("t")	3	Number of occurrences of the search string
a.replace("e", "oa")	"My toast string"	Replace search string
a.upper()	"MY TEST STRING."	Convert to capital letters
a.lower()	"my test string."	Convert to small letters
a.split()	['My', 'test', 'string.']	Split at each whitespace.

Table 3: String methods; a="My test string."

Format

The formatting operator brings the string output in the desired format. The formatting operator ("%") is used in the same way as printf in the language C.

```
>>> a = "He is %d years old." % (40)
>>> a
'He is 40 years old.'
>>>
>>> "The value is %3.2f ." % (1.234567)
'The value is 1.23 .'
```

identifier	Description
%d	Whole numbers with a sign
%u	Whole numbers without a sign
%x	Hexadecimal
%f	Floating point number
%s	String

Table 4: Formatting identifier

Other flags which control the output may be present between the formatting operator ("%") and the identifier. The meaning of the flags can be found in an extended documentation.

Concatenation

If one string is written directly after another, they will be concatenated automatically.

```
>>> a = "Part1" "Part2"
>>> a
'Part1Part2'
```

Use of variables and strings

1) Use the interactive mode for this task.
2) Create variables with numeric types (integer, float, complex, Boolean).
3) Create a variable of the type string.
4) Use escape sequences and raw strings.
5) Concatenate two strings.
6) Use the "replace" method on a string variable.

7) Use the format operator on a string variable.

8) Try to change the content of a string variable via accessing the index.

The string is a data type which cannot be changed (immutable). Changing the content via index operator is therefore not allowed and leads to an exception.

2.13.2.2 Python Type List

A list is an ordered collection of objects in Python. Lists can contain objects with different data types. The list can be changed during runtime (mutable). Elements can be changed, added or deleted.

A list is created using a square bracket.

```
>>> a = [ 44, "Text", "ABBA"]
>>> type(a)
<type 'list'>
```

Operations

Operation	Result	Description
a = [ ]	[ ]	Empty list created
a = [44, "Text", "ABBA"]	[44, "Text", "ABBA"]	List with content created
a[2]	"ABBA"	Access via index
a[1:]	["Text", "ABBA"]	Section / Part
a[0]=555	[555, 'Text', 'ABBA']	Change content via index
del a[0]	['Text', 'ABBA']	Delete elements

Table 5: List operations

Methods

The "list" data type has powerful methods. Here is a short excerpt from that.

Method	Result	Description
a.append(1)	[1]	Append element
a.extend(b)	[1, 'a', 'b']	Hang all elements from b to a
a.count("a")	1	How often is the search string present in the list
a.remove("a")	[1, 'b']	Remove the segment.
a.reverse()	['b', 1]	Reverse order
a.sort()	['b', 1]	Sort list

Table 6: List methods; a=[ ]; b=["a","b"]

2.13.2.3 Python type Tuple

A tuple a sequence of objects in Python. In contrast to lists, tuples cannot be changed (immutable). After the tuple has been created, nothing more can be removed or added. If anything should be added to an existing tuple, a new tuple must be created.

A tuple is created using round brackets.

```
>>> a = ( 44, "Text", "ABBA")
>>> type(a)
<type 'tuple'>
```

Operations

Operation	Result	Description
a = ( )	( )	Empty tuple created
a = (44, "Text", "ABBA")	(44, "Text", "ABBA")	Tuple with content created
a[2]	"ABBA"	Access via index
a[1:]	["Text", "ABBA"]	Section / Part

Table 7: Tuple operations

Methods

The "tuple" data type has few methods in comparison to a list.

Method	Result	Description
a.count(44)	1	How often is the search string present in the list
a.index(44)	0	Output the index of the element

Table 8: Tuple methods; a = (44, "Text", "ABBA")

Use of the list type

1) Use the interactive mode for this task.
2) Create a shopping list.
3) Create a variable of the type "list" for this purpose.
4) Use the "append", "remove", "sort" and "reverse" methods on your list variable.
5) Change an element in your list by using an index access.

Use the tuple type

1) Use the interactive mode for this task.
2) Create a variable of the type "tuple".
3) Use the "count" method on your tuple variable.
4) Try to change an element in your tuple by using an index access.

A tuple is a data type which cannot be changed (immutable). Changing the content via index operator is therefore not allowed and leads to an exception.

2.13.3 Python type Dictionary

The "Dictionary" (dict) Python type is a unsorted collection of objects. This is a collection of key/value pairs. Each objects can be accessed by using a key.

The keys must be unique. Only objects which cannot be changed (immutable) are allowed as a key.

The values must not be unique. The same value can therefore exist several times (with different keys) in the dictionary. Values can have data types which are changeable and unchangeable.

A dictionary is created using curly brackets.

```
>>> a = { "Joe":"red", "Henry":"green", "Adam":"orange"}
>>> type(a)
<type 'dict'>
>>>
>>> a = { 1:"text", 90:5, "keytext":(10,20,100) }
>>> type(a)
<type 'dict'>
```

Operation	Result	Description
a = { }	{ }	Empty dictionary created
a = ("Joe":"red", "Henry":"green", "Adam":"orange")	{'Joe': 'red', 'Henry': 'green', 'Adam': 'orange' }	Dictionary with content created
a["Joe"]	"red"	Access using key
"Adam" in a	True	Membership
del a["Henry"]	{'Joe': 'red', 'Adam': 'orange'}	Delete element

Table 9: Dictionary operations

A whole series of methods have been defined for dictionaries.

Method	Result	Description
a.keys()	['Joe', 'Henry', 'Adam']	Creates a list of the keys
a.values()	['red', 'green', 'orange']	Creates a list of the values
a.has_key("Adam")	True	Check membership
a.clear()	{ }	Delete dictionary content

Table 10: Dictionary methods; a = { "Joe":"red", "Henry":"green", "Adam":"orange"}

Use the dictionary type

1) Use the interactive mode for this task.
2) Create a variable of the type "dictionary".
3) The dictionary keys should be used for names.
4) The dictionary values should contain telephone numbers.
5) Output the number of elements contained by using the built-in "len" function.

6) Use the "keys" and "values" methods on your dictionary variable.

7) Delete an element in your dictionary by using the "del" operator.

2.14 Python Functions

In order to increase the re-usability of written code and avoid unnecessary redundancy, it is recommended that code is written in functions.

The "def" key word defines a function in Python. The name of the function comes after the "def", and then all of the variables which should be transferred in brackets. Finally, a colon is set and all of the program lines which belong to the function body are indented one level.

If the function should return a result to the main program, the key word "return" must be used.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def myAddFu(var1, var2):
    result = var1 + var2
    return result

calcResult = myAddFu(1,2)
print calcResult
```

2.14.1 Python Functions Parameters

There are different possibilities to transfer parameters to a function.

Positional Parameters

This is the standard way of transferring parameters. All of the parameters which are defined by the function must be transferred. The order must also be the same as defined in the function.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def myAddFu(var1, var2):
    result = var1 + var2
    return result

calcResult = myAddFu(1,2)
print calcResult
```

Optional parameters

Python also allows function call parameters to be optional. A default value must be specified in the function definition, for each optional parameter, in order to make this possible. A parameter which is not optional is not allowed to come after an optional parameter.

```
#!/usr/bin/env python
```

```
# -*- coding: utf-8 -*-

def myAddFu(var1, var2, var3=10):
    result = var1 + var2 + var3
    return result

calcResult = myAddFu(1,2)
print calcResult

calcResult = myAddFu(1,2,11)
print calcResult
```

Key Word Parameters

The parameters are allocated over their position when calling and using the positional parameters. With the parameter key word, the allocation takes place over the name of the parameter. If you know the name of the parameter, it can be written in any place when calling. This does not change anything for the function definition.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def myAddFu(var1, var2, var3=10):
    print "var1: %d \t var2: %d \t var3: %d " % (var1,var2,var3)


myAddFu(1,2)

myAddFu(var3=17,var2=10,var1=3)
```

Use of Functions

1) Create a source file with the help of the 'kwrite'.
2) Insert the Shebang line and the encoding line.
3) Program a function which is intended to be called with positional parameters. Call the function.
4) Program a function which contains optional parameters. Call the function.
5) Call a function by using the parameter key word.
6) Also use the "return" statement in one of your functions.

2.15 Python modules

One the one side, modules encapsulate program code which belongs together in order to structure a large program. On the other side, modules are collections of functions, classes, methods, and constants for certain purposes (libraries). Python offers extensive standard libraries. The corresponding modules only have to be imported in order to be able to use the library functions.

Create your own module

Your own code can be encapsulated simply in a module. The module's file extension must be ".py". The module can only then be integrated into another program.

The module should be stored in the same directory as the main program file. Or in a directory which is in the Python system path (sys.path).

Import module

A library is incorporated with the "import" key word. The module is incorporated together with its name-space here. All functions and constants can be accessed via the "<module name>.<function name>" syntax.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import math

print math.sin(1.5)
```

It is also possible to import a module with a different namespace. This is done using the "as" key word.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import math as myMath

print myMath.sin(1.5)
```

The entire content of a module can also be imported into the global namespace of the program. The content of the module can then be accessed directly via the names. The prefix "module name" is dropped in this case.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from math import *

print sin(1.5)
```

Single functions or constants can be incorporated by using the "from <module name> import" statement.

```
#!/usr/bin/env python
```

```
# -*- coding: utf-8 -*-

from math import sin

print sin(1.5)
```

Integrated dir() function

The built-in "dir" function lists all of the identifiers which are defined in a module.

```
>>> import math
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp',
'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Use of modules

1) Use the interactive mode for this task.
2) Import the "math" module.
3) Use the module's "pi" attribute.
4) Use the built-in "dir" function on the "math" module to show its content.

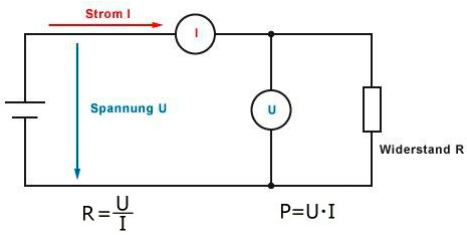
2.16 Python program examples

2.16.1 Ohm's law

A draft should be made for a menu guided program which can calculate the following elements:

- the voltage **U** [V]
- the current **I** [A]
- the resistance **R** [Ω]
- the power **P** [W]

The user should also have to enter their first name and surname after starting the program. After the user has made the entries, they should be welcomed and able to navigate in the menu.



2.16.1.1 Short description

Program: OhmschesGesetz.py

It is possible to select different calculations from a menu in this program. The voltage, current, electrical resistance, and the electrical power can be calculated. The values which are necessary for this must be entered beforehand. Furthermore, you can close the program by using a menu item. The input of a false value is also taken into account and locked in the menu.

The user is greeted when starting the program. This is done with a function.

The following Python elements are included:

- Declare and call functions
- IF - statement
- WHILE - statement
- Output / read string
- Convert data type
- Comparison operators
- Logic operators
- Basic types of calculation
- Source code comments

Formula:

Ohm's law is:

U = R * I

The formula for the electrical power is:

P = U * I

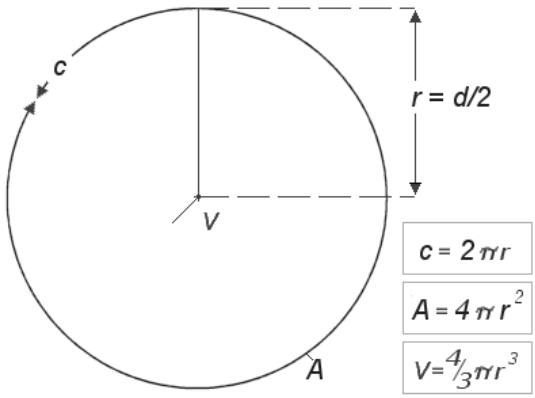
2.16.1.2 Python code

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

eingabe=1
#Wird mit 1 beschrieben, um das Erste mal in die
#while- Schleife zu gelangen (0=beenden)
# - - - Funktion begruessung() - - -
def begruessung(Vorname,Nachname):
    print "\nHallo",Nachname,Vorname,"!"

# - - - OhmschesGesetz.py - - -
print "___--- DAS OHMSCHE GESETZ ---___\n"
Vorname=raw_input("Ihr Vorname lautet: ")
Nachname=raw_input("Ihr Nachname lautet: ")
begruessung(Vorname,Nachname) #Funktionsaufruf

# - - - Programm läuft solange eingabe nicht gleich 0 ist- - -
```

```
while (eingabe!=0):
    # - - - Menue und Auswahl- Eingabe - - -
    print "\nMenue:\n* [0] Beenden\n* [1] die Spannung ausrechnen\n* \
[2] die Stromstaerke ausrechnen\n* [3] den Widerstand ausrechnen\n* \
[4] die Leistung ausrechen"
    print "Bitte waehlen Sie ein Modul aus!\n"
    eingabe=raw_input("Auswahl: ")
    eingabe= int(eingabe) #Datentyp konvertieren

    # - - - Berechnen der Spannung - - -
    if eingabe==1: #U=R*I
        print "\nBerechnen der Spannung [V]: U = R*I\n"
        I=raw_input("Stromstaerke [A]: ")
        R=raw_input("Widerstand [Ohm]: ")
        R = int(R) #Datentyp konvertieren
        I = int(I)
        U=R*I
        print "\nErgebnis: %d" % (U),"V"

    # - - - Berechnen der Stromstaerke - - -
    if eingabe==2: #I=U/R
        print "\nBerechnen der Stromstaerke [A]: I = U/R\n"
        U=raw_input("Spannung [V]: ")
        R=raw_input("Widerstand [Ohm]: ")
        U = int(U) #Datentyp konvertieren
        R = int(R)
        I=U/R
        print "\nErgebnis: %d" % (I),"A"

    # - - - Berechnen des Widerstandes - - -
    if eingabe==3: #R=U/I
        print "\nBerechnen des Widerstandes [Ohm]: R=U/I\n"
        I=raw_input("Stromstaerke [A]: ")
        U=raw_input("Spannung [V]: ")
        U = int(U) #Datentyp konvertieren
        I = int(I)
        R=U/I
        print "\nErgebnis: %d" % (R),"Ohm"

    # - - - Berechnen der Leistung - - -
    if eingabe==4: #P=U*I
        print "\nBerechnen der Leistung [W]: P = U*I\n"
        U=raw_input("Spannung [V]: ")
        I=raw_input("Stromstaerke [A]: ")
        U = int(U) #Datentyp konvertieren
        I = int(I)
        P=U*I
        print "\nErgebnis: %d" % (P),"W"

    # - - - Eingabe für falsche Werte sperren - - -
    if ((eingabe!=0) and (eingabe!=1)
    and (eingabe!=2) and(eingabe!=3) and (eingabe!=4)):
```

```
        print "Waehlen Sie ein Modul zwischen 0 und 4aus!"

# - - - Programmende - - -
print "\nDas Programm wurde beendet!"
```

2.16.2 Volume of a ball



Enter the radius of the ball in the unit centimeters. Output the volume in cm³ and liters. Also round the result.

2.16.2.1 Short description

Program: Kugel.py

The use can enter a radius in this program. The program then calculates the volume of a ball which has that radius in different units.

The following Python elements are included:

- Functions from the math module

Formula:

The formula for the volume calculation is:

$$V = \frac{4}{3} * \pi * r^3$$

2.16.2.2 Python code

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

```
# - - - Volumen einer Kugel berechnen - - -
from math import * #math Funktionen importieren

r = raw_input("Geben Sie den Radius Ihrer Kugel in cm ein:")
r = int (r) #Datentyp konvertieren
print "\nV = (4 * pi * r**3) / 3"
print "\nV = (4 * pi *",r,"**3) / 3\n"
V = (4 * pi * r**3) / 3
V1 = V / 1000
print "Ihre Kugel mit dem Radius",r,"hat ein Volumenvon",V,"cm3",V1,"l",
V = ceil (V) #aufrunden

V1 = ceil(V1)
print "\nAufgerundet ergibt das",V, "cm3",
print "\nbeziehungsweise ein Fassungsvermoegen von",V1,"l",
```

2.16.3 Address Book

Enter five first names, five surnames, and the corresponding telephone numbers. Then output the numbers which were entered.

2.16.3.1 Short description

Program: Adressbuch.py

The user enters five names and the corresponding telephone numbers in this program. The names and numbers should then be output.

The following Python elements are included:

- Arrays

2.16.3.2 Python code

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

i = 0 #Zaehlvariable auf Null setzen
print "Geben Sie bitte fuenf Namen und Telefonnummern ein,\
mit denen ihr Telefonverzeichnis beschrieben werden soll!\n"
```

```
Eintrag1 = 0
Eintrag2 = 0
Eintrag3 = 0
Eintrag4 = 0
Eintrag5 = 0
TelBuch = [Eintrag1,Eintrag2,Eintrag3,Eintrag4,Eintrag5]
#Erzeugen des Array's

# - - - Beschreiben der Eintraege mit einem Array
for Eintrag in TelBuch:
    print "Geben Sie die Daten fuer Eintrag",i+1;"ein./n"
    a = raw_input("Name: ")
    b = raw_input("Tel: ")
    b = int (b)
    TelBuch[i] = [a,b]
    i = i + 1

# - - - Ausgabe des Gesamten Array's - - -
print TelBuch

# - - - Ausgabe jedes einzelnen Elementes für sich - - -
print "\nTELEFONBUCH"
print "Name, Telefonnummer\n"
print TelBuch[0]
print TelBuch[1]
print TelBuch[2]
print TelBuch[3]
print TelBuch[4]
```

3 Using Python in APROL

3.1 Motivation

This Python expansion makes it possible for you to implement customized demands in the visualization (also in the operation) corresponding to your operating philosophy. ***APROL*** can and should replace existing „old process control systems“.

These “old” systems often work with an operating philosophy which is not implemented in ***APROL***. On the one side, the operation of ***APROL*** technology should remain the same, on the other side, you as the customer who has changed over to ***APROL***, should also be offered the possibility to implement your previous operation technology. Both options must be able to exist alongside each other, without having to customize ***APROL*** for each customer.

Note:

Using Python interfaces is not a replacement for the engineering of function plans!
This interface gives you the option of being able to access certain system properties of ***APROL***, and the handling can be customized to your needs.

3.2 APROL components with a Python interface

There are several APROL components which have Python interfaces.

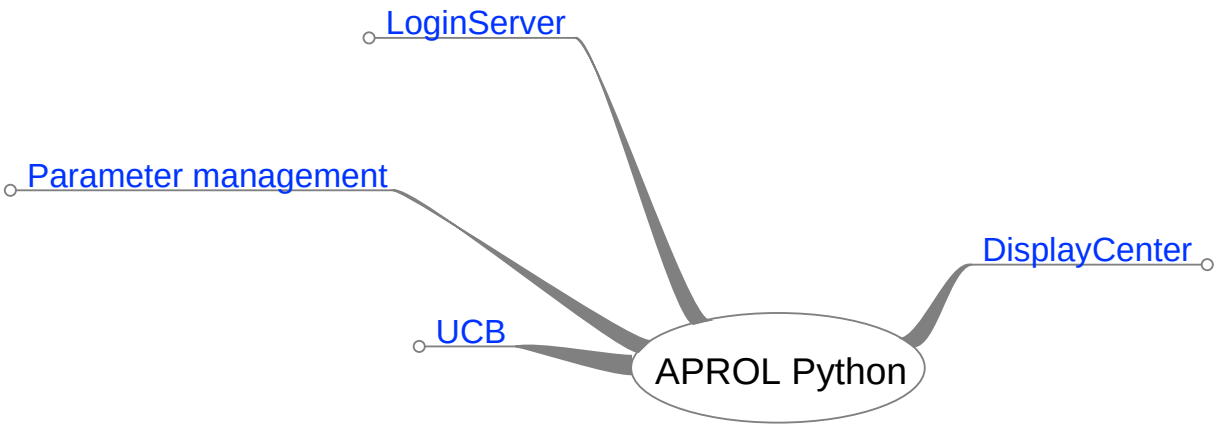


Figure 13: APROL components with a Python interface

3.3 APROL Python Documentation

The APROL product documentation for the Python API can be found in manual **D2**.

3.4 Python code context

The Python code is executed within a certain context. The Python code runs in the context of the project part in which it is placed.

If the Python code is written in a global module for the DisplayCenter, the code will be active on the corresponding runtime system as long as the corresponding system is active.

Python code is only processed once from top to bottom.

The code can also use hooks. These are callback functions which will be processed when certain events occur. Only the intended functions are hung onto the callback stack of the respective event while the code is being processed from top to bottom. All of the functions in an event's callback stack will be executed if the desired event occurs at a later stage (the top to bottom code processing has then normally finished).

The execution may take a long time, because of the use of loops or time-intensive functions. An application may be impeded by the Python code which is running in its context.

The use of loops or time-intensive functions may block or delay the execution of the application.

Python code is written in the context of a graphic block of a CAE library. The code is active as long as the process graphic where the block was placed is open. As soon as another process graphic which does not contain this block is opened, the code will no longer be present. Python code and its objects therefore have a limited time of effect.

Python code is not suitable for cyclic programming. Its lifespan is limited to the context!

3.5 APROL Python Hooks

The components which offer a Python API in the APROL system have different hooks. A user can hang any, and mostly self-written functions on these hooks. Depending on the context, there are many **"add_hook"** functions to connect the hook to the user function.

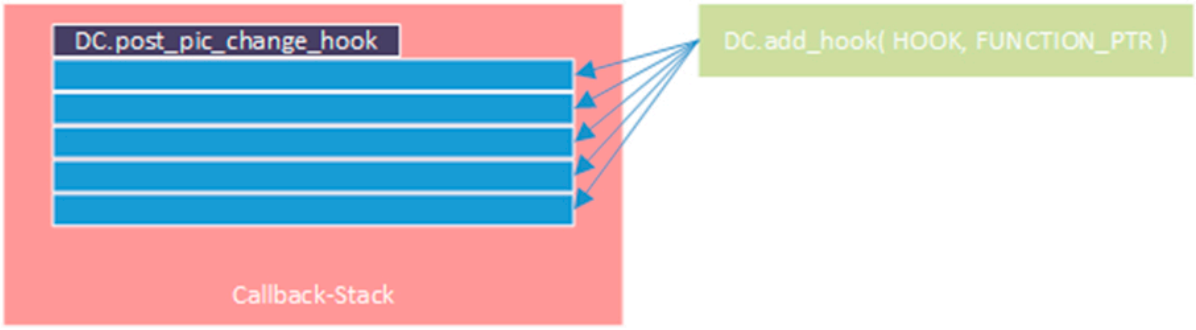


Figure 14: Hook and Callback Stack

A **Hook (=anchor)** is a way to execute a desired function in the system if a certain **Event** occurs.

Hooks are understood as being callback functions which will be processed when certain events occur. Only the intended functions are hung onto the callback stack of the respective event while the code is being processed from top to bottom. All of the functions in an event's callback stack will be executed if the desired event occurs at a later stage (the top to bottom code processing has then normally finished).

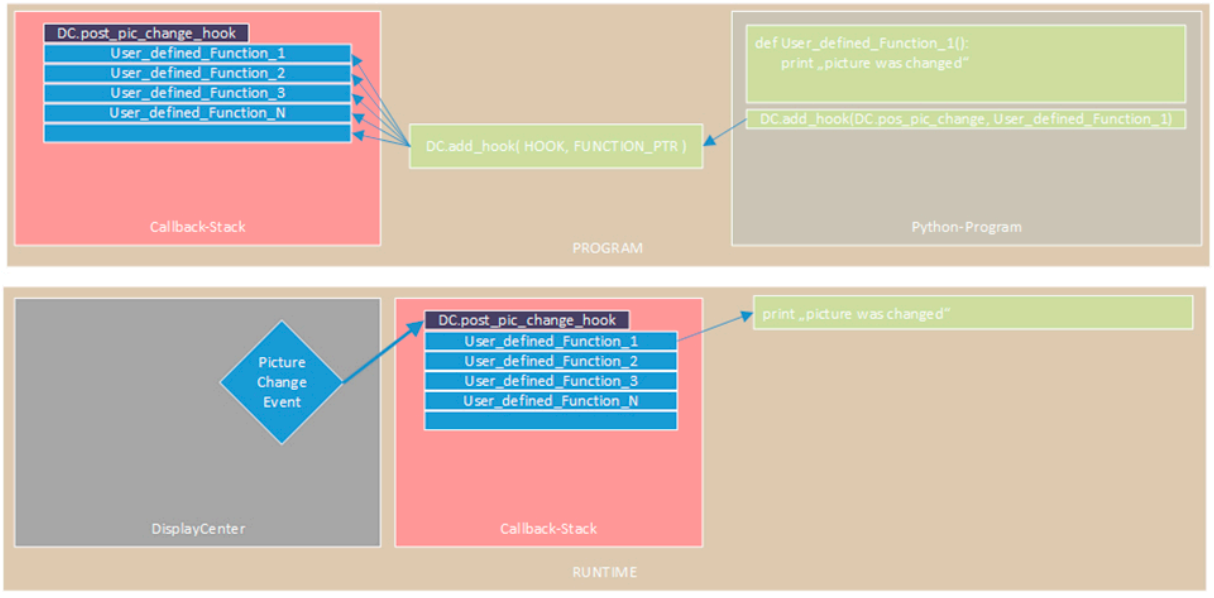


Figure 15: Hook in a program and event during runtime

3.6 APROL Python in the DisplayCenter

The python API in the DisplayCenter offers extensive ways to extend the visualization. It refers to the DisplayCenter and therefore has a strong connection with the visualization. The API can be used in different areas.

Areas of use

The python API for the DisplayCenter can be used in the following project parts.

- Global module for the DisplayCenter
- Global module for process graphics
- Global module for CAE libraries
- Process graphic (Python code tab)
- Graphic macros and templates in a CAE library (Python code tab)

The global "**Python Module for the DisplayCenter**" is conceived to store a collection of Python code in one module. This module is then assigned to a runtime or operator system. This assignment is made in the APROL system project part (runtime or operator). The desired Python module must be selected in the "**-pythonModule**" module in **"Monitoring->DisplayCenter"**. Attention: only Python modules of the type "Python module for the DisplayCenter" are shown. A different "Python module for the DisplayCenter" can be used for each runtime or operator system. The module is loaded as soon as the DisplayCenter is started. The Python code remains active during the entire lifespan of the DisplayCenter. This type of module is suitable for code which should be available **globally in the DisplayCenter**. One application could be setting the AlarmMonitor alarm filter each time a graphic is switched to.

The global "**Python Module for process graphics**" is conceived to be a collection of Python code which is used more than once in the project. This module can then be imported into any process graphics, and the code in the module is then available in the process graphic. You could therefore create something like a function which sets the AlarmMonitor alarm filter to a certain value. Some process graphics may need this function, some not. Graphics which need the function can import the **"Python module for process graphics"** and thus access the function. All of the other process graphics do not need to import the module and do not use valuable computing time on the Python code. The instance name of the global **"Python module for process graphics"** is essential for importing instructions in the Python code.

The global "**Python module for CAE libraries**" is used for the encapsulation and modularization of Python code in a CAE library. It is a collection of Python function code, classes, methods, and constants which can then be imported selectively into the graphic macros or faceplates of the corresponding library. The name of the **"Python module for CAE libraries"** is essential for the import instruction. A function could be created in a global module which calculates a string for setting the AlarmMonitor alarm filter. This function can then be used in several different blocks by importing the module and calling the function.

The python API can also be used directly in a **"process graphic"**. Process graphics have their own tab for entering Python code. The code which is created in a process graphic is only available and active in that process graphic. The same principle applies to **"graphic macros"** and **"faceplates"** in the context of a CAE library.

API functionalities

The different methods, functions and objects which are supplied by the API are described in detail in **manual D2 System API**. The following excerpt should only give an overview of some possibilities:

- Load process graphic
- Create a snapshot of the process graphic
- Open graphic macro in faceplate
- Close graphic macro in faceplate
- Process graphic geometry functions
- Start browser
- Start TrendViewer
- Start AuditTrail
- Start alarm report
- Start certain reports
- Write entry in AuditTrail
- Login/Logout functions
- Process graphic history functions
- Output name/instance/description of the current process graphic
- Output the instances of all process graphics
- AlarmMonitor: Show alarm list
- AlarmMonitor: Acknowledge alarms
- AlarmMonitor: Acknowledge horn
- AlarmMonitor: Open alarm locking dialog
- Alarm filter: Set filter (divers settings)
- Login: Query rights of block pins
- Timer functions
- Faceplate geometry functions
- Read/Write value of a block pin (graphic macro/faceplate)
- Read/Write value of a local (Python) variable
- Fill and show context menu
- Widgets: Fill/Change IOBox, ComboBox, ListView dynamically

Apart from the APROL Python API functions, the extensive functions in the standard Python libraries are always available.
This means that you can import the **"time"** module into your code at any time, for example, and use the **"time.ctime()"** function to read out the current date and time of your system.

Application examples

The following applications can be realized by using the listed functionalities:

- Login/Logout via buttons
- Navigating over a process graphic history via buttons
- Show any filled context menu (e.g. for navigation)
- Show the name/instance/description of the current process graphic
- Create AuditTrail entries with operator/timestamp when images are switched
- Button to acknowledge alarms
- Button to acknowledge alarm horn
- Set special alarm filters when a process graphic is opened
- Create a faceplate with all of the alarms of a tag number. Show text, priority, etc. Acknowledgement and locking possibilities.
- Start AlarmReport with preset alarm filter
- Check if an operator has the right for a faceplate pin. Color the I/O box according to the rights (editable look & feel)

- Turn visual elements over a time period without CC or CTRL logic code
- Query pin values and carry out calculations -> Useful for dynamizations
- Fill/Change widgets during runtime. Show/Edit/Download parameter sets in the ListView.
- Start TrendViewer with preconfigured trend curves via button
- Start AuditTrail via button from faceplate
- Dynamize the display of SVGs from the image container

Events (Hooks)

The hooks are events in the system. A customer-specific code should be executed when the event occurs. It is possible, for example, to call a function which reads out the system time and displays it in the process graphic each time a button is pressed **("button_press_hook")**.

All hooks are described in detail in the **APROL product documentation manual 'D2 System API**'. The following is an excerpt of some important hooks in the context of the DisplayCenter.

Hook	Event description
DC.init_hook	DisplayCenter was started.
DC.post_pic_change_hook	An image change has taken place. Old picture is unloaded. New is loaded.
DC.monitor[0].alarmserver_connected_hook	Connection to the AlarmServer is established
DC.monitor[0].filter.changed_hook	Alarm filter has changed
DC.login.user_changed_hook	The user which is logged in has changed. A login or logout has taken place.
block.init_hook	Belongs to a graphic macro or faceplate. An instance of this block was loaded into the process graphic for the first time.
block.show_hook	Belongs to a graphic macro or faceplate. An instance of this was shown in the current process graphic. Comes each time the block is opened. Also when it is not opened for the first time.
block.local.<name>.button_press_hook	Belongs to a Python button The Python button was pressed.
block.pin.<name>.changed_hook	Belongs to the pin of a graphic macro or faceplate. The value of the pin has changed.
block.widget.<name>.button_press_hook	Belongs to a widget (e.g. ListView). A line in the widget was clicked.

Table 11: DisplayCenter Hooks

3.6.1 Debugging

The debugging must be activated first, in order to obtain extended debug possibilities. For this purpose, the "**-showPythonConsole**" option must be used in the runtime system, in **"Monitoring->DisplayCenter"**. The integer value which is specified defines the number of lines to be shown in the debug console. Attention: the setting can only be seen if the **priority 3** options are made visible.

When the Python console is activated, all of the Python code standard outputs are diverted to the Python console. **print commands** can be added to the code, in order to obtain a debug output.

If an error occurs in the Python code, the Python console will be opened, independent of if it was activated or not.

There is also the possibility to activate the interactive mode for the Python console. An interactive console input is then possible. The "**-pythonDebugging**" option must be activated for the interactive debugging.

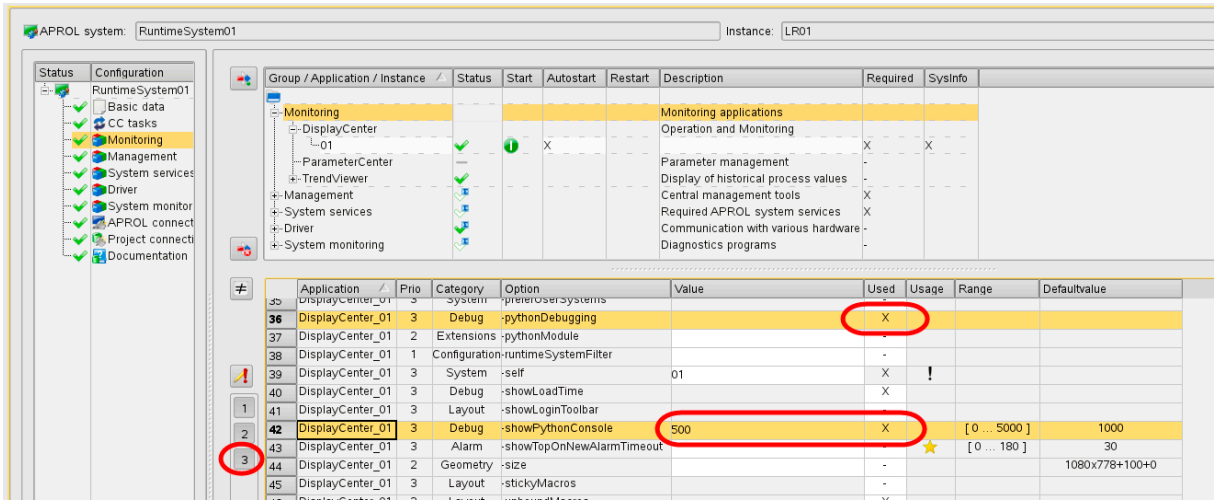


Figure 16: Activate DisplayCenter Python debugging

After the debugging has been switched on and the changes have been transferred to the runtime system, the **"Python console"** item is visible in the DisplayCenter's **"View"** menu. The Python console can be opened with that.

3.6.2 Lambda Functions

The key word **"lambda"** is used to define a **nameless (anonymous) function**.

The syntax of a lambda function is introduced by the key word **"lambda"** and is followed by a variable number of parameters. The parameters are followed by a colon which introduces the rest of the expression. Thus, any mathematical or logical expression comes after the colon.

ATTENTION: This is not allowed to contain a control structure (if, else, etc.)!

The result of the expression produces the return value of the lambda function.

```
>>> lambda a,b: a+b
```

The **lambda function** creates the same sort of function as a **"def" statement**. The difference is that a "def" statement always has a function name and therefore is always bound to that function name.

This is not the case for the **lambda expression**. The lambda expression can also obtain a name if this is desired. The lambda expression can then be used in the same way as a normal function.

```
>>> refName = lambda a,b: a+b
>>> refName(10,15)
25
```

A **lambda expression** can also be called directly.

ATTENTION: The lambda expression must be set in brackets for this purpose.

```
>>> (lambda a,b: a+b) (55,33)
88
```

Background information

In detail, the lambda expression creates a **function object (=address in memory where the function is stored)**:

```
>>> lambda a,b: a+b
<function <lambda> at 0x02B3FD70>
```

A function created with **"def"** also delivers **a function object**.

```
>>> def myAddFu(a,b):
 return(a+b)
>>> myAddFu
<function myAddFu at 0x02B68CF0>
```

Advantages

- A **lambda expression** can be placed anywhere in the code where expressions can be placed. That could be places where a "def" may not be placed. For example, in the definition of list literals (list expressions).
  ```
  >>> a = [lambda x:x**2, lambda x:x**3]
  >>> a[0](2)
  4
  >>> a[0](3)
  9
  ```
  If it were not for lambda, two different functions would have to be created with "def".
- The **lambda expressions** are practical to query **function parameters** quickly without having to create a function with "def" beforehand.
  ```
  >>> chr( (lambda x:65+x)(5) )
  'F'
  ```
- It is normal that GUI system (Graphical User Interface) APIs have **hooks** defined to execute **return functions**. In the other case, it is very practical to define the attached function in the line. This is also the case in the APROL Python API.

Use in APROL

What can lambda expressions be used for in APROL?

- The examples which have already been seen show that the different hooks in APROL deliver a different number of parameters. The **init_hook** delivers no argument, the **changed_hook** delivers one argument, and the **post_pic_change_hook** delivers two arguments. In order to be able to hang the same function on all hooks, we use the **fact** that **lambda can accept any number of parameters**. All hooks can therefore use the same function, and only the number of lambda parameters are transferred that the hook delivers. It does not matter if the parameters are used or not.
  Advantage: Modularity / re-usability of functions. The same function can be connected to all possible hooks.

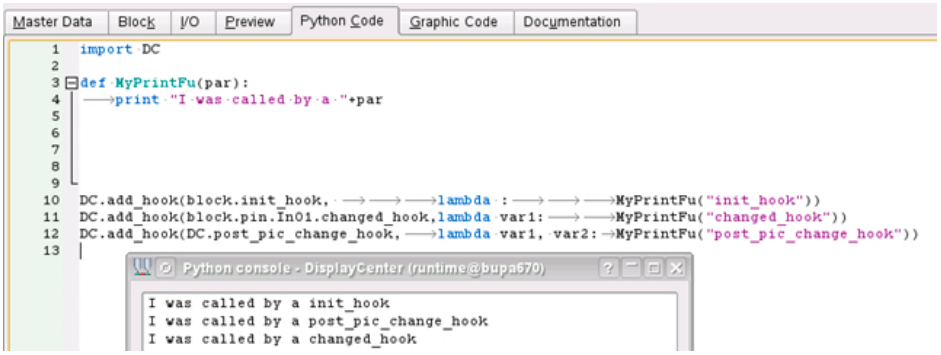


Figure 17: The same function for different types of hooks

- Only **one function object** is allowed to be hung on a one **hook**. I.e. it is only allowed to be the name of the object or in the case of lambda, it delivers a function object. It is not allowed to specify a function including function parameters, because that is a function call and not a function object. Here, we use the advantage that the **lambda expression** is only limited to being **one** expression. An expression can also be composed of a function call. It is thus possible to accommodate the function parameters in a lambda expression.
  Advantage: Modularity, re-usability of functions, efficiency: One can use functions that adopt parameters. Otherwise, an individual function must be written for the same functionality which is dependent on one parameter.

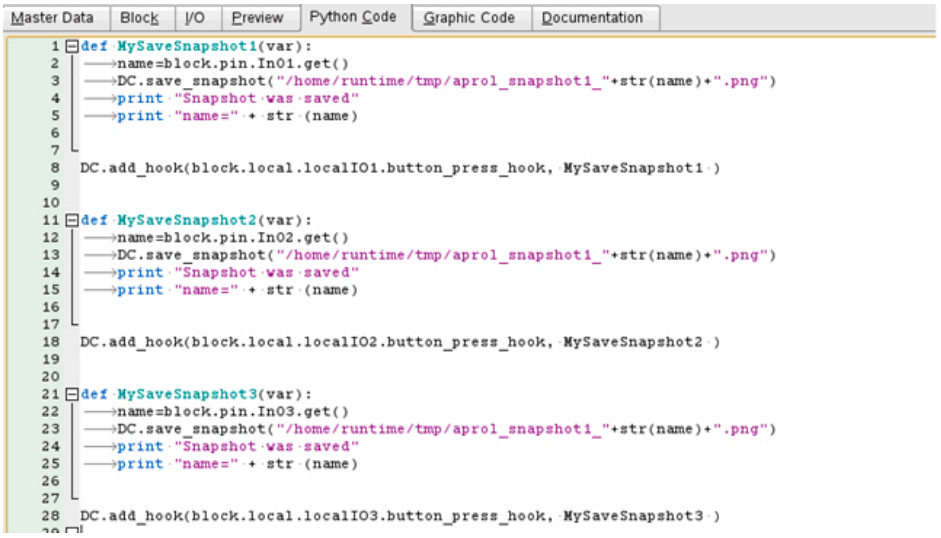


Figure 18: Parameter transfer without lambda not possible -> much code

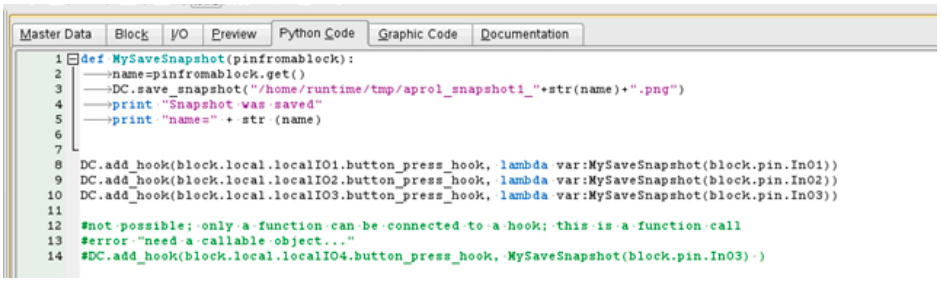


Figure 19: Parameter transfer with lambda - more efficient, re-usable code

3.6.3 Example task

Python buttons control the loading of a process graphic

A Python button should be created. A process graphic should be loaded when someone presses the button. It should be possible to input the process graphic instance in an **"InputOutputBox"** widget.

1) A new graphic macro is created in the context of a CAE library. No pins are necessary.

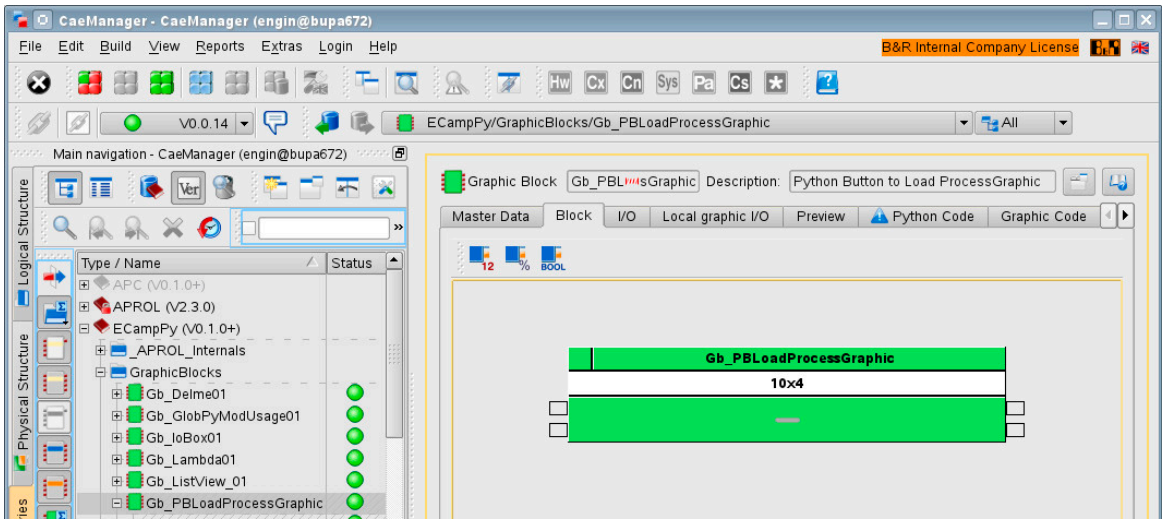


Figure 20: Create a new graphic macro

2) In the graphical part of the block, a Python button and an InputOutputBox are created.

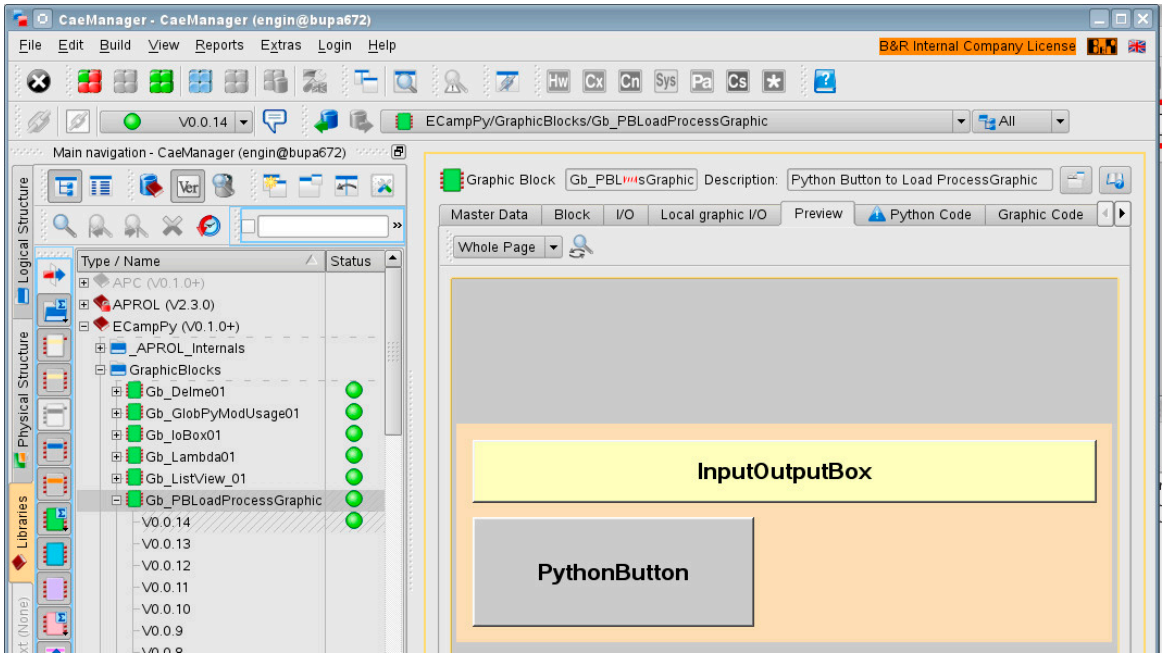


Figure 21: Graphic part of the graphic macro

3) For the Python button, it is important to set a **local variable** in the **"graphic I/O"** in the drop-down list. A variable with the name "locBtn" is used in the example.

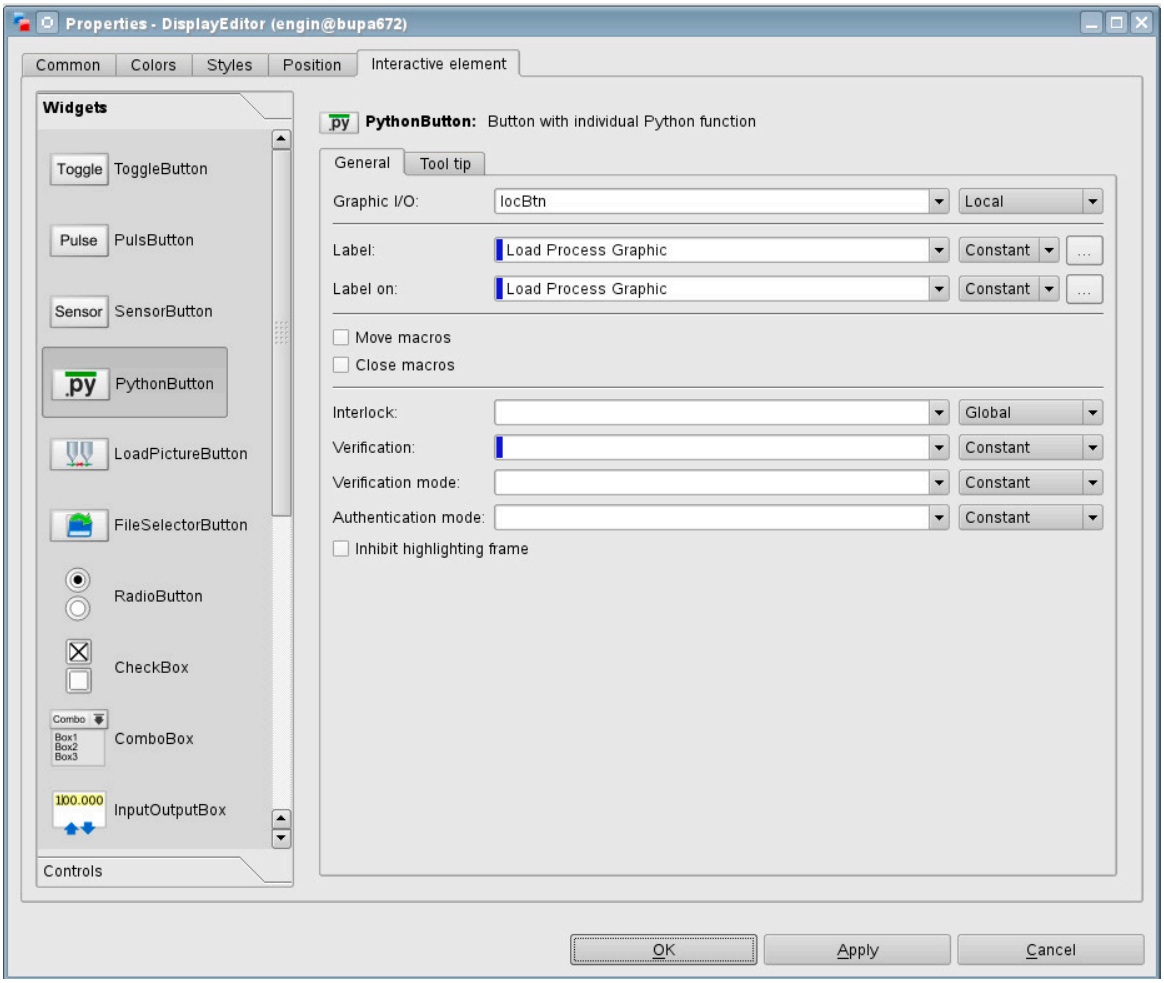


Figure 22: Configuration of the Python button

4) A **local variable** is also used as a **"control value"** for the InputOutputBox. A variable with the name "locPicInst" is used in the example.

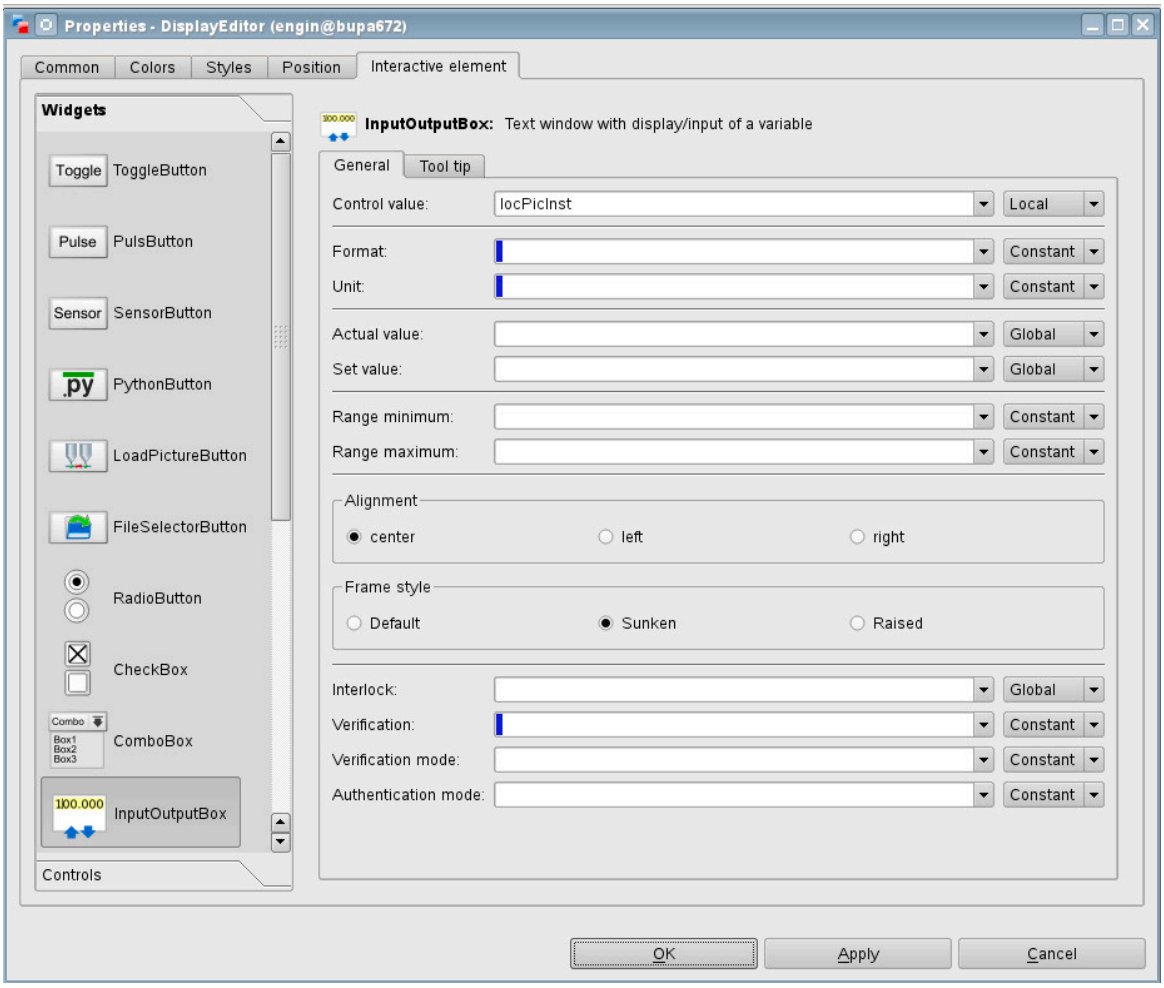


Figure 23: Configuration of the InputOutputBox

5) The following code is then entered in the **"Python code"** tab.

```
#Initialization
def myInit():
    #Set a default value to the output pin of the IOBox
    block.local.locPicInst.set("defaultPicInst")
DC.add_hook(block.init_hook, myInit)

#Functionality for process graphic change
def myFunction():
    picinst = block.local.locPicInst.get()

    result= DC.load_picture(picinst)
    if(result == True):
        print "Process graphic change successful!"
    else:
        print "Process graphic change error!"
DC.add_hook(block.local.locBtn.button_press_hook, lambda var: myFunction() )
```

The **"locPicInst" local variable** which uses the InputOutputBox is set to an **initial value** in the **"myinit"** initialization function. In this way, the box always has a defined value when it is shown in the graphical part of the graphic macro. This initialization function is hung onto the **"init_hook"** of the block. The function is therefore called every time a process graphic which contains an instance of the graphic macro is opened.

The **"myFunction"** function contains the code to load a process graphic. At first, the string which is in the **"locPicInst" local variable** is read. This string is written by the InputOutputBox. The **"load_picture"** method of the **"DC" module** is then used to load a process graphic. This method delivers a return value which tells us if loading the process graphic was successful. Depending on the success or failure of the action, the corresponding text is output in the Python console with the print statement.

This function is hung onto the **"button_press" hook** of the Python button. The function is then called every time someone presses the button. A **lambda expression** was used to receive the transferred parameter, because the hook delivers one parameter. The parameter which was transferred will not be used anymore.

6) The graphic macro is then ready and can be compiled.

7) A new CFC can now be created in the context of a project. One instance of the graphic macro is dragged from the library into the CFC.

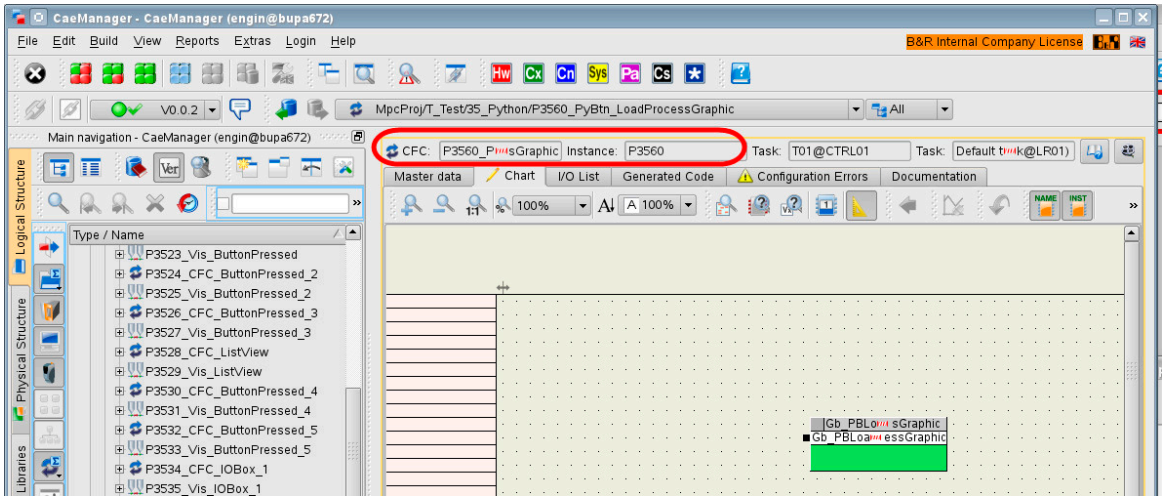


Figure 24: One instance of the graphic block in the CFC

8) A new process graphic is created afterwards. An instance of the block placed in the CFC is dragged into the process graphic.

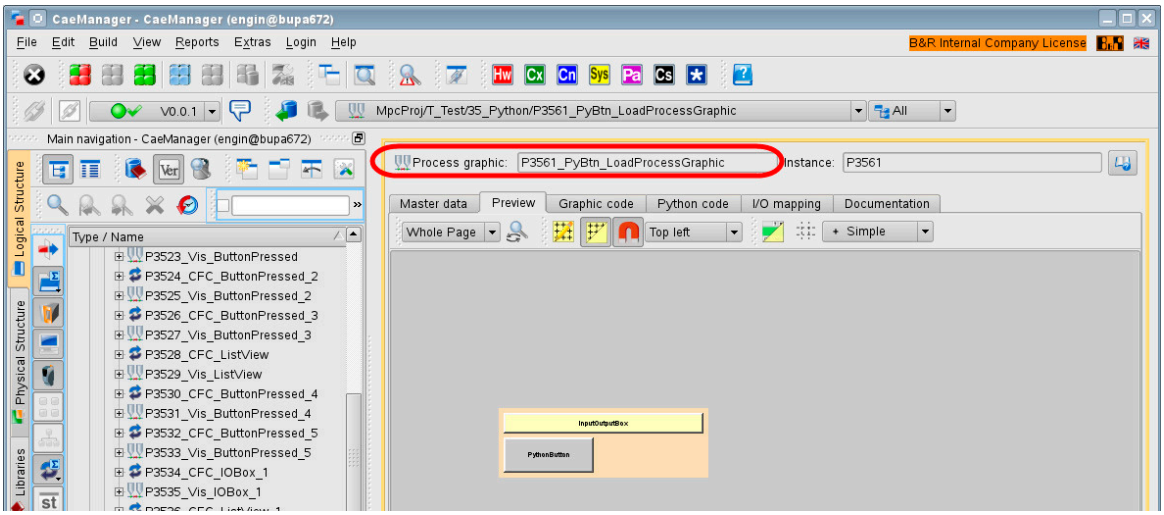


Figure 25: Instance in the process graphic

9) A build of the project is then carried out and the project is loaded subsequently to the target system.

10) If the button in the DisplayCenter is then pressed during runtime, the output will probably appear as follows: The switching of graphics is not successful because a process graphic with the instance "defaultPicInst" does not exist.

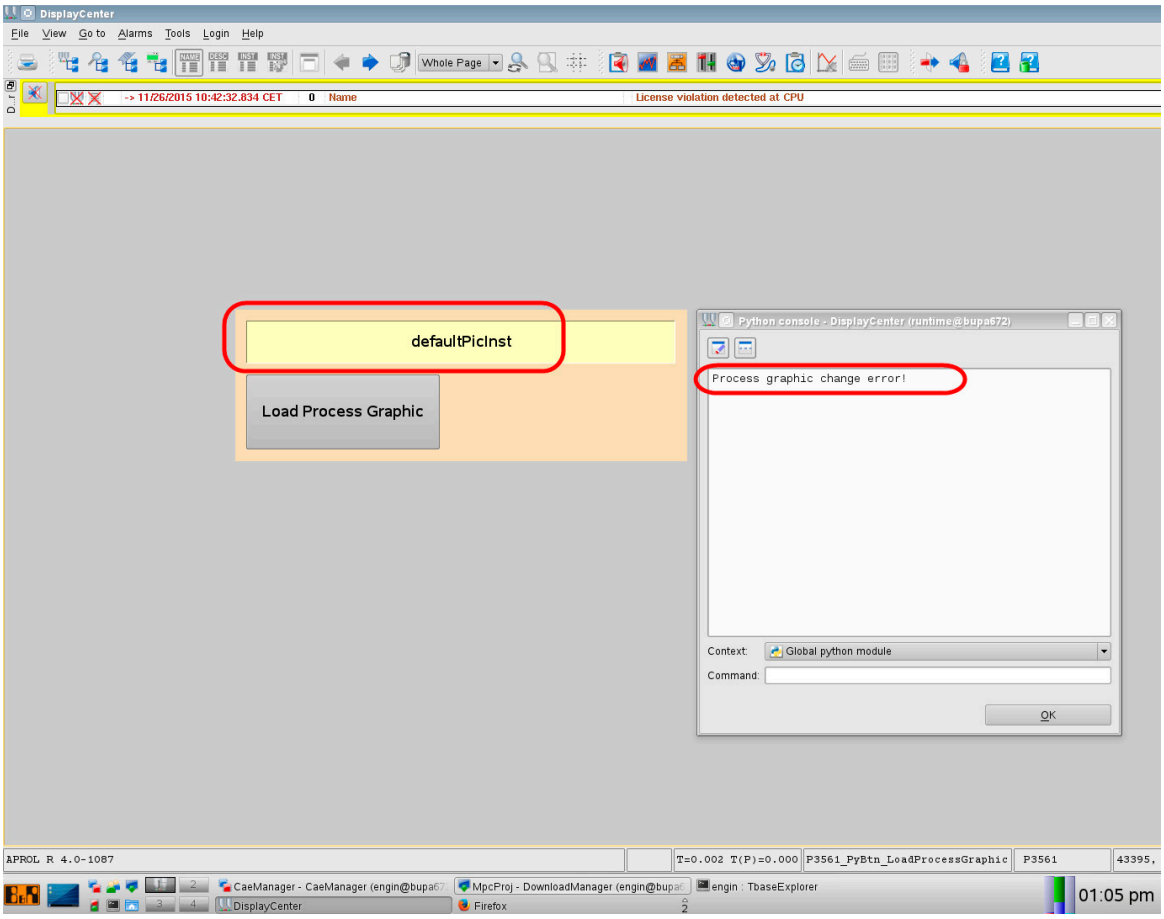


Figure 26: Loading of process graphic not successful

11) A valid value for a process graphic instance is now entered in the InputOutputBox. There must be a process graphic with this instance name in your project.

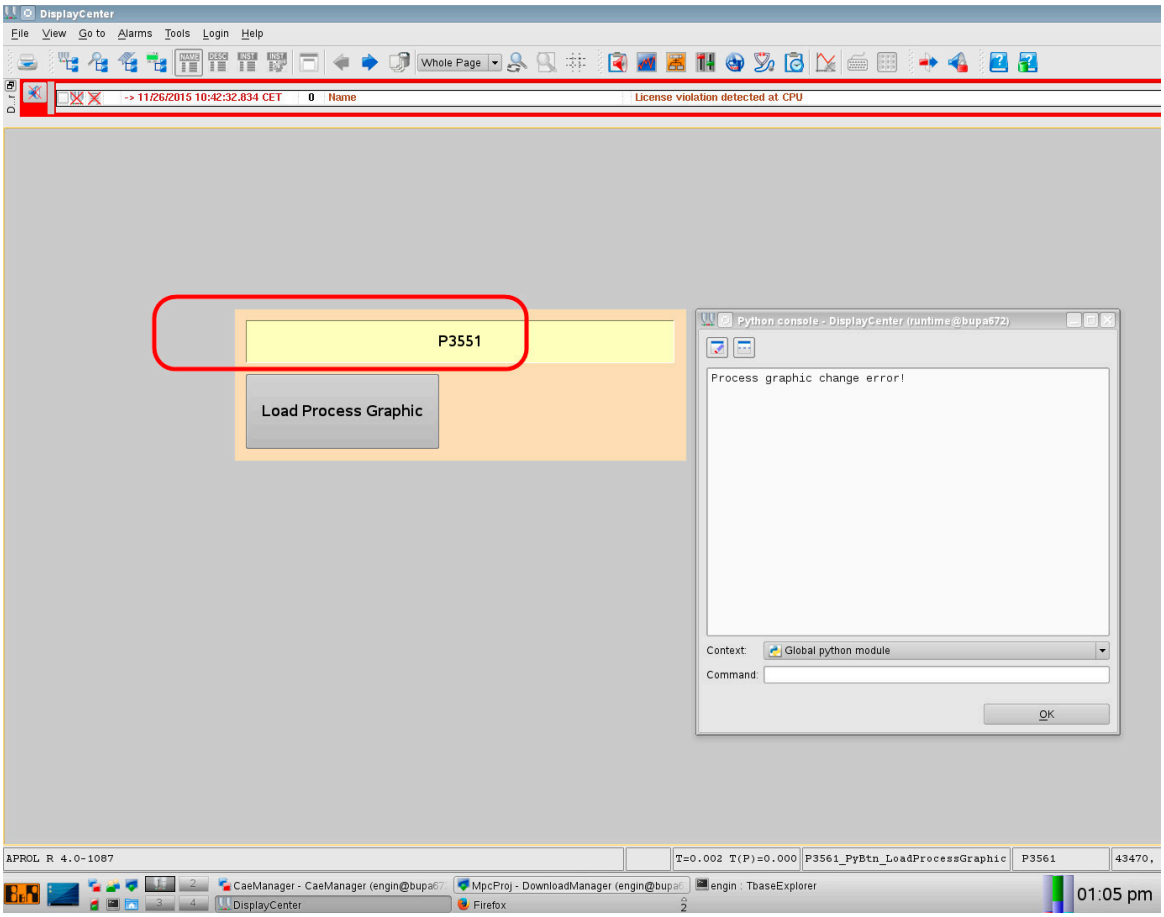


Figure 27: Inputting a valid process graphic instance

12) If the button is now pressed, the switch is made to a new process graphic and you can see this was successful in the Python console.

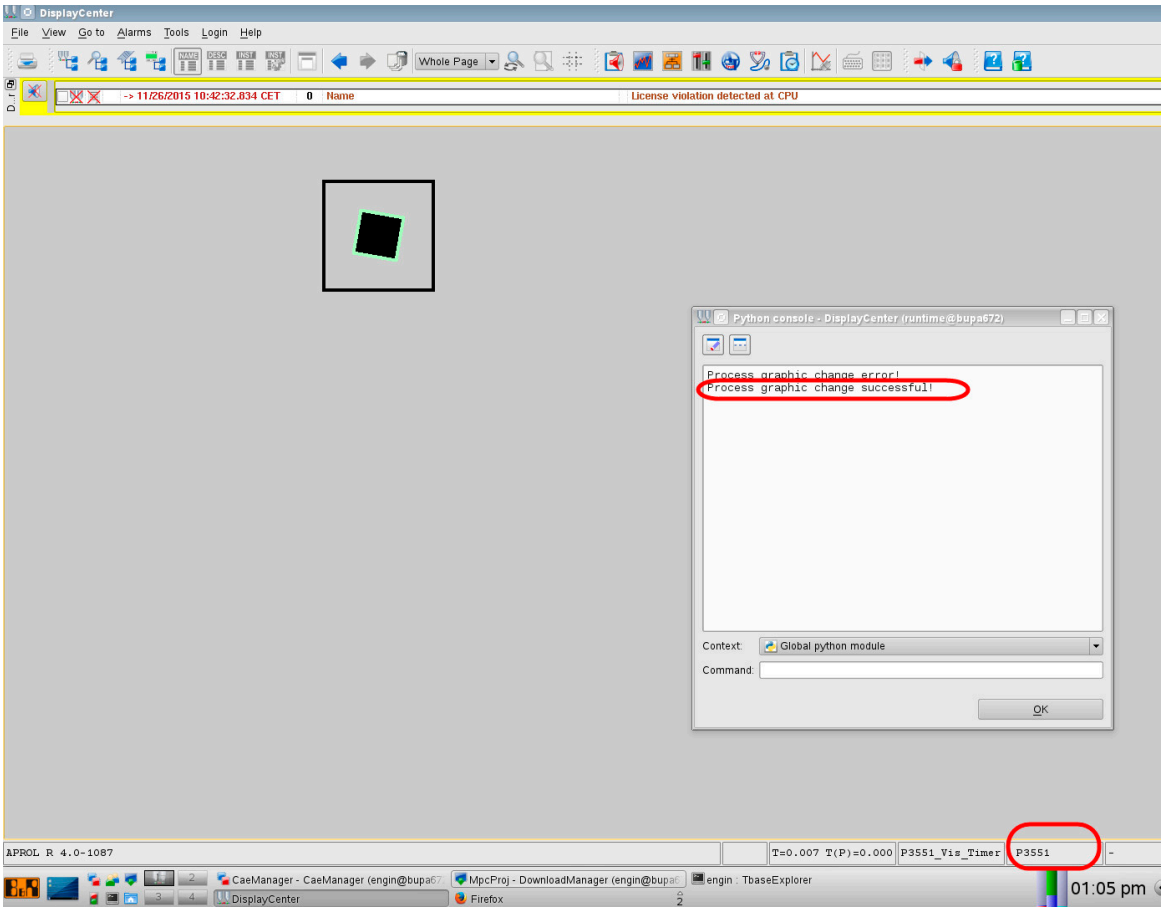


Figure 28: Loading process graphic successful

3.7 APROL Python in the LoginServer

The login process can be influenced with the **Python API** for the **LoginServer** .

Areas of use

There is a special project part available for using the API. The global "**Python Module for the LoginServer**" is conceived to store a collection of Python code in one module. This module is then assigned to a runtime or operator system. This assignment is made in the APROL system project part (runtime or operator).

The desired Python module must be selected in the **"-pythonModule**" module in **"System services ->LoginServer"**.

Attention: Only Python modules of the type **"Python module for the LoginServer"** are shown.

A different **"Python module for the LoginServer"** can be used for each runtime or operator system. The module is loaded as soon as the LoginServer is started. The Python code remains active during the entire lifespan of the LoginServer.

API functionalities

The different methods, functions and objects which are supplied by the API are described in detail in **manual D2 System API**. The following excerpt should only give an overview of some possibilities:

- Show logged in users
- Show user stack
- Show login timestamp
- Log in a user
- Log a user out
- Log out all users
- Log in predicate function of a user. Access possibilities before the user is logged in.
- Log out predicate function of a user. Access possibilities before the user is logged out.
- Log out predicate function of all users. Access possibilities before all users are logged out.

The predicate functions mentioned allow intervention when an attempt is made to log in or out. The **login/logout attempt** can be checked at this point and rejected or allowed.

Apart from the **APROL Python API functions**, the extensive functions in the **standard Python libraries** are always available.
This means that you can import the **"time"** module into your code at any time, for example, and use the **"time.ctime()"** function to read out the current date and time of your system.

Application examples

The following applications can be realized by using the listed functionalities:

- **Login/Logout** of a user depending on events (or via timer, etc.)
- **Customer application exclusive access:** There is always at least one default user logged in which as minimum rights (read rights). A maximum of one other user can also be logged in (query of the entries in the user stack). If another user should be logged in, the old user is logged out first and the new logged in, so that only one user is logged in at a time **(using the predicate function)**. An exclusive access for the user which is logged in can be realized in this way.

Events (Hooks)

All hooks are described in detail in the **APROL product documentation manual 'D2 System API'**. The following is an excerpt of some important **hooks** in the context of the **LoginServer**.

Hook	Event description
LS.init_hook	LoginServer was started.
LS.user_changed_hook	A new user has logged in or the old user was logged out.

Table 12: LoginServer Hooks

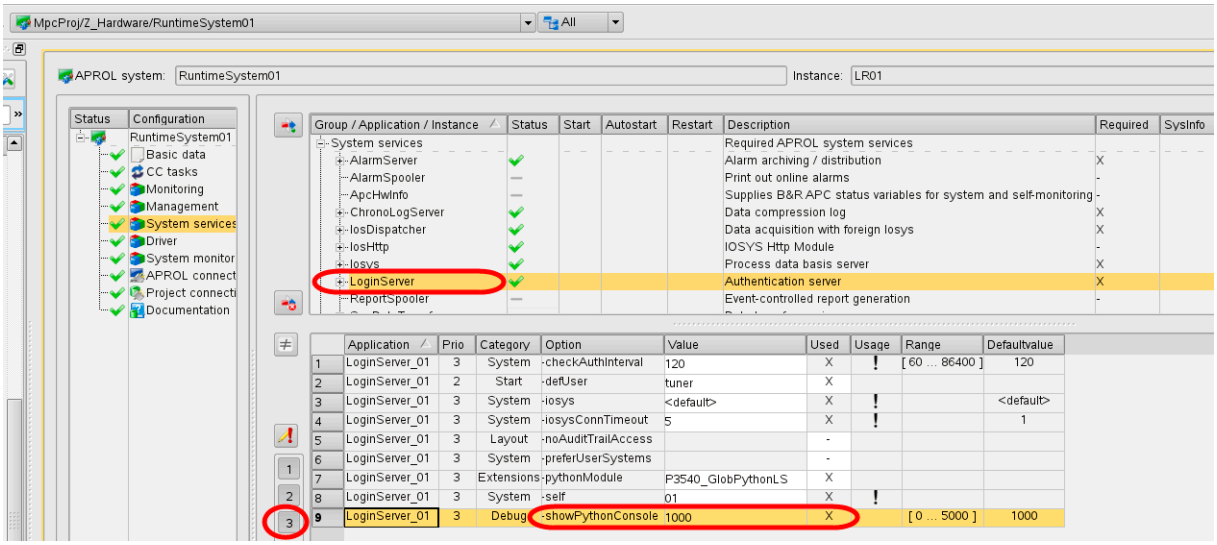
Debugging

Similar to the debug console in the context of the DisplayCenter, there is also a debug console for the LoginServer. For this purpose, the "**-showPythonConsole**" option must be used in the runtime system, in **"System services->LoginServer"**. The integer value which is specified defines the number of lines to be shown in the debug console. Attention: the setting can only be seen if the **priority 3** options are made visible.

When the Python console is activated, all of the Python code standard outputs are diverted to the Python console. print commands can be added to the code, in order to obtain a debug output.

If an error occurs in the Python code, the Python console will be opened, independent of if it was activated or not.

In contrast to the DisplayCenter console, the LoginServer does not have an interactive console. Entries are not possible in this console.



After the debugging has been switched on and the changes have been transferred to the runtime system, the console can be opened. For this purpose, there is a context menu for the **runtime login icon** in the system tray (key with cog wheel). The context menu has the **"Python Console"** entry to open the console.

3.7.1 Example task

Output name and login timestamp when changing operators

Each time an operator is logged in or out, the name of the operator which is currently logged in should be output on the Python console.

1) A **"Python control Module (LoginServer)"** project part is created in the context of a project.

2) We write the following code directly in the this project part:

```
import time

def myFunc1():
    print LS.user_stack[0].login()

    #Returns time as a REAL value
    timeReal=LS.user_stack[0].loginTime()
    #Convert to a struct_time
    timeStruct=time.localtime(timeReal)
    #convert struct_time to string
    print time.strftime("%a, %d, %b, %Y, %H:%M:%S ", timeStruct)
```

```
LS.add_hook(LS.user_changed_hook, myFunc1)
```

The **"time" module** is imported and the different functions for changing the time information can be used.

The **"myFunc1" function** prints out the operator which is currently logged in to the console with a print command. Access to the currently logged in operator takes place over the member variable (system variable) "LS.user_stack". The member variable depicts the a stack. This means that there may be more than one entry, i.e. one for each user which is logged in. The **uppermost user in the stack** is the **active user**, and can always be accessed with **index 0**. The **"login"** method delivers the name of the user which is logged in.

The second part of the function reads out the timestamp of the login with the **"loginTime"** method. This is how you can detect the time when an operator logged in. The **"loginTime"** method delivers a float value. The **"time"** module functions are necessary to convert the timestamp into a readable format.

The **"myFunc1"** user action is hung onto the **"user_changed_hook"** of the LoginServer. The function is then called every time a user is logged in or out.

3) The **"Python Control Module (LoginServer)"** is ready and can be compiled.

4) This Python module must be assigned to the LoginServer and runtime system.

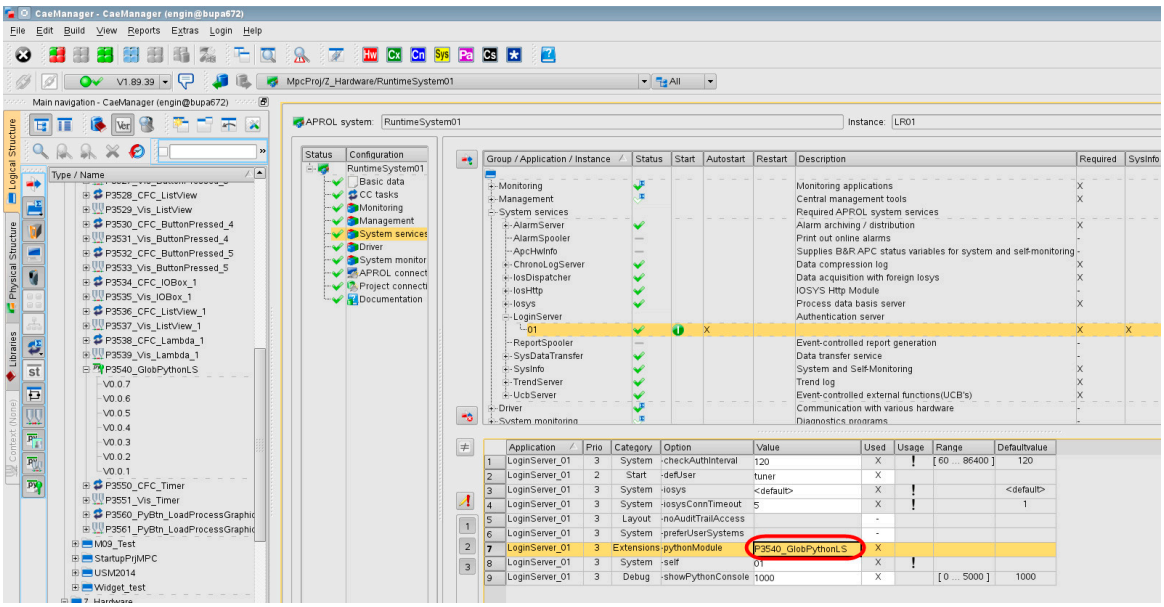


Figure 29: Assigning the Python module to the runtime system

5) This console must be activated in order to see the output on the Python console of the LoginServer. This is also done in the LoginServer options in the runtime system hardware.

6) A build of the project can then be carried out. A download to the runtime system is necessary.

7) If the LoginServer Python console has been activated for the first time, the LoginServer must be restarted to adopt the changes. You can log yourself out and back into the runtime system to do this. This leads to a restart of the runtime system LoginServer.

8) The LoginServer Python console can be opened using the context menu for the runtime login (key icon with cog wheel) in the systray.

9) Each time an operator is logged in or out, the name of the operator which is currently logged in and the login timestamp should be output on the Python console.

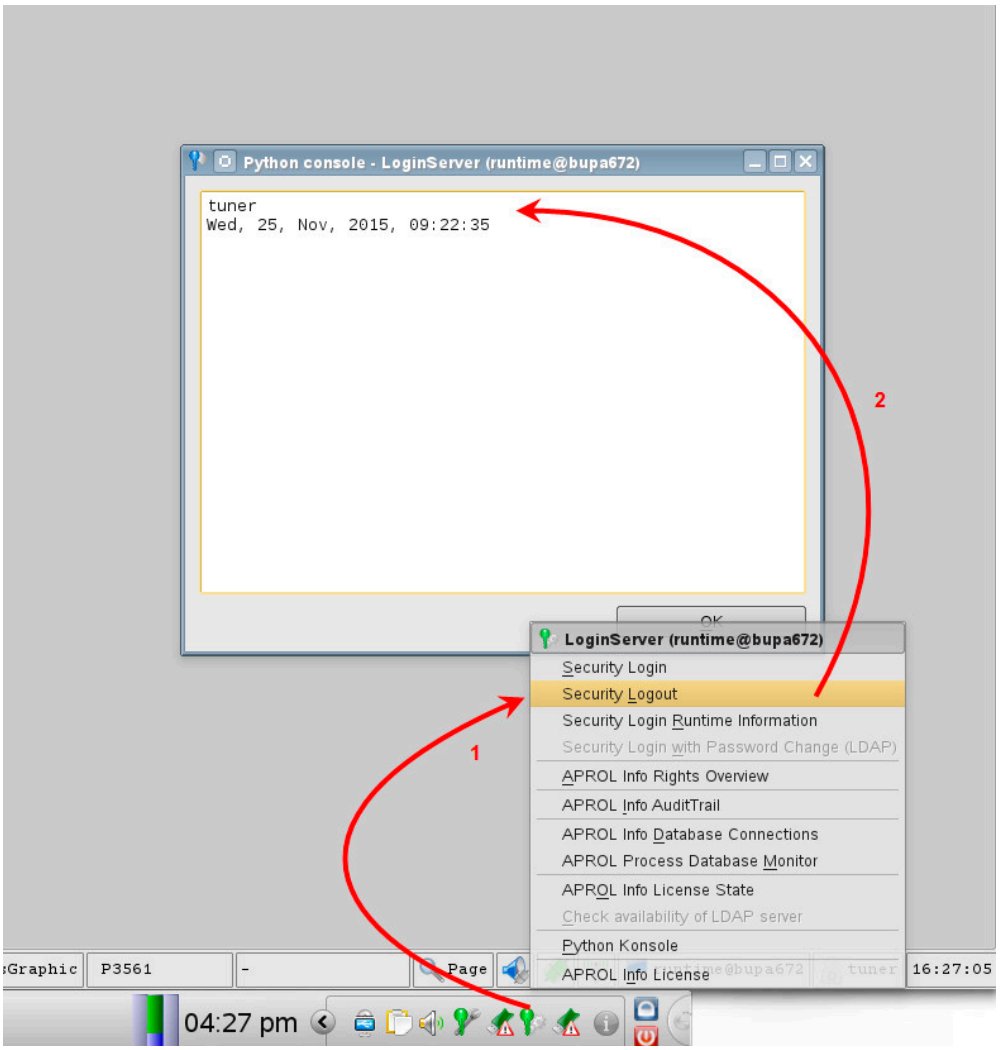


Figure 30: Output on the console

3.8 APROL Python in UCB blocks

The **UCB mechanism** offers you an **interface** between **APROL** and **operating system functionalities**. With it you have the possibility of automatically reacting to any system state, and to transfer information that has been won back into the process control system in order for it to be processed further.

Note:

Different languages can be used in the scripts. Python is only one example.

Areas of use

The **"UCB block"** is available for use in the context of a CAE library:

The specific rules for creating a UCB block can be found in the APROL product documentation **B3**.

The block which was created in the library must then be placed in a project. A Boolean signal is connected to the **"UcbTrigger"** input pin. The code in the UCB block will be executed each time there is a rising edge on the input pin. The communication between the block input pin and the code is over **environment variables**. The communication between the code and output pins is over **"print" statements**.

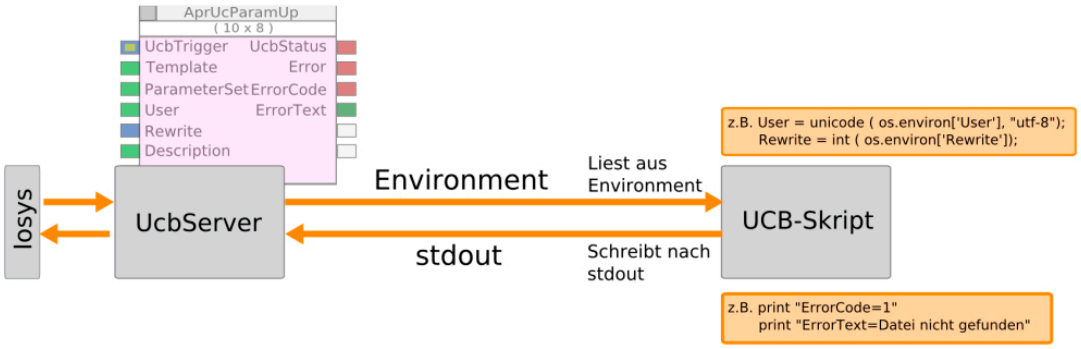


Figure 31: UCB structure

Application examples

Any number of scripts can be executed.

Examples:

- ChronoLog data query
- Execute backup script
- Check Ethernet link status
- Check NTP synchronization status
- Show the server's uptime
- Show which Linux users are logged in

Information about UCB block creation

There are important information about creating UCB blocks in **B3** , in the **APROL product documentation**.

We would like to especially point out the code sections for handling signals, initialization, and de-initialization. The UCB scripts are launched with the corresponding parameters while starting or stopping the UcbServer, a redundancy failover, or a download. This is why these situations should be handled properly in your code. The documentation is the basis for a Python UCB script.

It is greatly recommended to use the basis described in the documentation for your UCB scripts!

3.8.1 Example task

Checking the link status of an Ethernet interface

The name of an Ethernet interface which is being checked should be specified on an input pin.

The link status should be output on a pin after having been checked.

1) A new UCB block is created in the context of a CAE library.

2) The block automatically obtains a "UcbTrigger" input pin and a "UcbStatus" output pin.

3) An additional "DavName" input pin of the tpye "SSTRING" should be created for this application. This pin is for specifying which Ethernet interface should be checked. The pin obtains the default value "eth0".

4) A "LnkStat" output pin of the type "BOOL" is created.
The current LinkStatus should be on this pin.
Status 1 means that the link is okay.
Status 0 means that the link is not okay.

5) A "LoopCnt" output pin of the type "UDINT" is created. This variable is incremented each time the UCB script is executed. The user can read how often the script was executed with this variable.

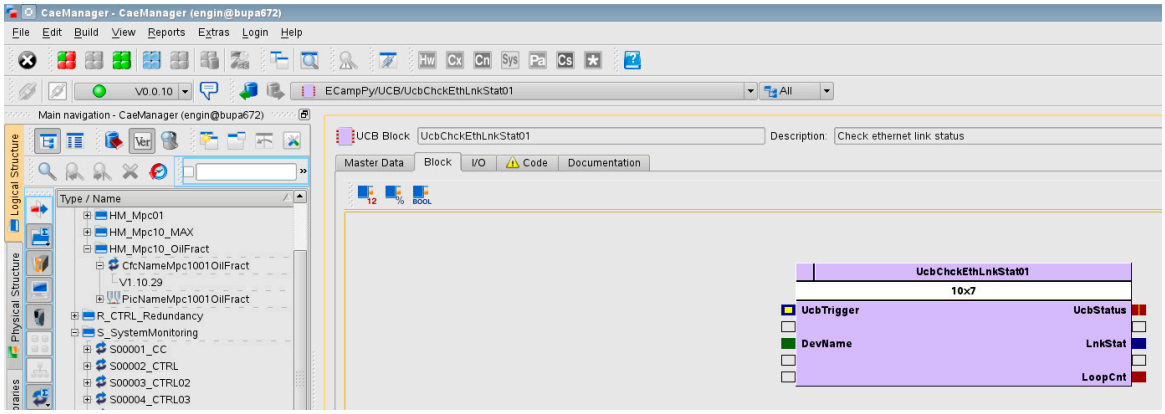


Figure 32: Create blocks and pins

6) Enter the Python code for compatibility which is listed at the end of the examples in the **"Code"** tab
The code consists of the following parts:

The specification of the **interpreter (Shebang)** and the line for the encoding are at the beginning of the file.

The **import instructions** for the module which are to be used follow.

The parts for **signal handling,** initialization, and de-initialization from the templates are used for a UCB script.

One section reads the values from the input pins into the Python script. The script can read these values using **environment variables** . The **"os.environ"** method of the **"os"** module is used for this. The value of the "DevName" input pin is read in the example, and contains the interface which should be checked.

The code part which a **Linux shell command is executed** follows after that. The Linux command is executed in a **subprocess**. This is necessary because of two reasons:
(a) When the Linux shell command was executed directly with "os.system()", for example, it would not be possible to control the duration of execution. This means that if the call took a long time, our UCB script would be block during this period of time. There may even be a continuous loop.
(b) Furthermore, the return value of the Linux shell command cannot be grasped with "os.system()".
The solution for both problems is the **"subprocess"** module. The commands are each executed as

a subprocess and the return values can be grasped. The code is also structured to check the runtime of the script. If there is still no return value from the Linux shell command after the maximum runtime has expired, the subprocess is terminated with a "kill". The return value then finally evaluated.

The Boolean result of the link status is transferred from the Python script to the APROL UCB block on the **standard output (STDOUT)**.

An execution counter can also be implemented. The value of the "LoopCnt" output pin is read first before each run. This value is then incremented and the new value is then written to the output pin.

Code for the example

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import os
import sys
import string
import signal
import subprocess
import time

#########################################################################
###    Signal handling (caused e.g. by RuntimeSystem Download)
#########################################################################
def on_term(signum, frame):
    sys.exit( 1 )
def on_usr(signum, frame):
    pass

signal.signal(signal.SIGTERM, on_term)
signal.signal(signal.SIGUSR2, on_usr)
signal.signal(signal.SIGUSR1, on_usr)


#########################################################################
###    Initialization phase (script called by UCB Server start)
#########################################################################
INSTANZ = sys.argv[ 0 ]
if len(sys.argv)>2 and sys.argv[2] == '-init' :
    sys.exit(0)

#########################################################################
###    De-Initialization phase (script called by UCB Server stop)
#########################################################################
if len(sys.argv)>2 and sys.argv[2] == '-deinit' :
#########################################################################
###    Forced termination phase (script called by forced UCB Server termination)
#########################################################################
    if len(sys.argv)>3 and sys.argv[3]  == '-forced' :
```

```
        if len(sys.argv)>4 :
            ListOfTerminatedInstances=string.split( sys.argv[4] , ',' )
            for inst in ListOfTerminatedInstances:
                # cleanup actions for terminated instances possible here
                pass
    sys.exit (0)


#########################################################################
###    Read inputs 8environment variables)
#########################################################################
locDevName = unicode( os.environ['DevName'], "utf-8" )


#########################################################################
###    Linux commands
#########################################################################
LinuxCmd1="/sbin/ifplugstatus"


#########################################################################
###    System call
#########################################################################
ReturnCode1=-1

RunTimeStart = time.time()    # save start timestamp
RunTm=-1
TmOut=10 # timeout in seconds

#call linux command via subprocess
Return1 = subprocess.Popen( [LinuxCmd1,locDevName], shell = False, \
    stdin = subprocess.PIPE, stdout = subprocess.PIPE, \
    stderr = subprocess.PIPE )

#check runtime of linux command
while ( Return1.poll() == None ):
    RunTm = time.time() - RunTimeStart
    #timeout exceeded
    if RunTm >= TmOut:
        # Command will be killed
        Return1.kill()
        Return1.communicate() #intercommunication scipt<->subprocess
        break
    time.sleep( 0.5 )

ReturnCode1=Return1.poll()

#evaluate return code: 2 means link ok
if (int(ReturnCode1)==2):
    locRetVal=1
else:
    locRetVal=0
```

```
#########################################################################
### Write Output-Pins
#########################################################################
print "LnkStat="+str(locRetVal)


#########################################################################
###    Check how often UCB-script was run
#########################################################################
LoopCnt = int( os.environ['LoopCnt'] )         # read variable
LoopCnt = LoopCnt + 1                         # increment variable
print "LoopCnt=" + str( LoopCnt )             # write variable
```

3.9 APROL Python for parameter management

The APROL **parameter management** component offers the possibility to connect a parameter and recipe management in projects. The Python API offers a comprehensive way to manage parameters and recipes.

Areas of use

The parameter management API can be used in two different areas:

- In the scope of the DisplayCenter. This means that the API is either used in a **"process graphic",** a **"graphic macro",** or **"Faceplate"**. The processing of the API functions is asynchronous. Several functions can be called "simultaneously". When calling several functions 'simultaneously', the order in which the callback functions are triggered is completely independent of the calling order. The name of the current operator is detected.

- The API can also be used in the context of a **"UCB block"**. This sort of usage is realized in the **"APROL"** library by the **ParameterManagement** block group. The processing of an API function is carried out synchronously, therefore functions can only be called strictly one after the other. The name of the4 operator must be explicitly set with the PM.set_user_name(...) function

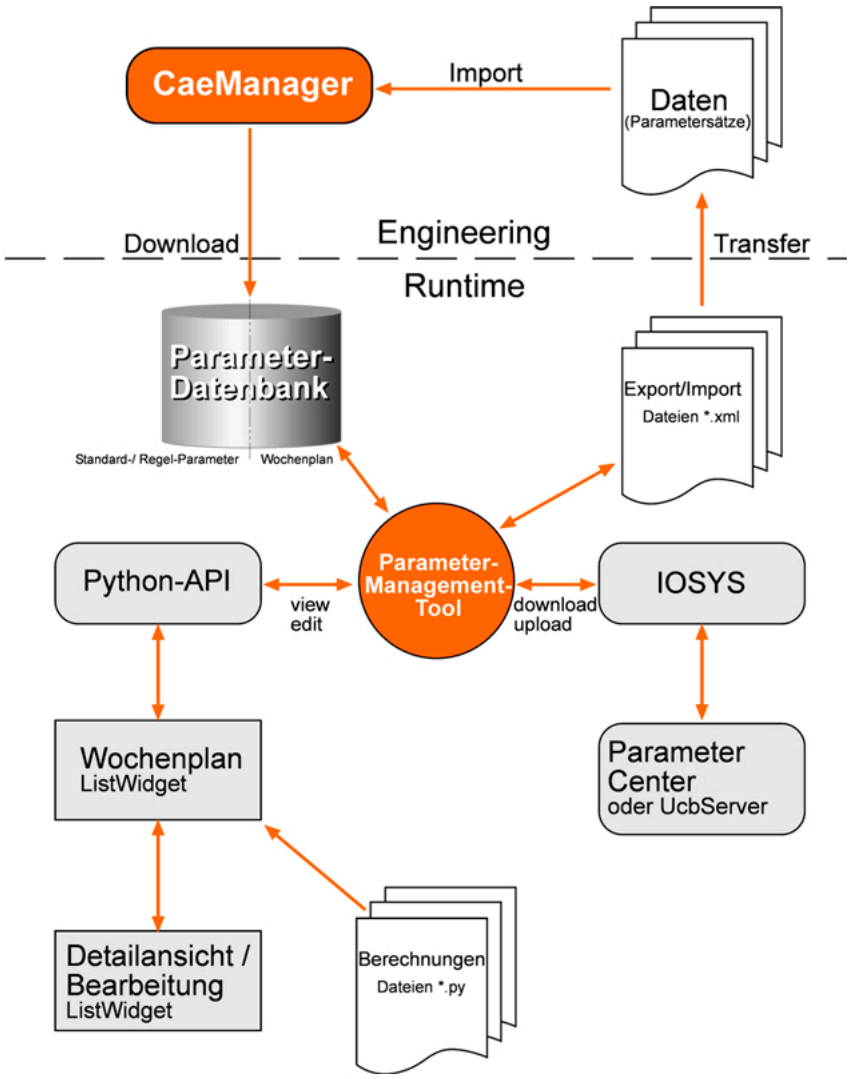


Figure 33: Parameter management structure

API functionalities

The different methods, functions and objects which are supplied by the API are described in detail in **manual D2 System API**.

All function calls depend on the following import statements:

```
import APROL.ParamMgmt as PM
```

The following excerpt should only give an overview of some possibilities:

Function	Description
PM.get_templates	Read one or more parameter set templates.
PM.get_template_params	Read the parameter (attribute) of a parameter set template.
PM.get_paramsets	Read one or more parameter sets from a parameter set template.
PM.get_paramset_params	Read the parameter data of a parameter set.

Table 13: Parameter management API

Function	Description
PM.create_paramsets	Creates one or more new parameter sets in the database.
PM.update_paramsets	Changes one or more parameter sets in the database.
PM.update_param	Changes the value of a single parameter in the database.
PM.delete_paramsets	Deletes one or more parameter sets in the database.
PM.exists_template	Check for the existence of a template name.
PM.exists_paramset	Check for the existence of a parameter set.
PM.download_paramset	Writes the value of a parameter set in the runtime variables.
PM.upload_paramset	reads the runtime value of a parameter set and stores this in the database.
PM.export_paramsets	Exports one or more parameter sets in XML format into a file.
PM.import_paramsets	imports one or more parameter sets in XML format into the database.
PM.rename_paramset	Renaming a parameter set.
PM.set_paramset_description	Setting the parameter set description of a parameter set.
PM.get_user_name	Only in UCB blocks: Delivers the set operator names.
PM.set_user_name	Only in UCB blocks: Sets operator names.

Table 13: Parameter management API

Application examples

The following applications can be realized by using the listed functionalities:

- Complete handling of the parameter management
- There are predefined UCB blocks for parameter handling in the **"APROL"** system library. (read, delete, upload, download, etc. parameter sets)

Events (Hooks)

DC hooks can be used in the context of the DisplayCenter, to hang the parameter management API on the return functions.

The script execution of a UCB block is started with the UcbTrigger. The API functions are then called in the code. The API functions are called in a serial manner in this case.

3.9.1 Example 1 - UCB

Reading a list of existing parameter sets (use of AprUcListParam)

The instance of one parameter set suffices as input value. The parameter sets which exist for this parameter set template should be read out.

This task should be realized with a UCB block supplied by the APROL system. The **"AprUcListParam"** block from the **"APROL"** library should be used.

An **equipment** and **parameter set template** must already exist in the project in order to be able to complete this task. Here is a brief summary of the necessary steps in case this has not already been prepared.

- The parameter management must be activated when the APROL runtime system is configured with AprolConfig.
- The ParameterCenter should be activated in the monitoring section of runtime system in the project, so that the ParameterCenter can be used at a later stage.
- Create equipment, define parameters in the equipment.
- Use the variables defined in the equipment (parameters) in the function charts (CFCs).
- Create parameter set template, insert parameters from the equipment.
- Create a parameter set on the basis of the parameter set template.
- All created and modified project parts must be saved, activated and compiled.
- Perform the build for the affected task. Download to the affected target systems

Use of the **AprUcListParam block** is as follows:

1) Creating a new CFC.

2) Place an instance of the "AprUcListParam" block from the APROL library in the CFC.

3) A trigger signal must be connected to the "UcbTrigger" input of the block. The UCB block will be executed if there is a positive edge on the trigger signal. The trigger signal can be generated by a pulse button, for example. A block which can be used immediately for this purpose would be the "Pulse" block from the "PB_Bas" library.

4) The parameter set template instance is specified in the "Templateinstance" input of the UCB block. The parameter set template instance which was specified during the creation must be entered here.

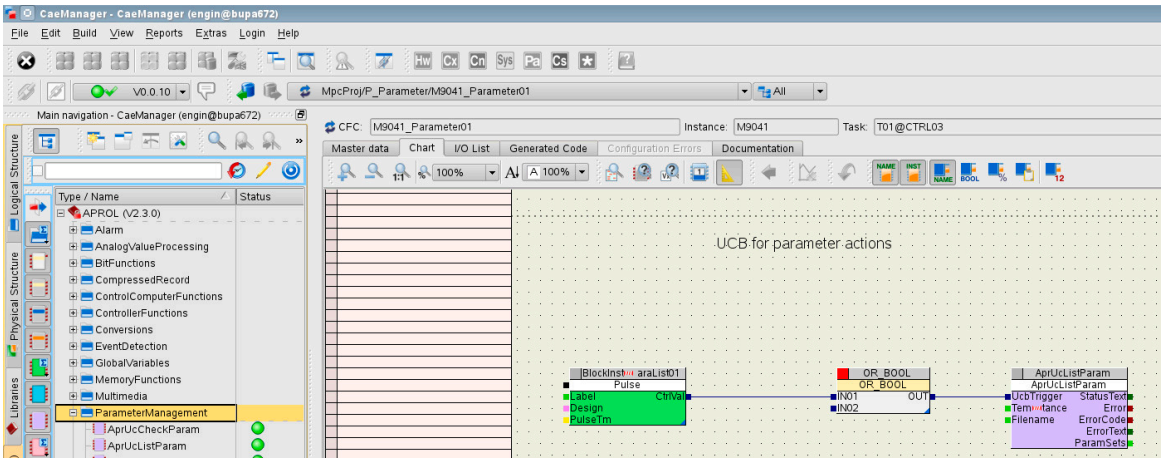


Figure 34: Parameter management with UCB blocks: CFC

5) Save and compile the CFC

6) If a button is used to create the trigger signal, an instance of the button must be placed in a process graphic. The process graphic must also be saved and compiled.

7) Build all necessary project parts.

8) Download to the necessary target systems

9) Triggering is possible with the button in the process graphic, in the DisplayCenter. If the block could be executed successfully, a list with all of the available parameter sets will be visible in debugging on the "ParamSets" output.

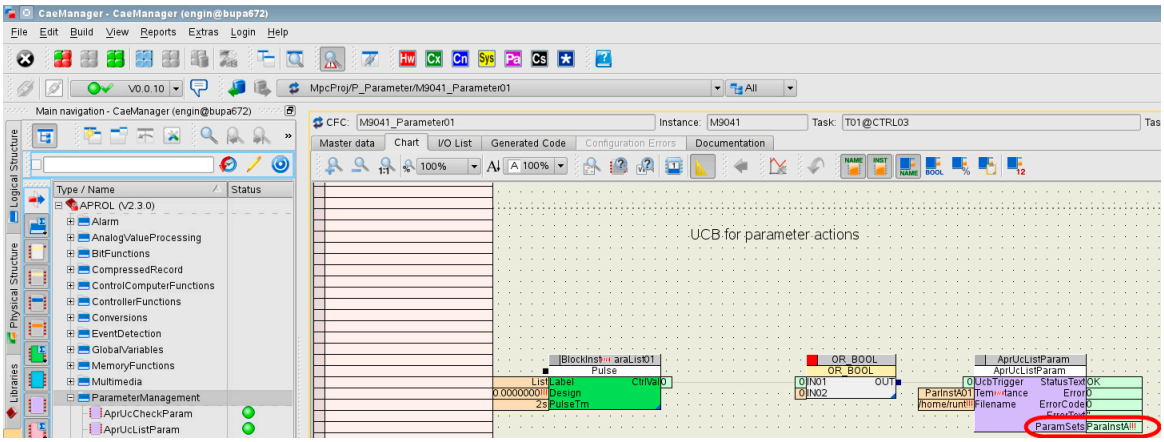


Figure 35: Parameter management with UCB: Result

3.9.2 Example 2 - Graphic macro

Display of all parameter values of a parameter set in a ListView widget

A graphic macro should be created for this output. The graphic macro should contain a Python button. The parameter values should be read out when the button is pressed. The DisplayCenter API for parameter management should be used to read out the values. The results should be displayed in a **ListView widget**.

The instance of a parameter set template and the name of a respective parameter set suffices as input value. The values and available properties of the parameter should be read out and shown in a ListView widget.

An **equipment** and **parameter set template** must already exist in the project in order to be able to complete this task. Here is a brief summary of the necessary steps in case this has not already been prepared.

- The parameter management must be activated when the APROL runtime system is configured with AprolConfig.
- The ParameterCenter should be activated in the monitoring section of runtime system in the project, so that the ParameterCenter can be used at a later stage.
- Create equipment, define parameters in the equipment.
- Use the variables defined in the equipment (parameters) in the function charts (CFCs).
- Create parameter set template, insert parameters from the equipment.
- Create a parameter set on the basis of the parameter set template.
- All created and modified project parts must be saved, activated and compiled.
- Perform the build for the affected task. Download to the affected target systems

The following steps show how to complete the example:

1) A new **graphic macro** is created in the context of a CAE library.

2) Two inputs of the type LSTRING are created.
The "ParTmplInst" input is used to adopt a parameter set instance.
The "ParSet" input is used to specify a parameter set name.

3) Two outputs of the type INT are created.
The "ErrorCode" output is used to output an error number.
An extended error number which depends on the error code is output on the "ErrorSubcode" output.

4) The graphic part of a graphic macro is composed of a Python button and a ListView.

5) The name of a local variable must be entered in the graphic I/O of the PythonButton. The name can normally be chosen freely. The button can be identified with this name in the Python code. The name "locBtn" should be used in the example.

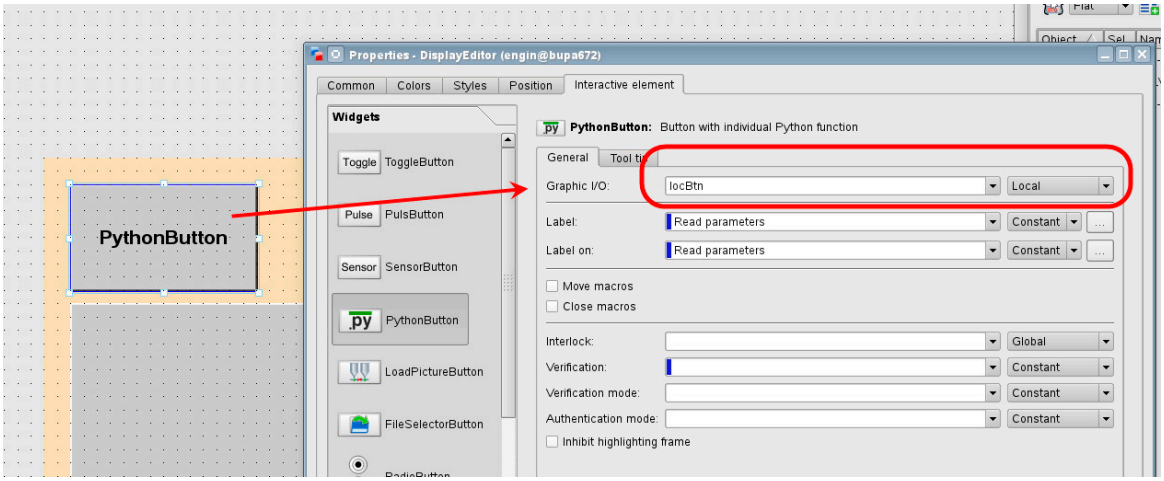


Figure 36: Python button properties

6) It is important to enter **object names** in the properties of the ListView. The ListView is identified with this name in the Python code. The **object name "LV"** should be used in the example.

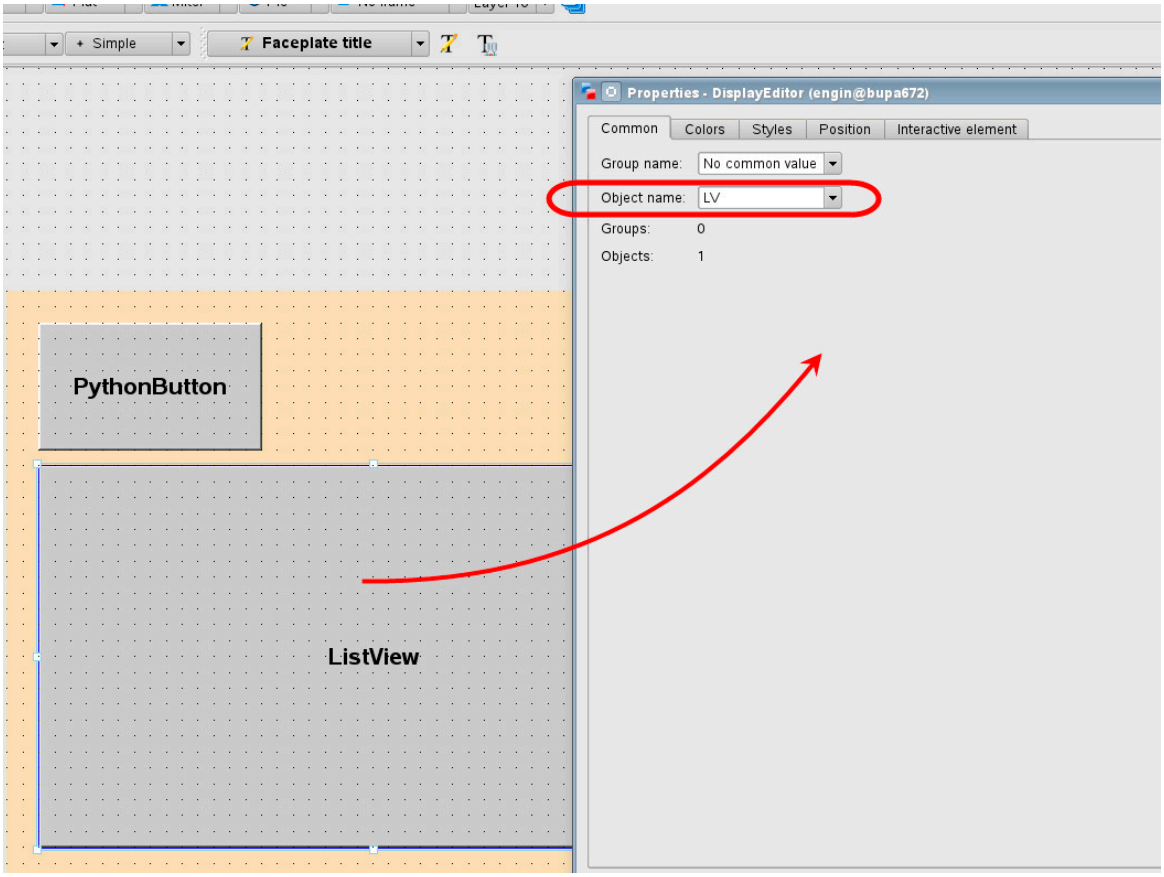


Figure 37: ListView properties: Object name

7) Furthermore, a local variable named "locSel" must be entered in the properties of the ListView, in the **"ControlValue"**.

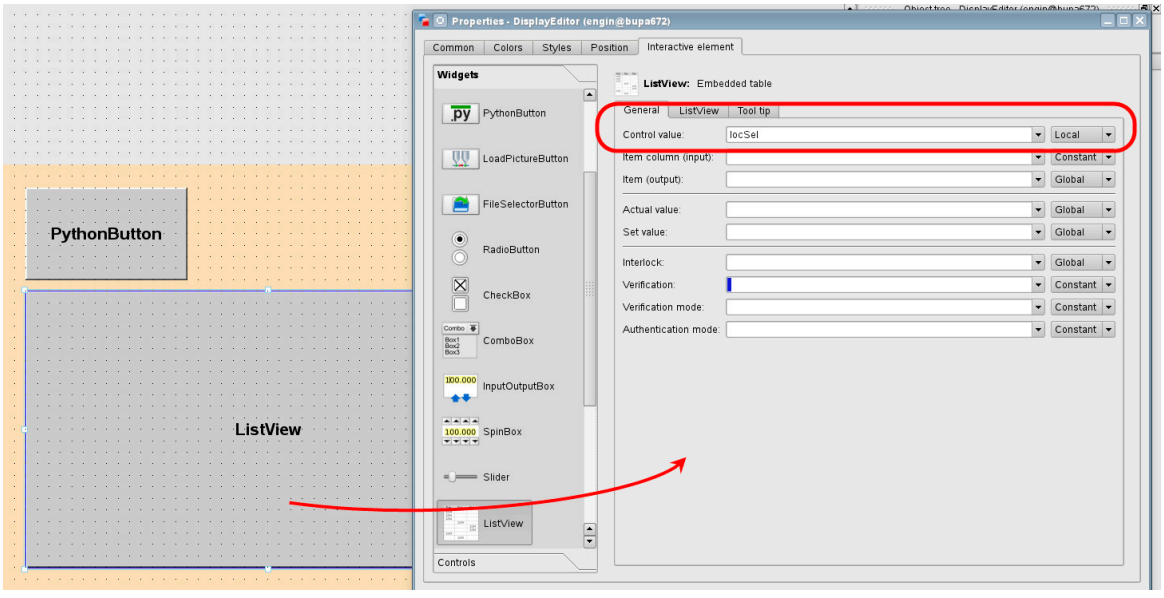


Figure 38: ListView properties: Control value

8) Python code is then required. The Python code used is listed at the end of the examples for compatibility. There is also a description of the individual code blocks there.

9) The graphic macro must then be compiled.

10) A new CFC is then created in the context of a CAE project.

11) One instance of the newly created graphic macro is inserted into the CFC.

12) I/O boxes can be connected to the "ParTmplInst" and "ParSet" input pins, in order to allow their values to be variable. It is then possible to change the strings for the parameter set template instances and parameter sets in the DisplayCenter at any time. A block which can be used immediately for this purpose would be the "IOBoxSSTRING03" block from the "PB_Bas" library.

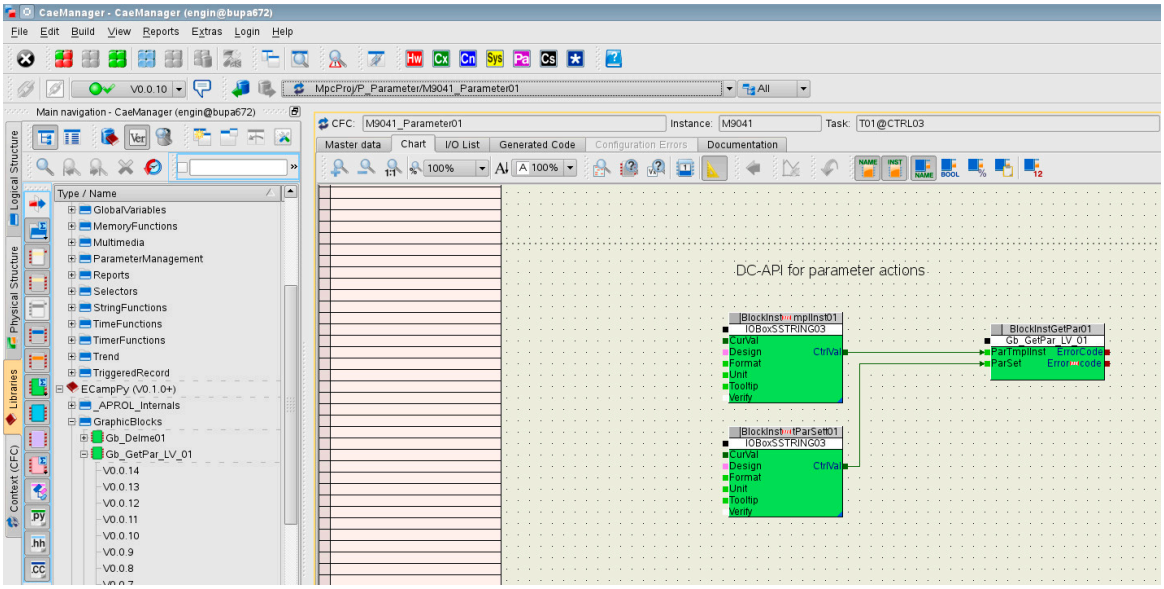


Figure 39: Graphic macro nesting in the CFC

13) The CFC must then be saved and compiled.

14) A new process graphic is then created. An instance of the graphic macro is inserted in the process graphic. And also an instance of both I/O boxes. The process graphic is also saved and compiled.

15) A subsequent build of the required parts in the project is necessary.

16) A download is then made to the target systems.

17) It is then necessary to set the parameter set templates and parameter sets to sensible values in the DisplayCenter. If the Python button is pressed, this triggers the reading of the parameter sets. If this is possible without an error, the resulting data will be shown in the ListView.

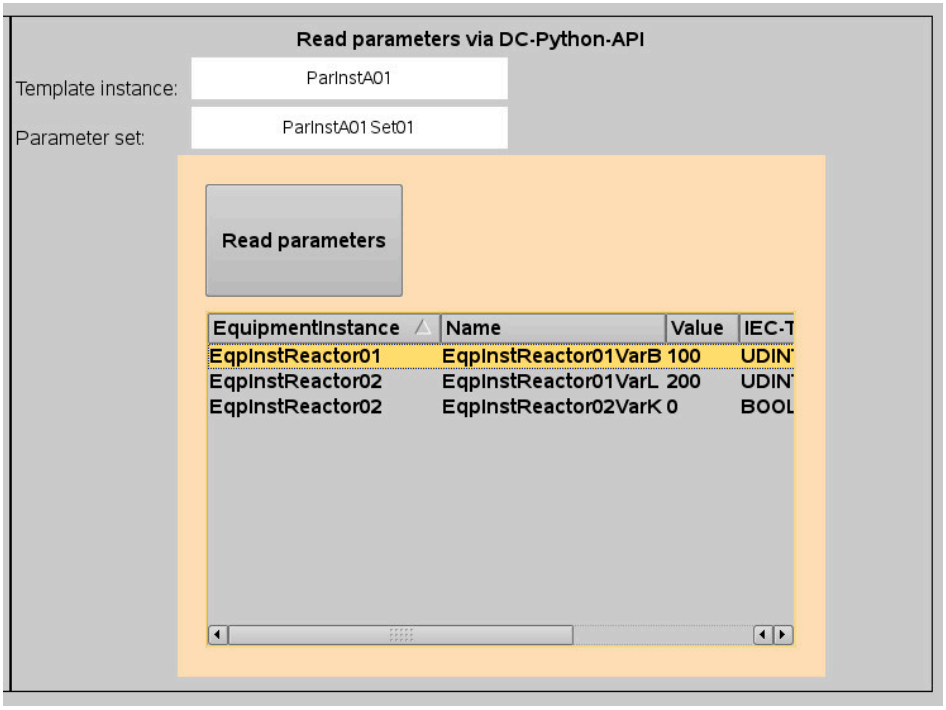


Figure 40: Display of the results in the ListView

Python code

```
# -*- coding: utf-8 -*-

#pylint: disable=E1101
## E1101: Module 'APROL.ParamMgmt' has no 'ErrorResponse' member

#Import of Python API for APROL Parameter Management
#for usage with module name PM
import APROL.ParamMgmt as PM

LastResultParamset = None
LastResultTemplate = None

block.local.locSel.set(0)    #selected item


#Callback function for get_template_params
def cbTemplateParams(result):
    global LastResultTemplate
    LastResultTemplate = result
    if isinstance(result, PM.ErrorResponse):
        block.pin.ErrorCode.set(result.error_code())
        block.pin.ErrorSubcode.set(result.error_subcode())
```

```
    else :
        show_list()
        block.pin.ErrorCode.set(0)
        block.pin.ErrorSubcode.set(0)


#Callback function for get_paramset_params
def cbParamsetParams(result):
    global LastResultParamset
    LastResultParamset = result
    if isinstance(result, PM.ErrorResponse):
        block.pin.ErrorCode.set(result.error_code())
        block.pin.ErrorSubcode.set(result.error_subcode())
    else :
        show_list()
        block.pin.ErrorCode.set(0)
        block.pin.ErrorSubcode.set(0)


#Build up content for ListView-widget
def show_list():
#If both results (1-from PM.get_template_params;
#    2-from PM.get_paramset_params) are ready
#Both results are delivered from callback functions;
#    they are processed asynchronously
#Both results will be ready when the second callback
#    (the slower one) is finished
#When the 1st callback function calls show_list
#    ->goto else-branch (only one ready)
#When the 2nd callback function calls show_list
#    ->build up list (both ready)
    if (    type(LastResultParamset) is PM.ParamsetParamList ) and \
        (     type(LastResultTemplate) is PM.TemplateParamList ):
        block.widget.LV.clear()
        line = 0
        for i in LastResultParamset:
            for j in LastResultTemplate:
                if (  i.name == j.name and \
                i.equipment_instance == j.equipment_instance ):
                    item = [     i.equipment_instance, \
                                 i.name, \
                                 i.value, \
                                 j.iec_type, \
                                 j.default_value, \
                                 j.min_value, \
                                 j.max_value, \
                                 str(j.fix)]
                    block.widget.LV.set_item( line, item )
                    line += 1
    else:
        #Only one of the two results is ready - do nothing;
        #The ListView will be built up in the next show-list-call
```

```
        pass


#Read parameters from database
def myGetParaFu():
    if ( block.pin.ParTmplInst.get() and block.pin.ParSet.get() ):
        global LastResultParamset, LastResultTemplate
        LastResultParamset = None
        LastResultTemplate = None
        #Read template parameters - use callback
        PM.get_template_params(    cbTemplateParams, \
                                   block.pin.ParTmplInst.get(), \
                                   run_inst = block.run_inst() )
        #Read parameterset parameters - use callback
        PM.get_paramset_params( cbParamsetParams, \
                                   block.pin.ParTmplInst.get(), \
                                   block.pin.ParSet.get(), \
                                   '.*', \
                                   '.*', \
                                   run_inst = block.run_inst() )


#Initialization
def blockInit():
    columns = [     "EquipmentInstance", "Name", "Value", "IEC-Typ", \
            "Default", "Min", "Max", "Fix" ]
    block.widget.LV.set_columns( columns )#Write the header of the ListView
    myGetParaFu()    #read parameters from database


#Initialization
DC.add_hook( block.init_hook, blockInit )

#Python button pressed
DC.add_hook( block.local.locBtn.button_press_hook, \
lambda var: myGetParaFu())
```

- The module for the APROL parameter management is imported first. It is then available with the name "PM" in the code.

- The example uses the global "LastResultTemplate" and "LastResultParameter" variables to manage the data for the parameter set.

- The **"PM.get_template_params"** and **"PM.get_paramset_params"** functions are used by the **Parameter-Management-API**. These, and all other parameter management API functions, work with so-called return functions. This means that one of this function's transfer parameters is a function in itself.
  The sequence starts by calling the actual API function. It then tries to read out the data asynchronously from the parameter management in the background. This may take a considerable amount of time. The API function calls the transferred function when the parameter management data is ready.
  Suitable return functions must therefore be defined for both API functions which are used in the example. Those are the functions which are executed when the parameter management data has arrived. Those are the "cbTemplateParams" and "cbParamsetParams" functions in the example.
  These functions are structured similarly: Both try to interpret the result of the API function. There are two possibilities for doing that. The call has caused an error if the API function result is of the type "PM.ErrorResponse". If that is not the case, the call was successful and the returned data can be processed. The actual processing of the data then takes place in the **"show_list" function**.

- The **"show_list" function** checks to see if the results of both API function calls exist. Because the API function calls are processed asynchronously, there are different times until both calls are finally ready. One is ready earlier, the other later. This is why the show_list call from the first return function always lands in the else branch of the large if query and does nothing. The return function which is ready in the second place makes **show_list** call the if block. both API calls are **"synchronized"** with this technique.
  The main code then combines the results from the parameter set template with the parameter set and creates a list. This list is then added to the ListView widget line by line using **"block.widget.LV.set_item()"**.

- The call for the API functions is stored in the "myGetParaFu" function. The global LastResultTemplate and "LastResultParameter" variables are initialized with "None" before the call.

- The **"blockInit"** function creates the header of the ListView. Apart from that, a first attempt is made to read out the data from the parameter management. This function is hung onto the **block.init_hook** .

- The **block.local.locBtn.button_press_hook** of the Python button ensures that the "myGetParaFu" function is executed each time the button is pressed, and therefore that the data is read out of the parameter management.

4 Summary

A basic insight into the Python programming language has been given. The basics were enforced by many practical examples.

There are different areas of application for the Python code in APROL. The most important API functions for each of these application areas were shown and typical application examples were named. Depending on the use, the Python code is also executed upon different results. The most important events were listed and extensive examples were given for each application area.

You should also be able to use this training manual for referencing detailed information at a later stage.

With this information, you can use the APROL Python API to realize your own ideas.

5 Appendix

5.1 Objectives checklist (no questions about the topic)

5.2 Solutions to the objectives checklist

Please take note of the necessary additional information.

Appendix

Appendix

Appendix