Joint Institute for Nuclear Research
SCIENCE BRINGING NATIONS TOGETHER

Python useful features for experiment control systems programming

FLNP

Frank Laboratory of Neutron Physics, Joint Institute for Nuclear Research, 141980 Dubna, Moscow Region, Russia.
E-mails: akirilov@nf.jinr.ru, akirilov@jinr.ru

Python programming language is popular, in particular for the programming and control systems as a basic programming language and as a language for programming user interfaces (GUI). In the programming complex Sonix+ the Python successfully used for both of these. The presentation will be devoted to some nice language features useful for programming instrument control software.

Use the Python introspection for the GUI universalization

There is a widget for manual device control in the Sonix+ GUI. This widget allows the User to select device from list and then select one of the available command for this device. It is important to emphasize that necessary information is obtained automatically so the widget code is instrument independent. In order to organize a unified device independent manual interface it is necessary to obtain list of available devices and a list of commands available for each device.

In the Sonix+ these information is concentrated in several Python files. The instrument configuration file contains full list off devices with appropriate names and IDs. Besides the are device description files to describe functionality of concrete device as a Python class. The necessary data is extracted directly from the Python environment.

This extraction is illustrated for the ***YuMO*** instrument as an example. At first import the configuration file and have a look at the object list

```
>>> import yumo_python_configuration as y
>>> dir(y)
['ASExecute', 'ASSignalize', 'CloseSession', 'CommError', 'CreateVarmanVariableBA', 'DeviceInfo',
'GetASResult', 'GetVersion', 'InitAsStruct', 'LoadDB', 'OpenSession', 'ReadAsString', 'ReadAsStruct',
'SExecute', 'SSignalize', 'SaveDB', 'SendCommand', 'WaitLocker', 'WriteAsString',
. . .
'vanady1_2det_soft', 'vanady2', 'vanady2_1det', 'vanady2_1det_soft', 'vanady2_2det',
'vanady2_2det_soft', 'vanady2_soft']
```

This list includes among others device names, which we need to select. It is easy because all device instances belong to ***device*** class.

```
>>> s = []
>>> for i in dir(y):
... d = getattr(y, i)
... if isinstance(d, y.device):
... s.append(i)
...
>>> s
['beam_shutter', 'chopper', 'collimator', 'configList',
. . .
'vanady1_2det_soft', 'vanady2', 'vanady2_1det', 'vanady2_1det_soft', 'vanady2_2det',
'vanady2_2det_soft', 'vanady2_soft']
>>>
```

This list s is the actual instrument device list. Next step is to obtain list of available commands (class methods) for each device. For instance lets consider device **g_table**. Get a reference to the component by its name.

```
>>> g = getattr(y, 'g_table')
```

Its directory

```
>>> dir(g)
['GetStateId', 'GetStateName', 'Set', 'SetByName', '__doc__', '__init__', '__module__', 'id', 'name',
'server_name']
```

Look name of the module (server) for the device. (It may be useful to check the mailbox in the database, i.e., that the module was really loaded, and it is possible to send a command).

```
>>> getattr(g, 'server_name')
'tabular_adapter'
```

Get a list of object's methods, i.e. a list of device commands

```
>>> import inspect
>>> inspect.getmembers(g, inspect.ismethod) ## not isfunction !!!
[('GetStateId', <bound method tabular.GetStateId of <adapters_def.tabular instance at 0x00F0FCD8>>),
('GetStateName', <bound method tabular.GetStateName of <adapters_def.tabular instance at 0x00F0FCD8>>),
('Set', <bound method tabular.Set of <adapters_def.tabular instance at 0x00F0FCD8>>), ('SetByName',
<bound method tabular.SetByName of <adapters_def.tabular instance at 0x00F0FCD8>>), ('__init__', <bound
method tabular.__init__ of <adapters_def.tabular instance at 0x00F0FCD8>>)]
>>> gsm = inspect.getmembers(g, inspect.ismethod)
>>> len(gsm)
5
```

Select one of the options, for example, ***Set*** command

```
>>> gsm[2][1]
<bound method tabular.Set of <adapters_def.tabular instance at 0x00F0FCD8>>
>>> f = gsm[2][1]
```

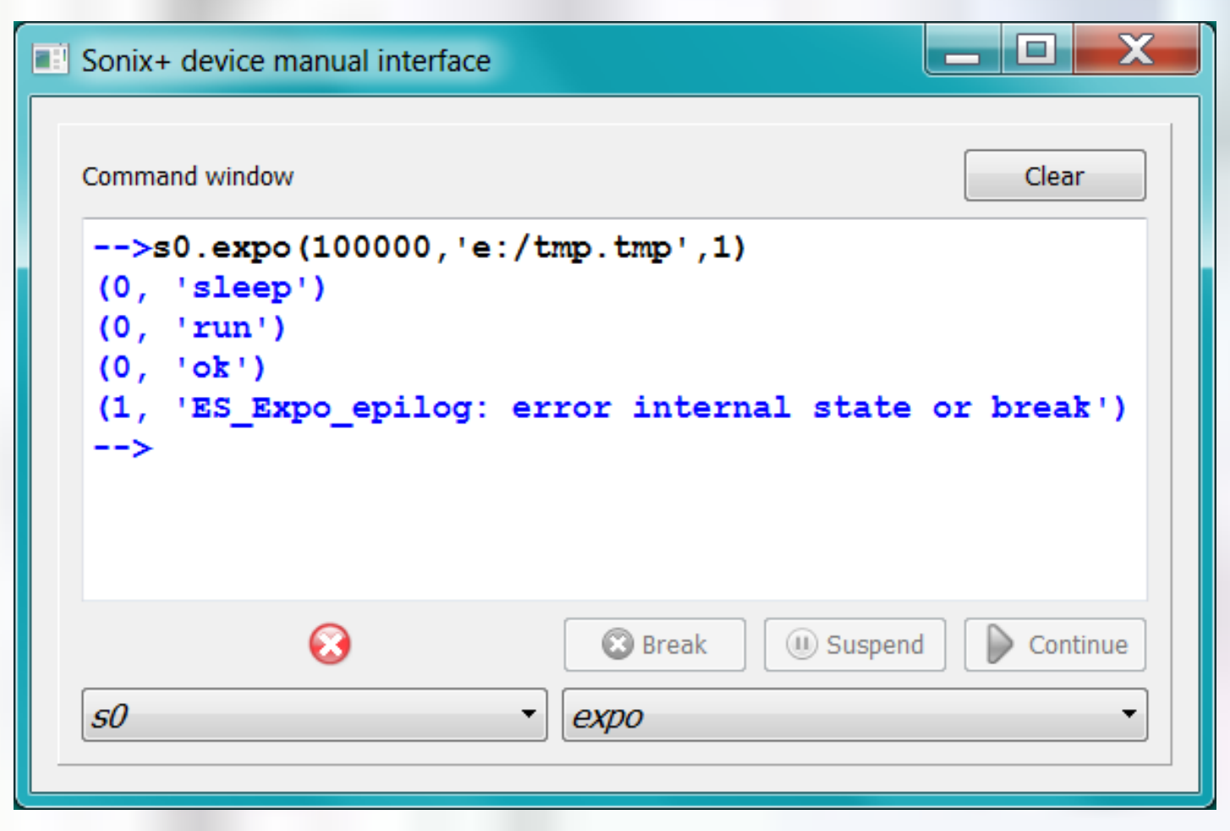
One can get it source

```
>>> lfs = inspect.getsourcelines(f)
>>> lfs[0]
['\tdef Set (self, state_id):\n', '\t\tif ( type(1) == type(state_id) ):\n', '\t\t\treturn
SExecute(self.server_name, 14000, str((self.id, state_id)))\n', "\t\treturn 'error state id'\n"]
```

and the first line (for the parameter prompts)

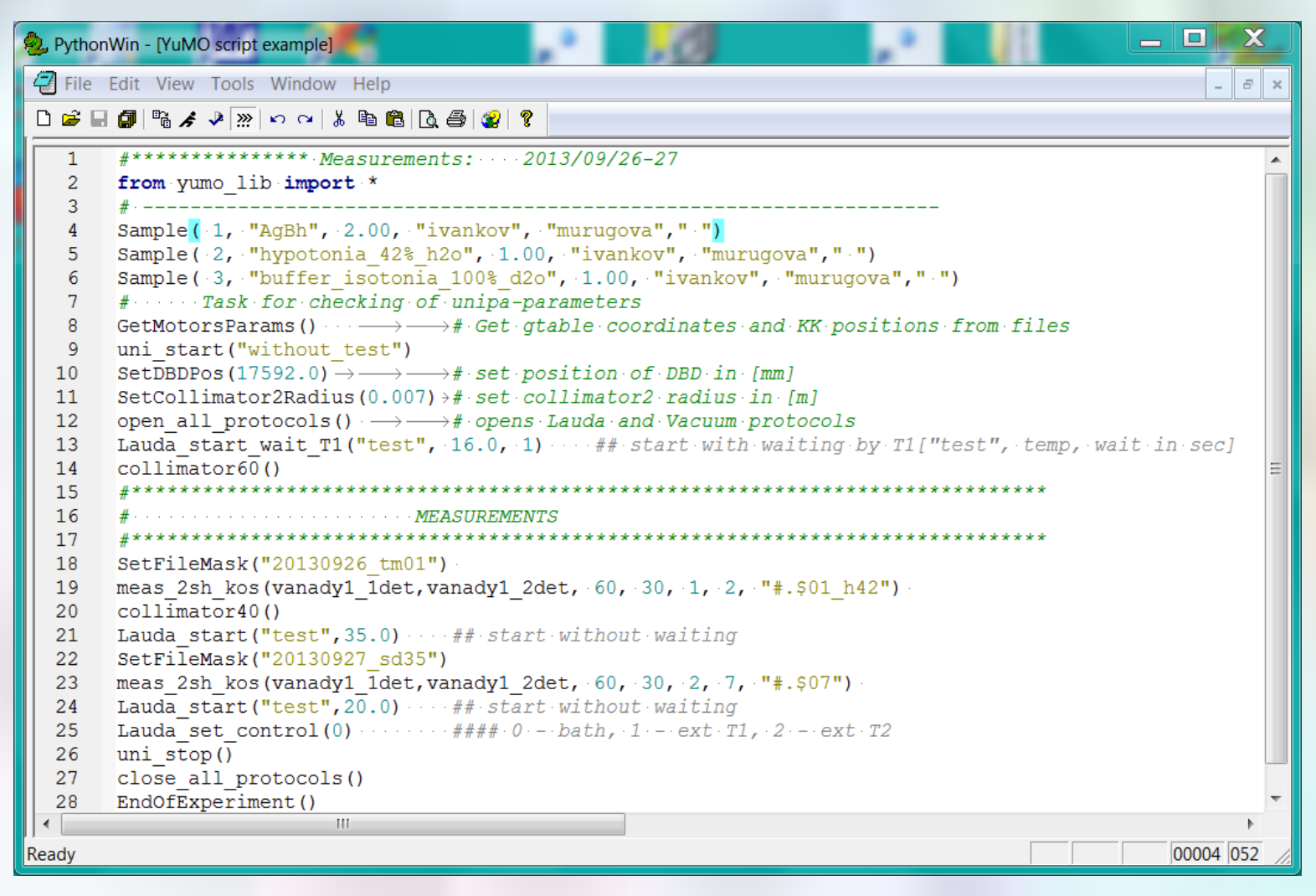
```
>>> lfs[0][0]
'\tdef Set (self, state_id):\n'
```

Thus, from the configuration file, we can get a list of all announced devices, for each device - to form a full list of commands, for each command - prompt with the names of the parameters.



```
-->s0.expo(100000,'e:/tmp.tmp',1)
(0, 'sleep')
(0, 'run')
(0, 'ok')
(1, 'ES_Expo_epilog: error internal state or break')
-->
```

Exposure manipulation with the help of the ***manual control widget***


PythonWin - [YuMO script example]
File Edit View Tools Window Help
Ready
00004 052

```
#*************** Measurements:    2013/09/26-27
from yumo_lib import *
# ----------------------------------------------------------------------
Sample( 1, "AgBh", 2.00, "ivankov", "murugova"," ")
Sample( 2, "hypotonia_42%_h2o", 1.00, "ivankov", "murugova"," ")
Sample( 3, "buffer_isotonia_100%_d2o", 1.00, "ivankov", "murugova"," ")
#      Task for checking of unipa-parameters
GetMotorsParams()               # Get gtable coordinates and KK positions from files
uni_start("without_test")
SetDBDPos(17592.0)              # set position of DBD in [mm]
SetCollimator2Radius(0.007)     # set collimator2 radius in [m]
open_all_protocols()            # opens Lauda and Vacuum protocols
Lauda_start_wait_T1("test", 16.0, 1)    ## start with waiting by T1["test", temp, wait in sec]
collimator60()
#**************************************************************************
#                        MEASUREMENTS
#**************************************************************************
SetFileMask("20130926_tm01")
meas_2sh_kos(vanady1_1det,vanady1_2det, 60, 30, 1, 2, "#.$01_h42")
collimator40()
Lauda_start("test",35.0)    ## start without waiting
SetFileMask("20130927_sd35")
meas_2sh_kos(vanady1_1det,vanady1_2det, 60, 30, 2, 7, "#.$07")
Lauda_start("test",20.0)    ## start without waiting
Lauda_set_control(0)        #### 0 - bath, 1 - ext T1, 2 - ext T2
uni_stop()
close_all_protocols()
EndOfExperiment()
```

The ***YuMO*** *instrument script example*

Checking the script correctness

Preliminary check of a correctness of a script significantly reduces probability of errors of its execution, first of all due to detection of syntax errors, misprints, etc. The Python built-in compile function does not guarantee the correctness of the script.
As the library of instruments commands can be appended or modified at any moment, so self-examination of the parameters shall be executed in each command of a spectrometer.
It is important that in the check mode the script should not be executed, otherwise it doesn't make a sense.
Used in scriptutils.py procedure ***TabNanny*** (***PythonWin***) additionally allows you to check the correctness of the indentation structure But it isn't enough. For example, this check can not find misprints in the parameter list of commands.
To check the correctness of the script before the actual implementation of the following scheme was proposed.
Each command primarily checks its parameters. For this purpose next statement is inserted into a body of a command

```
if CheckParameters (<list type command parameters>):
return (0, "No errors")
```

In normal operation mode ***CheckParameters*** procedure is ignored. In the test mode type compatibility of the actual command parameters is checked with one specified in ***CheckParameters***. In case of error ***SyntaxError*** exception with specifying of number of erratic parameter is generated. The Test mode is controlled with special external flag. A available types of parameters are numbers (***'int'***, ***'float'***), text strings (***'str'***), device names (' ***dev*** ') and ***Varman*** database variables names (' ***var*** '). All this looks approximately so

```
def ElementaryMeasurement (n_meas, i, n_lost, tmpfn, df, zsum_mask, sum_proc):
    cont_points.SetCPoint (1) # assign checkpoint
    if com.CheckParameters ( 'int', 'int', 'int', 'str', 'str', 'str'):
        return (0, "No errors")
. . .
```

During testing actual values of parameters are obtained from an external frame. Test procedure is performed automatically in the Sonix+ GUI after script selection.

Use breakpoints to control the process of script interpretation

In the Sonix+ script is a pure Python code. It mostly consists of calls functions from so-called ***instrument library*** - a set of Python procedures implementing typical operations. These procedures may be considered as instrument ***commands***.

The interpretation of this code is performed by special module "Interpreter of script"(***Is***) written in C++. To control the process the User needs to be able to ***suspend***/***resume***, or ***abort*** the interpretation. Thus, simple ***PyRun_String*** or ***PyRun_File*** are not acceptable. To realize these opportunities the Pythons debugger class (***Pdb***) can solve the problem. However it strongly slows down the process if there is a mathematical processing of large arrays in script.

For saving normal speed of execution it is offered to enter quantization of actions in a script by means of so called ***break points***. To do this, the special procedure module ***cont_points*** was developed. Each instrument ***command*** source begins with call of

```
   cont_points.SetCPoint (1)
```

or

```
  cont_points.SetCPoint (1, "Set Collimator 40 (1), wait ...")
```

where options specify the level of the reference point and, if desired, an auxiliary comment.

This call can be point of intervention into the interpretation process by the script interpreter. First parameter of the ***SetCPoint*** procedure is a control point level. It is assumed, that the top level 1 is assigned to spectrometer commands. The lower levels are assigned to less valuable operations. The lowest level 100 is devoted to commands of direct device control. Control points hierarchy allows the User to perform measurement procedure step-by-step at necessary level of detailing.

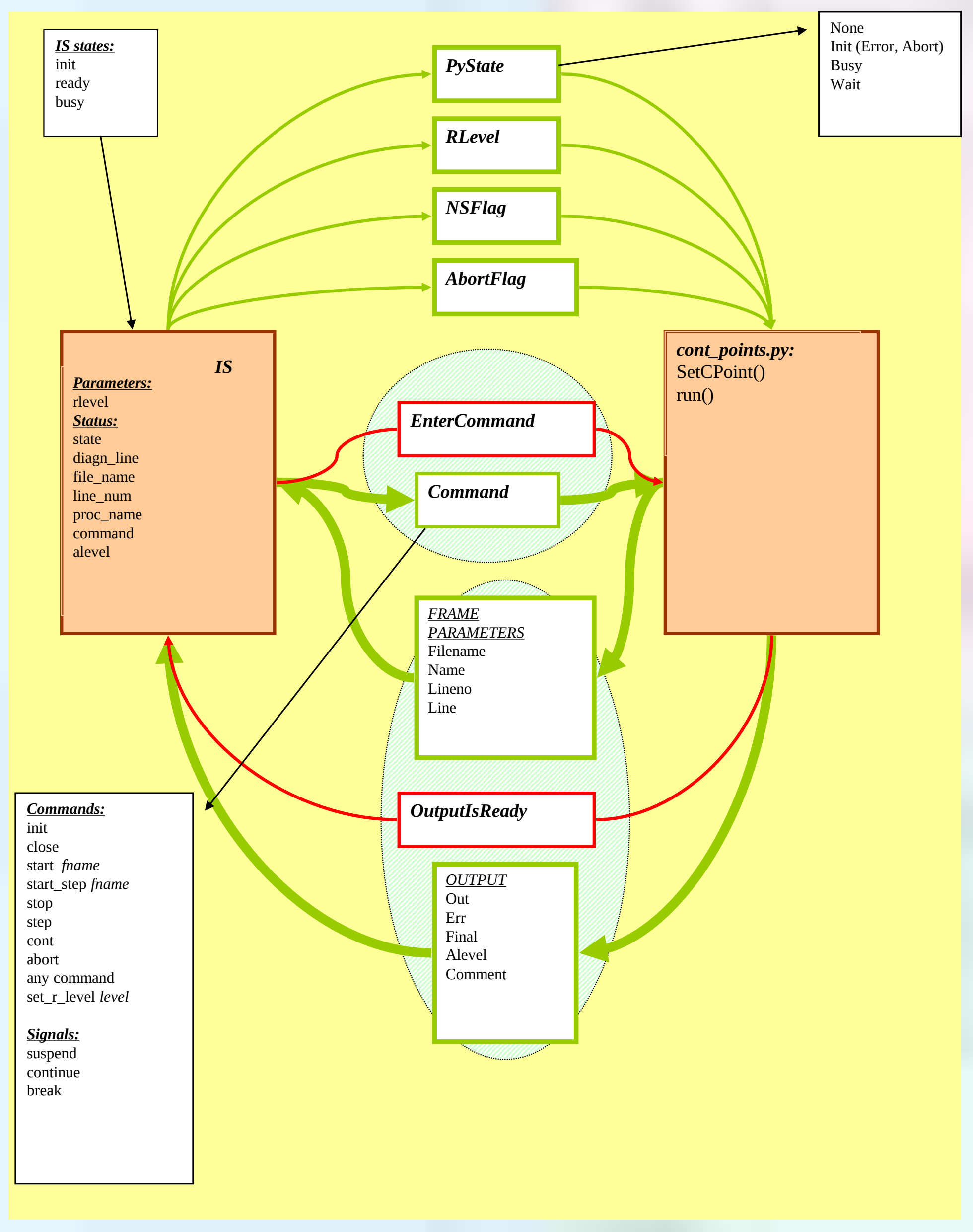
The diagram of communication of the interpreter and the executed script is given above.

The ***Is*** interpreter module interacts with the Python interpreter through the ***communication library*** (Python extension) which provides transmission of parameters, signals, flags. The program of measurement is launched via the ***cont_points*** module which implements the concept of ***break points***.

To implement the approach Python modules ***cmd***, and ***inspect*** were used. The first contains class ***Cmd***, which is very convenient for the realization of specialized interpretators based on the Python. Second - implements some useful functions to obtain information on existing objects, such as modules, classes, methods, etc.

The new class ***MyCmd*** on the base of original ***Cmd*** has been designed. New features like command to go the next break point, abort the execution, necessary flags and parameters were added. The possibility of input of arbitrary commands and execute it in current context was preserved also.

The cont_points module is supplied with a flag of ***DisableContPointsFlag*** which allows to control check permission. It is necessary, for example, when checking a correctness of a script.



The scheme of the **Is** *interpreter mplementation*

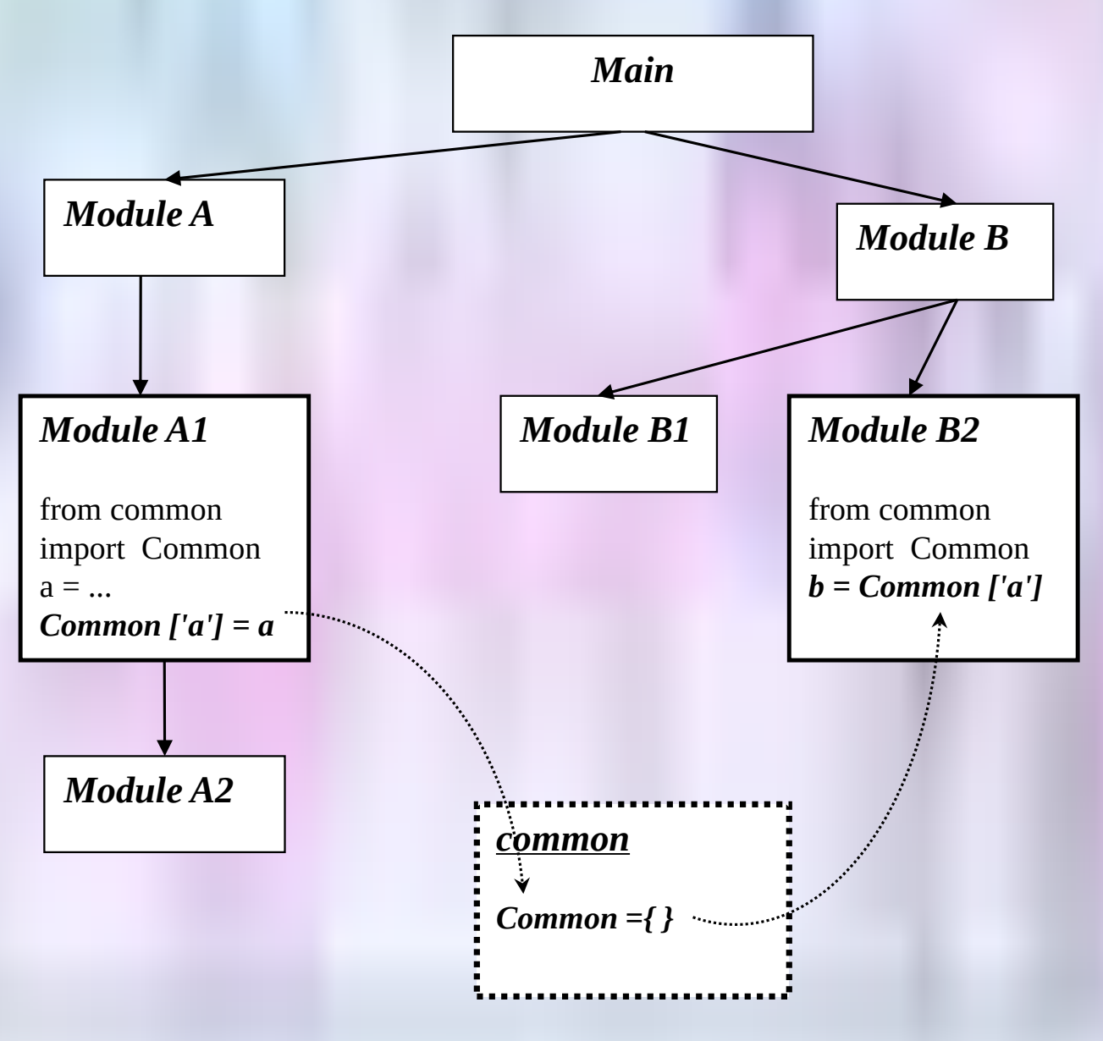
Programming cross-references for GUI widgets on PyQT

In case of assembly of user interfaces (GUI) from a set of components on there is a problem of data access arrangement, being in other branch of a structural tree. These data can be parameter values, links to methods (functions), etc. Especially as a part of components (widgets) can be used as independently, so as a part of other components. For example, the Sonix+ universal GUI contains the widget for direct stepper motor control. This opportunity is enabled before measurement start, but must be forbidden measurement script is executing.

The complexity of solutions due to the fact that the interface has a tree unpredictable structure of widgets that are created independently from each other. There is also a need to get a reference to concrete objects.

The Sonix+ has parameter storage **Varman** which could be help to parameters exchange. But the Varman can not transfer addresses of functions. Besides, for some programs it is desirable to make possible of using these programs without the Varman, for example, for off line spectra visualization.

One possible solution is put all the data structures in the top window widget. In this case, each class must be complemented by means of search the "***root***" window and storing and retrieving the necessary data. It is only possible to organize it if each widget in a hierarchical tree follows these rules as search of the main window is carried out strictly on widgets call chain.



The solution offered here is analogy to common block in the FORTRAN language. Let's select the separate module data contains cross-reference data from other modules. Actually Python create imported modules only once. So all other modules witch imports this "***post box***" module will be really connected to the same copy. In the example below the initially empty dictionary is devoted to be used by other modules.

Next sketch shows an example of use of this principle. The ***A1*** module has a variable (structure, function, and so on - the object), which is necessary in the ***B1*** module.

Reference   [Sonix +] http://sonix.jinr.ru/wiki/doku.php?id=en:ndex