1. Scripting Documentation . . . 3
- 1.1 Running and storing scripts . . . 3
- 1.2 Scripting tutorial - Developer license . . . 7
- 1.3 Scripting tutorial - Designer license . . . 15
- 1.4 schedule automatic scripts launching . . . 27
- 1.5 Using External Python Libraries . . . 29
- 1.6 Python API . . . 30
  - 1.6.1 Global functions and constants . . . 32
    - 1.6.1.1 Advanced functions . . . 43
  - 1.6.2 ArkAttribute objects . . . 44
    - 1.6.2.1 ArkAttribute - Advanced methods . . . 45
  - 1.6.3 ArkAttributeType objects . . . 46
  - 1.6.4 ArkChoice objects . . . 47
  - 1.6.5 ArkImage objects . . . 48
  - 1.6.6 ArkMatrixType objects . . . 48
    - 1.6.6.1 ArkMatrixType - Advanced properties . . . 52
  - 1.6.7 ArkObj objects . . . 52
    - 1.6.7.1 ArkObj objects - Advanced methods . . . 52
  - 1.6.8 ArkObjRef objects . . . 56
    - 1.6.8.1 ArkObjRef objects - Advanced methods . . . 63
  - 1.6.9 ArkPhase objects . . . 64
  - 1.6.10 ArkProject objects . . . 64
  - 1.6.11 ArkTreeViewObj objects . . . 64
    - 1.6.11.1 ArkTreeViewObj - Advanced properties . . . 68
  - 1.6.12 ArkSimulinkLib objects . . . 68
    - 1.6.12.1 ArkSimulinkConnection objects . . . 68
    - 1.6.12.2 ArkSimulinkObject objects . . . 69
  - 1.6.13 ArkVariant objects . . . 70
  - 1.6.14 ArkVariantFilter objects . . . 70
  - 1.6.15 ArkConnection Objects . . . 72
  - 1.6.16 ArkWorkspace Objects . . . 72
  - 1.6.17 ArkUser objects . . . 73
    - 1.6.17.1 ArkRule objects . . . 73
  - 1.6.18 ArkGroup objects . . . 74
  - 1.6.19 ArkTab objects . . . 74
  - 1.6.20 ArkBusyDialog objects . . . 77
  - 1.6.21 Custom View Filters . . . 78
- 1.7 Available libraries . . . 79
  - 1.7.1 utils . . . 79
    - 1.7.1.1 arkiexcel . . . 79
    - 1.7.1.2 arkiutils . . . 81
    - 1.7.1.3 xmltodict . . . 88
    - 1.7.1.4 utils.revisions . . . 90
    - 1.7.1.5 arkinternal . . . 90
    - 1.7.1.6 arki_deepcopy . . . 93
  - 1.7.2 arkiexport . . . 93
  - 1.7.3 arkilayout . . . 96
  - 1.7.4 arkimatrixexport . . . 97
  - 1.7.5 arkivariantsexport . . . 98
  - 1.7.6 matrixupdater . . . 99
  - 1.7.7 projectsmerger . . . 101
  - 1.7.8 ui . . . 103
  - 1.7.9 word_template . . . 105
  - 1.7.10 arki.graph . . . 109
    - 1.7.10.1 arki.graph.transfer_positions . . . 109
    - 1.7.10.2 arki.graph.graph_parser . . . 110
  - 1.7.11 advanced . . . 110
    - 1.7.11.1 references . . . 110
  - 1.7.12 MIC Model . . . 112
    - 1.7.12.1 MIC generic model . . . 113
    - 1.7.12.2 MIC arKItect interfaces . . . 114

1.7.12.3 MIC Simulink interfaces 114
1.7.12.4 MIC Amesim interfaces 114
1.7.12.5 MIC standalor editor interfaces 114
1.7.13 Generic Chains (GChains) API 114

Scripting Documentation

Content

- Running and storing scripts
- Scripting tutorial - Developer license
- Scripting tutorial - Designer license
- schedule automatic scripts launching
- Using External Python Libraries
- Python API
    - Global functions and constants
    - ArkAttribute objects
    - ArkAttributeType objects
    - ArkChoice objects
    - ArkImage objects
    - ArkMatrixType objects
    - ArkObj objects
    - ArkObjRef objects
    - ArkPhase objects
    - ArkProject objects
    - ArkTreeViewObj objects
    - ArkSimulinkLib objects
    - ArkVariant objects
    - ArkVariantFilter objects
    - ArkConnection Objects
    - ArkWorkspace Objects
    - ArkUser objects
    - ArkGroup objects
    - ArkTab objects
    - ArkBusyDialog objects
    - Custom View Filters
- Available libraries
    - utils
    - arkiexport
    - arkilayout
    - arkimatrixexport
    - arkivariantsexport
    - matrixupdater
    - projectsmerger
    - ui
    - word_template
    - arki.graph
    - advanced
    - MIC Model
    - Generic Chains (GChains) API

Download

You can download this documentation in pdf format.

Running and storing scripts

arKItect supports several ways of storing and executing Python scripts:

Content

- How to run scripts in arKItect
    - Scripts in object attributes
        - Using the attribute editor
        - Using the object context menu
        - Using events mechanism
            - Event parameters
    - Using Project Tools menu (root attributes)
    - Using the Common Tools menu (local scripts)
- How to catch errors
- Profiling scripts
- Using Bulk Mode
- Using Local Script Context
- Using the Wait dialog

How to run scripts in arKItect

Scripts in object attributes

This is most usual way. Scripts are stored in the **Program** attributes of an object. When executed, scripts in the attributes receive a reference to the object, used to run the script. Such script can be executed in the following ways:

Using the attribute editor

One of the ways to run script is to open the script editor and hit the **Run** button in it. Window below the script will show the output.

To open the attribute editor do one of the following actions:

- Click **Edit** in the object properties
- Right-click on the object attribute in the Internal Block Diagram, and choose **Edit** from the pop-up menu.

Using the object context menu

If the object attribute is displayed in the Internal Block Diagram, on the container of its parent object, it is possible to execute it by right-clicking on the attribute and choosing **Run** from the appearing context menu. This way, a script can be executed directly without accessing the script editor.

Using events mechanism

There is a specific mechanism of events which allows to launch scripts defined by object Program attributes on specific user actions.

These events are to be linked to object Program attributes in the specific interface of the abstract type definition (see Trigger Events).

Necessary steps to enable event triggering:

1. Define a Program attribute for the type where event is needed and set a default value for it
2. Ensure it is checked in the projection where it is planned to use the event
3. In the Events property page of the abstract type set a correspondence between the desired event and the Program attribute defined at the step 1.

Event parameters

When executing programs after events it is sometimes important to have access to additional information about modified objects or values. To do so an additional (optional) parameter is passed to the script of the Program attribute. This is a dictionary object where ('parameter' - 'value') pairs are added. For all events one parameter is always passed: 'eventType'

Today, only four event types - **Rename object**, **Change attribute value**, **Delete object** and **Change projection** - provide specific parameters for this extended functionality.

- Rename object:
    - additional parameter is the old name of the object ('oldName')
- Change attribute value:

    - additional parameter 1 - old value of the attribute ('oldValue')
    - additional parameter 2 - attribute name ('attributeName')
- Delete object:
    - additional parameter is the flag whether the last instance of the object is being deleted ('lastInstance')
- Change projection:
    - additional parameter 1 - previous active projection ('oldProjection')
    - additional parameter 2 - new active projection('newProjection')

The following code illustrates how this additional parameter can be treated in the event scripts:

```
import pyark
def runEvent(self, arg_dict):
 if( pyark.EVENT_ONRENAME == arg_dict['eventType'] ):
  print 'object old name is : ', arg_dict['oldName']
 elif( pyark.EVENT_ONCHANGEATTRIBVALUE == arg_dict['eventType'] ):
  print 'attribute : ', arg_dict['attributeName']
  print 'attribute old value is : ', arg_dict['oldValue']
  print 'attribute new value is : ',
self.GetAttribute(arg_dict['attributeName']).GetValue()
 elif( pyark.EVENT_ONDELETE == arg_dict['eventType'] ):
  print 'is last instance : ', arg_dict['lastInstance']
 elif( pyark.EVENT_ONCHANGEPROJECTION == arg_dict['eventType'] ):
  print 'previously active projection is : ', arg_dict['oldProjection']
  print 'newly active projection is : ', arg_dict['newProjection']
```

Using Project Tools menu (root attributes)

Root object of the architecture can also have Program attributes, like any other object. Root attributes are usually used for storing some project-specific utilities. Besides the usual way of running such scripts from the attribute editor or from the root context menu, they can be executed from the **arKItect**'s **Project Tools** in the **Scripting Panel of the Tools Category**.

This menu contains a list of all program attributes from the root object, visible in the currently active projection. Attribute groups can be used to organize scripts, they are shown as sub-menus.

These scripts are executed in the same way as the usual script attributes. They receive a reference to the root object in the currently active projection.

Using the Common Tools menu (local scripts)

Sometimes it is convenient to run a script without using attributes, or to have a script available to all projects.

Such Python scripts (files with **.py** extension) are located in the folder *Scripts* near the **arKItect** executable (scripts from the local machine) and they can be executed from the **arKItect**'s **Common Tools** in the **Scripting Panel of the Tools Category**.

This folder can contain sub-folders, they will be shown as sub-menus. Activating a menu item will execute corresponding script as if it were an "unattended" script, i.e. code is simply executed, and no object reference is passed to it.

However, the script can access currently active projection and selected objects, using the following API functions:

- **pyark.GetActiveView()**
  Returns tuple (**ArkObjRef**, **view_type**), describing the currently shown root and view type (IBD/RDB/tabular view).

- **pyark.GetSelection()**
  Returns list of **ArkObjRef** objects, corresponding to the current selection, if view type is Internal Block Diagram or Relation Block Diagram. For the tabular view, it returns an empty list.

How to catch errors

Script errors should be caught using the **arKItect** logging mechanism in order to notify the user when errors occur.

Usage:

```
from arki.utils import arkilogging as logging # import the logging module
logger = logging.getLogger("My script name or alias") # use any name you want to
identify your script
```

Whenever you catch an error, critical error, or a warning, you use the logger to raise it:

```
logger.error( "Failed to ...")
logger.critical( "Critical error...")
logger.warning( "This is a warning...")
```

After script termination, **arKItect** will show "Python Events" window with log messages, if any new message of sufficient severity was added (severity can be configured in the settings dialog of this window). By default, **warning** and higher (error, critical, exception) activate this window. This window also can be opened manually from the main menu: Tools > Python Events.

Normally, "Python Events" window is only opened after the termination of the script. To show it in the course of script execution, use method **pyark.SignalLogWindow()**. Note that this method only shows window, if events of sufficient severity were added.

Profiling scripts

you can store performance execution details in a text file in order to analyse in which function does the code mostly spend time.

```
def run(self):
    import cProfile, pstats
    pr = cProfile.Profile()
    pr.enable()
    ###################
    # your code here ##
    ###################
    pr.disable()
    with open(r"C:\somepath\profiling.txt", "w") as s:
        ps = pstats.Stats(pr, stream=s).sort_stats('cumulative')
        ps.print_stats()
```

Using Bulk Mode

Bulk mode allows **arKItect** to send packaged requests to server in order to optimize script performance.

The usage is very simple:

```
from arki.utils.arkinternal import BulkModeContext
with BulkModeContext():
 ###################
    # your code here ##
    ###################
```

As a general rule,use it only when you create / delete many objects or modify many attributes.

Using Local Script Context

Local Script Context allows **arKItect** to execute scripts locally without really updating the project. It means that all changes will be done in memory but not in the database. It maybe very useful when debugging a script which makes a lot of changes - until user is sure that the script works correctly he might use the Local Script Context.

```
from arki.utils.arkinternal import LocalScriptContext
with LocalScriptContext():
 ###################
    # your code here ##
    ###################
```

As soon as the project is refreshed, locally done changes are lost.

Usually after a script has been successfully debugged the **Local Script Context** is replaced with the **Bulk Mode**.

Using the Wait dialog

The wait dialog allows you to show a "please wait" dialog while you run some scripts. This helps user identify when the script execution is done (dialog closed).

usage:

```
import pyark

with pyark.ui.ArkBusyDialog("doing some long task\r\nPlease wait ...") : #\r\n allow
to add a carriage return
    doSomeLongTask()
```

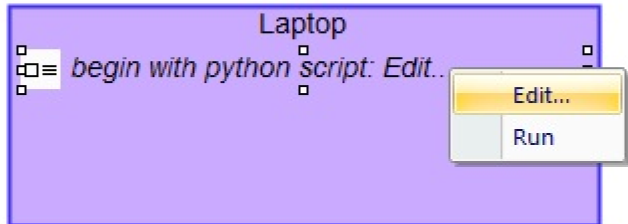
Scripting tutorial - Developer license

Prerequisites

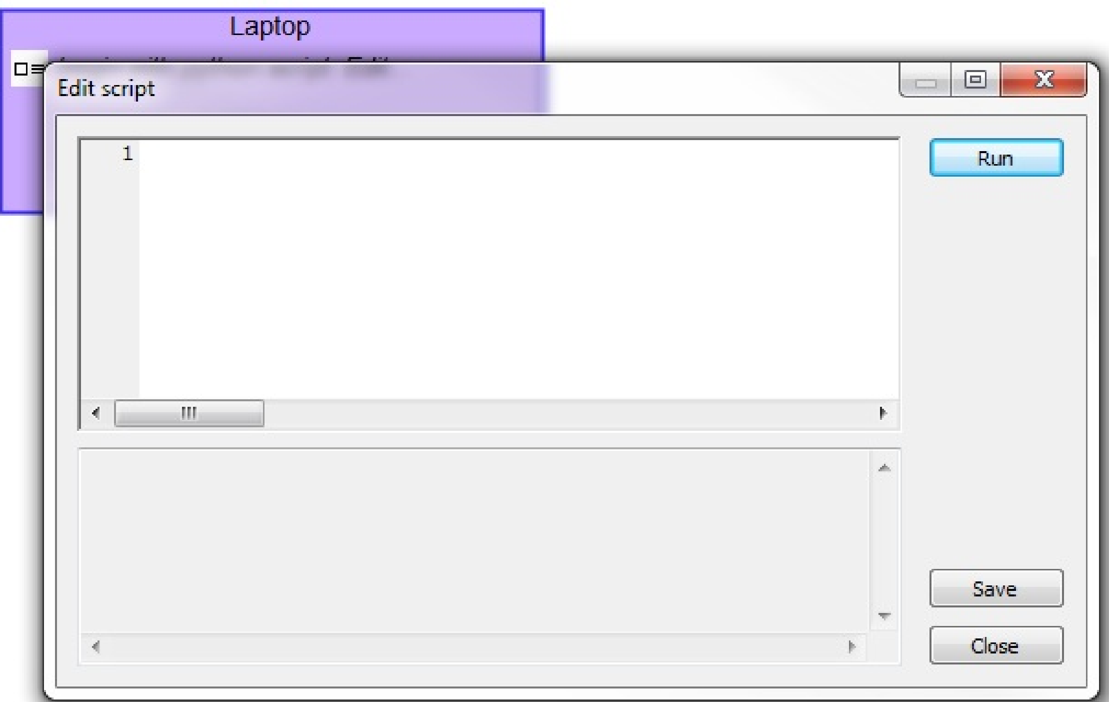
- You will need **Developer** license.
- You should have read the following tutorial: arKItect Developer Getting Started.
- You should have a **Meta-model** with at least one Program attribute in the project.

Hello world

Find a Program attribute in your project and click on the **Edit** button to open the script editor window.



You'll see the following window:



You can now add following code into the editor:

It is important to know that all the code must be written in the *run* function. The value of *self* represents the reference of the object where this script is run.

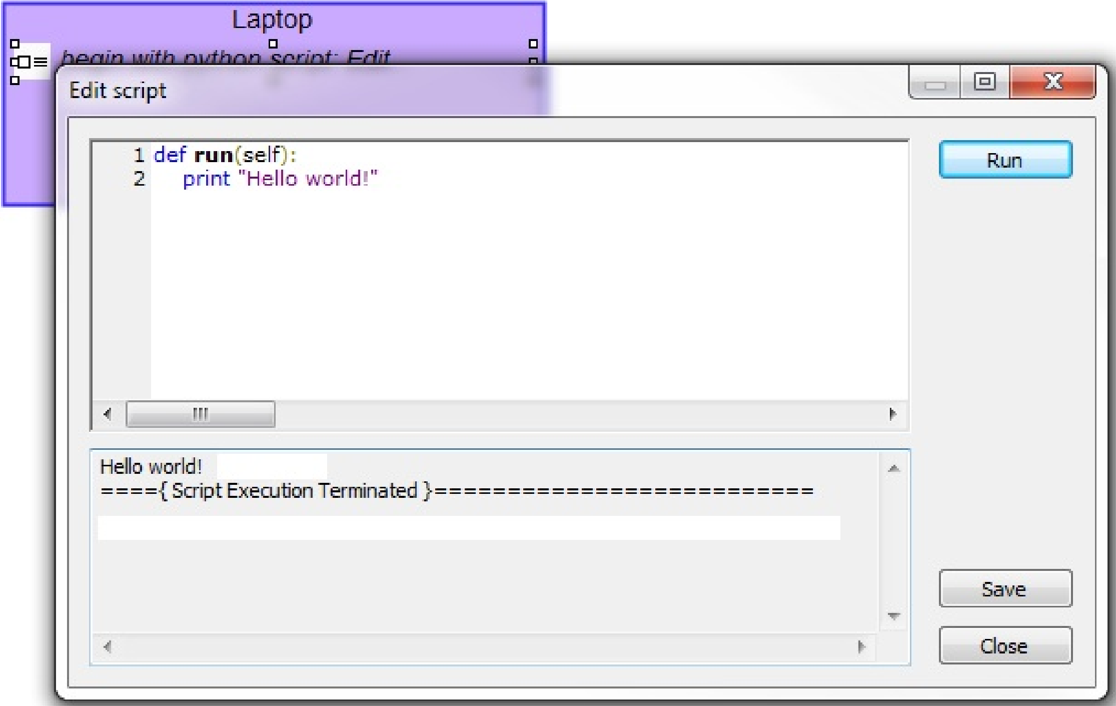
The name of the function is not important, here defined as *run*; you can choose your own name.

```
def run(self):
    print "Hello world!"
```

When you click on the **Run** button, the script is executed and result is printed out in the Program Trace:

```
Hello world!
===={ Script Execution Terminated }==========================
```

Congratulation! You have written your first python script in **arKItect**.



First script

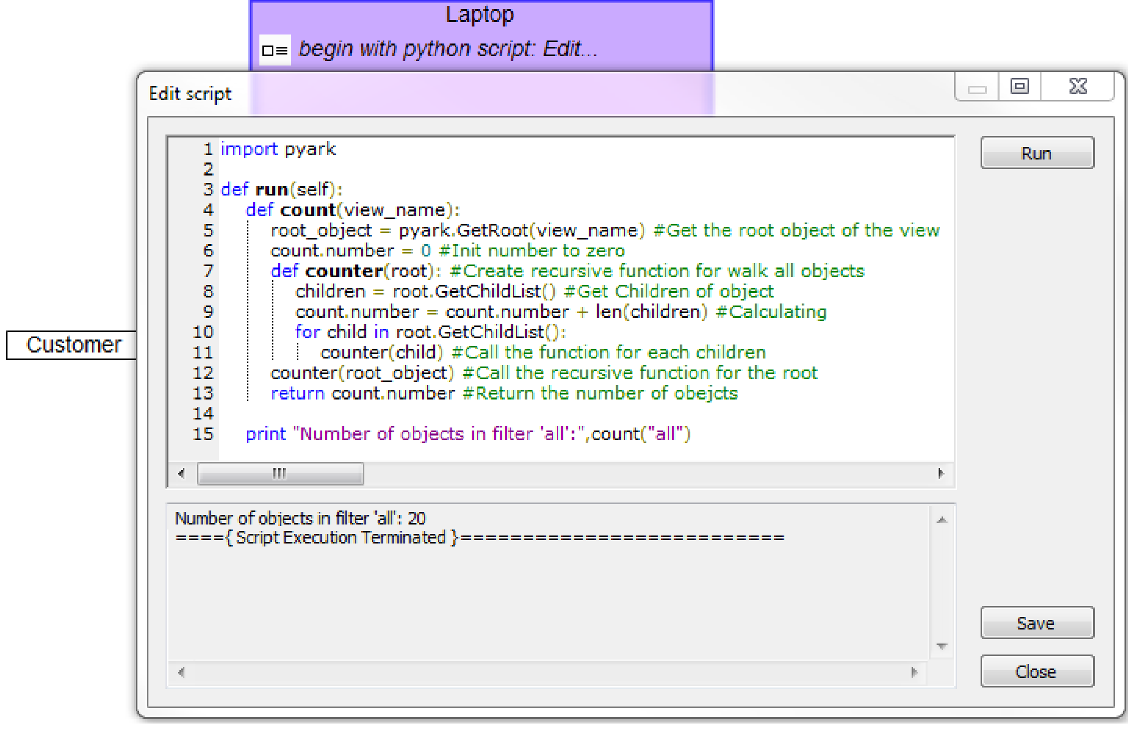
In this step, we will create a simple reporting tool for the project in a given projection.

Count all objects

In order to count all objects, we need to start on the root object in the given projection and read all children until we arrive at objects that have no children.

```
import pyark
def count(view_name):
    root_object = pyark.GetRoot(view_name) #Get the root object of the view
    count.number = 0 #Init number to zero
    def counter(root): #Create recursive function for walk all objects
        children = root.GetChildList() #Get Children of object
        count.number = count.number + len(children) #Calculating
        for child in root.GetChildList():
            counter(child) #Call the function for each children
    counter(root_object) #Call the recursive function for the root
    return count.number #Return the number of objects
```

If we run the script in **arKItect**:




```
import pyark

def run(self):
    def count(view_name):
        root_object = pyark.GetRoot(view_name) #Get the root object of the view
        count.number = 0 #Init number to zero
        def counter(root): #Create recursive function for walk all objects
            children = root.GetChildList() #Get Children of object
            count.number = count.number + len(children) #Calculating
            for child in root.GetChildList():
                counter(child) #Call the function for each children
        counter(root_object) #Call the recursive function for the root
        return count.number #Return the number of objects

    print "Number of objects in filter 'all':",count("all")
```

Result:

```
Number of objects in filter "all": 20
===={ Script Execution Terminated }==========================
```

It is very important to know that one object can have several references in one projections, so here in this script the references are counted.

List all types

To list all types, we need to get the root reference of Rules and read all children.

```
import pyark
def listTypes():
    root_matrix = pyark.GetArkMatrixType() #Get the root rule
    listTypes.types = list() #Init a list for types
    def walker(root): #Create recursive function for walk all rule
        for child in root.GetChildList(): #Get Children rules
            if not child.GetName() in listTypes.types: #If rule is not yet found
                listTypes.types.append(child.GetName()) #add it to the list
                walker(child) #Call the function for each children
    walker(root_matrix) #Call the recursive function for the root rule
    return listTypes.types #Return the list of types
```

If we run the script in **arKItect**:

```
import pyark

def run(self):
    def listTypes():
        root_matrix = pyark.GetArkMatrixType() #Get the root rule
        listTypes.types = list() #Init a list for types
        def walker(root): #Create recursive function for walk all rule
            for child in root.GetChildList(): #Get Children rules
                if not child.GetName() in listTypes.types: #If rule is not yet found
                    listTypes.types.append(child.GetName()) #add it to the list
                    walker(child) #Call the function for each children
        walker(root_matrix) #Call the recursive function for the root rule
        return listTypes.types #Return the list of types
    print listTypes()
```

Result:

```
['System', 'Component', 'Function', 'Link', 'Actor']
===={ Script Execution Terminated }==========================
```

Count objects by type

To count objects by types in a given projection, we can use our functions *count* and *listTypes*. We must only change the *count* function so that it works with a given type, and call it with each type retrieved by *listTypes*.

```
import pyark
def count(view_name,type=None): #type is not needed, can be None and return for all
types
    root_object = pyark.GetRoot(view_name) #Get the root object of the view
    count.number = 0 #Init number to zero
    def counter(root): #Create recursive function for walk all objects
        if type:
            children = root.GetChildList(type) #Get Children of object for a given
type
        else:
            children = root.GetChildList() #Get Children of object
        count.number = count.number + len(children) #Calculating
        for child in root.GetChildList():
            counter(child) #Call the function for each children
    counter(root_object) #Call the recursive function for the root
    return count.number #Return the number of objects
```

We just have to write a simple function like this:

```
def countByType(view_name):
    for type in listTypes():
        print "Number of objects of type '%s' in filter '%s':
%s"%(type,view_name,count(view_name,type))
```

Result:

```
Number of objects of type 'System' in filter 'all': 1
Number of objects of type 'Component' in filter 'all': 9
Number of objects of type 'Function' in filter 'all': 8
Number of objects of type 'Link' in filter 'all': 0
Number of objects of type 'Actor' in filter 'all': 2
===={ Script Execution Terminated }==========================
```

Display architecture with script

We will write a script that displays the architecture in a log window. For example, if we have two components (*Component1* and *Component2*) allocated to a system (*System*), result will be the following:

```
- System (System)
     - Component1 (Component)
     - Comonent2 (Component)
```

The function to do this will be :

```
def PrintArchitecture(view_name):
    root_object = pyark.GetRoot(view_name) #Get the root object of the view
    def printer(obj,level): #Create recursive function for walk all objects
        print "    "*level,"-",obj
        for child in obj.GetChildList():
            printer(child,level+1)
    print "Display architecture since root object in view '%s':"%view_name
    printer(root_object,0)
```

If we run the script in **arKItect**:

```
import pyark

def run(self):
    def PrintArchitecture(view_name):
        root_object = pyark.GetRoot(view_name) #Get the root object of the view
        def printer(obj,level): #Create recursive function for walk all objects
            print "    "*level,"-",obj
            for child in obj.GetChildList():
                printer(child,level+1)
        print "Display architecture since root object in view '%s':"%view_name
        printer(root_object,0)
    PrintArchitecture("all")
```

Result:

```
Display architecture since root object in view 'all':
 - My first python script (My first python script)
     - Laptop (System)
         - Screen (Component)
             - Display informations to user (Function)
         - Keyboard (Component)
             - Get user information to system (Function)
         - Mother board (Component)
             - Processor (Component)
                 - Manage all informations and components (Function)
             - Memory RAM (Component)
                 - Store informations (Function)
             - Graphical board (Component)
                 - Manage all graphical effect to display (Function)
             - Network board (Component)
                 - Allow user to connect laptop on a network (Function)
             - Hard Drive Disk (Component)
                 - Store data (Function)
         - Battery (Component)
             - Allow user to use laptop without electricity (Function)
     - Customer (Actor)
     - Supplier (Actor)
===={ Script Execution Terminated }=========================
```

Merge reporting

```
import pyark

def run(self):
    def count(view_name,type=None):
        root_object = pyark.GetRoot(view_name) #Get the root object of the view
        count.number = 0 #Init number to zero
        def counter(root): #Create recursive function for walk all objects
            if type:
                children = root.GetChildList(type) #Get Children of object for a given
type
            else:
                children = root.GetChildList() #Get  Children of object
            count.number = count.number + len(children) #Calculating
            for child in root.GetChildList():
                counter(child) #Call the function for each children
        counter(root_object) #Call the recursive function for the root
        return count.number
    def listTypes():
        root_matrix = pyark.GetArkMatrixType() #Get the root rule
        listTypes.types = list() #Init a list for types
        def walker(root): #Create recursive function for walk all rule
            for child in root.GetChildList(): #Get Children rules
                if not child.GetName() in listTypes.types: #If rule is not yet found
                    listTypes.types.append(child.GetName()) #add it to the list
                    walker(child) #Call the function for each children
        walker(root_matrix) #Call the recursive function for the root rule
        return listTypes.types #Return the list of types
    def PrintArchitecture(view_name):
        root_object = pyark.GetRoot(view_name) #Get the root object of the view
        def printer(obj,level): #Create recursive function for walk all objects
            print "    "*level,"-",obj
            for child in obj.GetChildList():
                printer(child,level+1)
        print "Display architecture since root object in view '%s':"%view_name
        printer(root_object,0)

    print "Number of objects in filter 'all':",str(count("all"))
    print "----"
    print "List of type in project:"
    for type in listTypes():
        print "    - %s: %s object(s) of this type in %s
filter"%(type,count("all",type),"all")
    print "----"
    PrintArchitecture("all")
```

The result of the reporting is:

```
Number of objects in filter 'all': 20
----
List of type in project:
    - System: 1 object(s) of this type in all filter
    - Component: 9 object(s) of this type in all filter
    - Function: 8 object(s) of this type in all filter
    - Link: 0 object(s) of this type in all filter
    - Actor: 2 object(s) of this type in all filter
----
Display architecture since root object in view 'all':
 - My first python script (My first python script)
     - Laptop (System)
         - Screen (Component)
             - Display informations to user (Function)
         - Keyboard (Component)
             - Get user information to system (Function)
         - Mother board (Component)
             - Processor (Component)
                 - Manage all informations and components (Function)
             - Memory RAM (Component)
                 - Store informations (Function)
             - Graphical board (Component)
                 - Manage all graphical effect to display (Function)
             - Network board (Component)
                 - Allow user to connect laptop on a network (Function)
             - Hard Drive Disk (Component)
                 - Store data (Function)
         - Battery (Component)
             - Allow user to use laptop without electricity (Function)
     - Customer (Actor)
     - Supplier (Actor)
===={ Script Execution Terminated }=========================
```

Other reporting scripts

You can use **arKItect** APIs to create objects, if your meta-model allows it. The reporting can be stored in an object (in the attributes) and a new object is created for each report. You can use versions for reporting.

For further information

See the documentation on Word document generation and create your own reporting in a .doc file.

Scripting tutorial - Designer license

Prerequisites

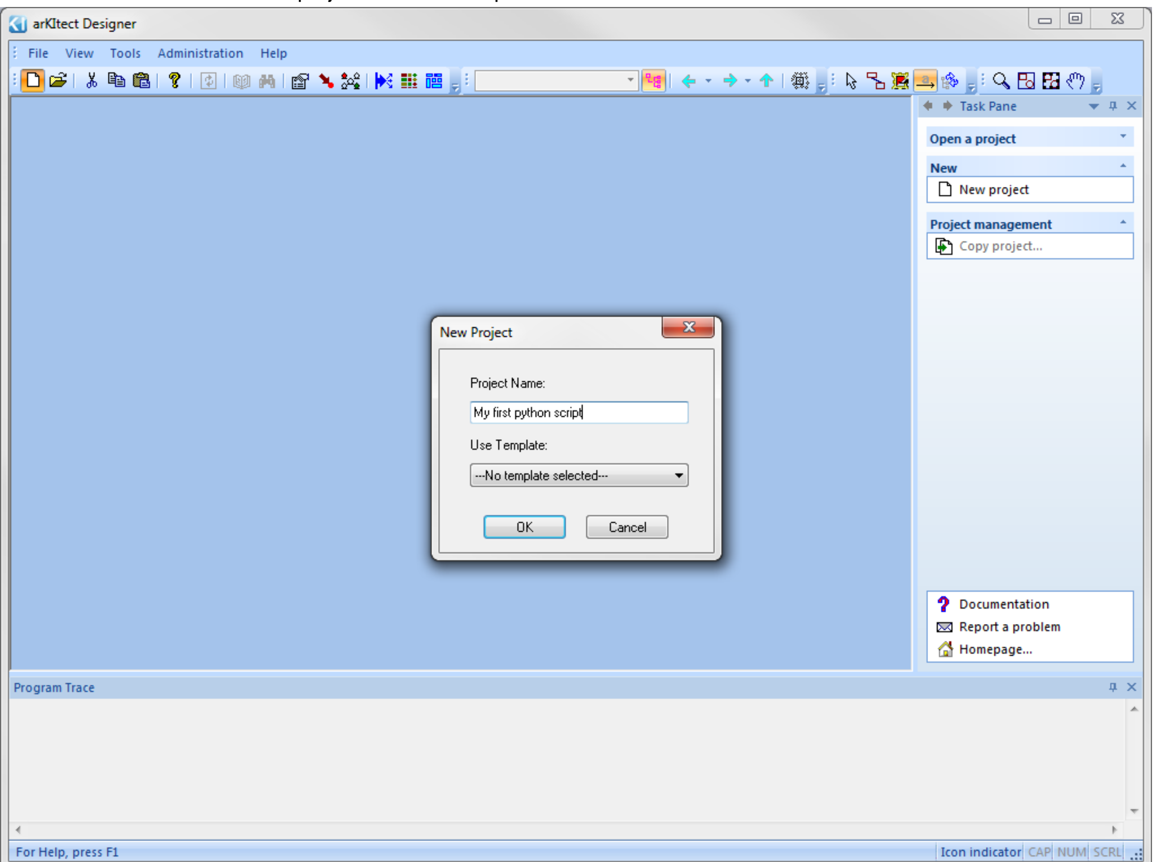
- You need a **Designer** license
- You should have read the following tutorial: arKItect Designer Getting Started
- You need to create a new empty project

Hello world

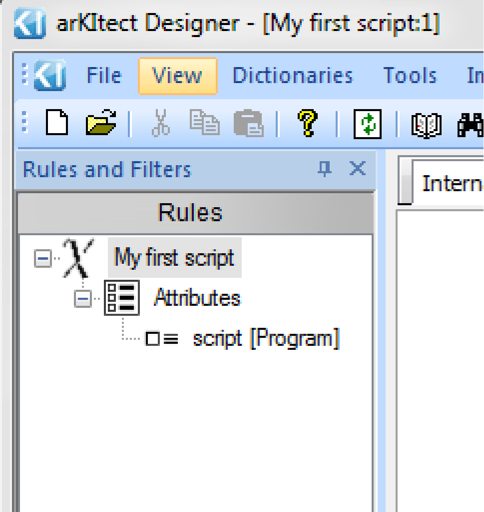
Meta-model prerequisites

To run a script, there are some prerequisites:

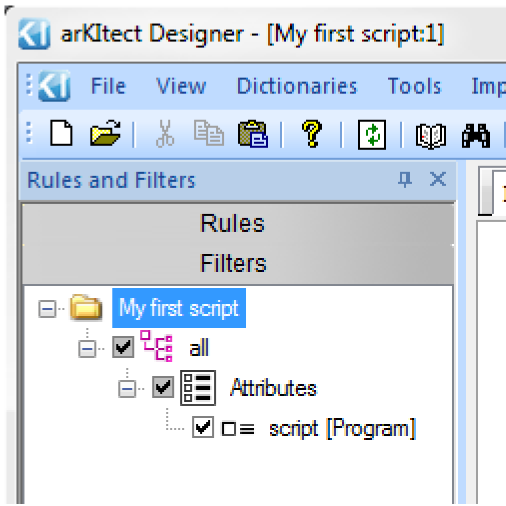
1. Run **arKItect** and create a new project without a template



2. Create a **Program** attribute at the root of Rules (used to store the script and launch it)



3. In order to launch the script, we need to have at least one projection where the script is accessible, so create a filter (named *all* for example) where all rules are checked



Creation of your first script

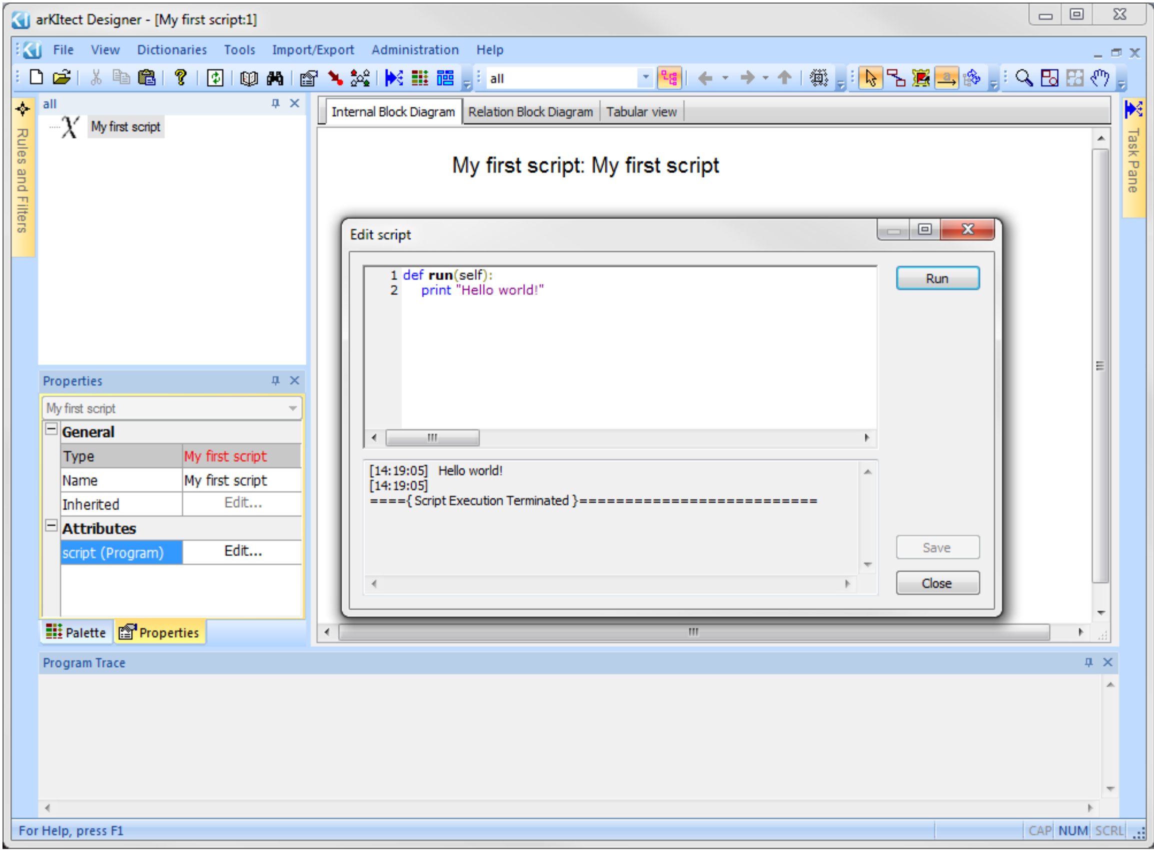
Now you can create and run your first script.

To get started, we can try to run a "Hello world" program:

```
def run(self):
    print "Hello world!"
```

Result:

```
Hello world!
===={ Script Execution Terminated }==========================
```



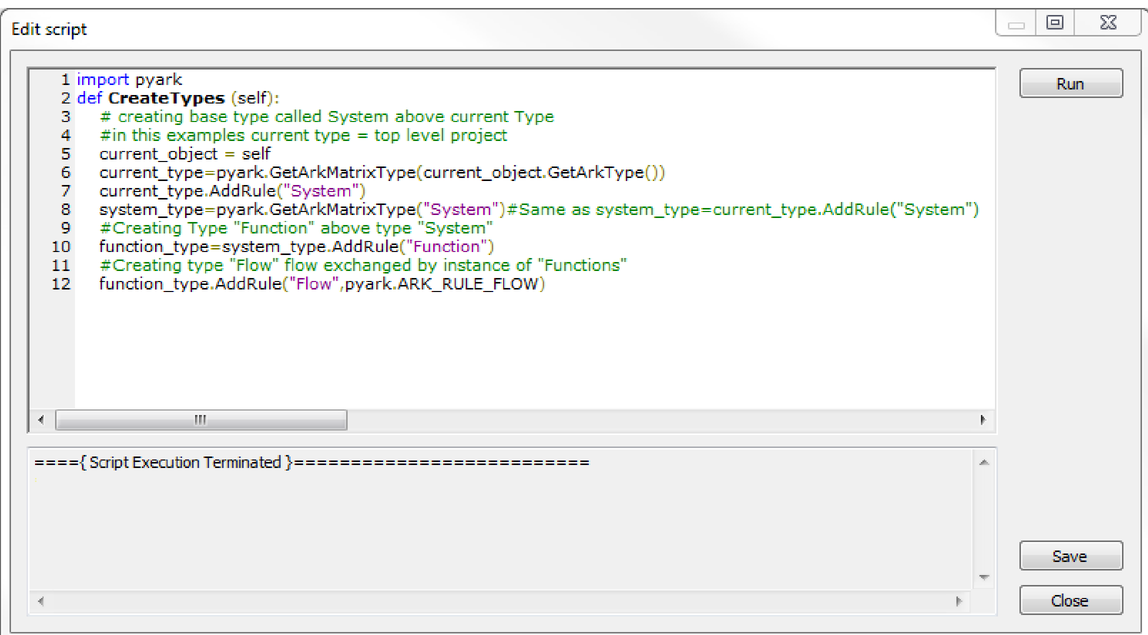
It is important to know that all the code you write must be in a function with a parameter *self* (the function can be named as you want). The value of *self* represents a reference to the object where this script is run (near the root object in view *all*).

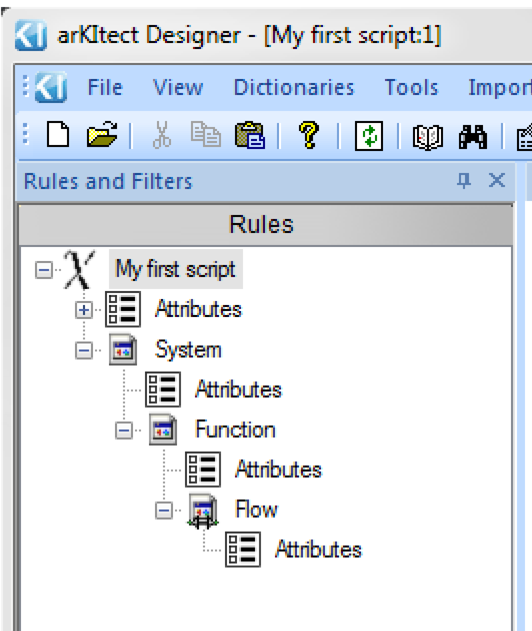
A script to create a simple meta-model

Create rules

Create a simple architecture with flow connecting functions allocated to systems.

```
import pyark
def CreateTypes (self):
    # creating base type called System above current Type
    #in this examples current type = top level project
    current_object = self
    current_type=pyark.GetArkMatrixType(current_object.GetArkType())
    current_type.AddRule("System")
    system_type=pyark.GetArkMatrixType("System")#Same as
system_type=current_type.AddRule("System")
    #Creating Type "Function" above type "System"
    function_type=system_type.AddRule("Function")
    #Creating type "Flow" flow exchanged by instance of "Functions"
    function_type.AddRule("Flow",pyark.ARK_RULE_FLOW)
```





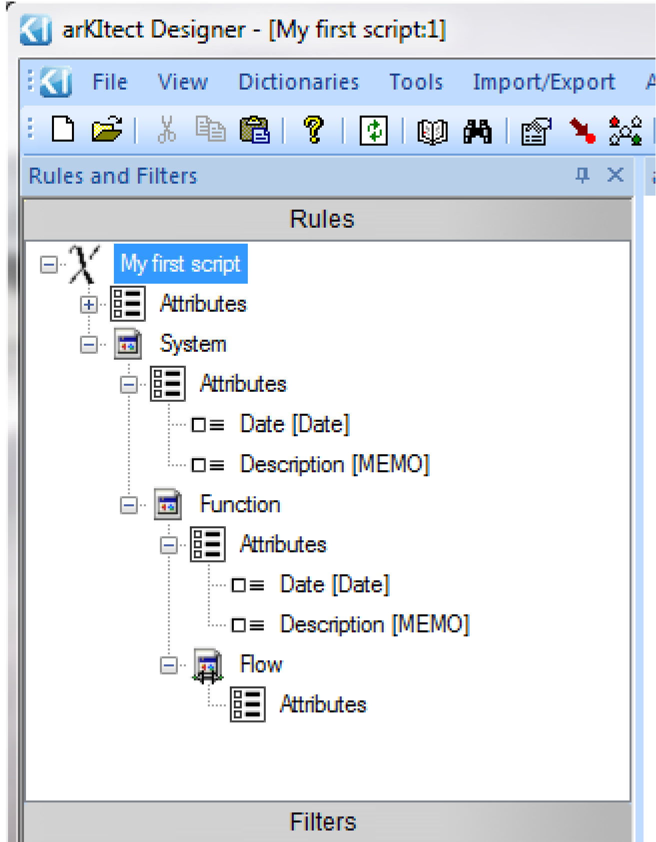
Add attributes to a given type

```
import pyark
def CreateTypes (self):
    #Get Reference of rule "System" and "Function" previously created
    system_type=pyark.GetArkMatrixType("System")
    function_type=pyark.GetArkMatrixType("Function")

    #Adding attributes to "System"
    system_type.AddAttribute("Description",pyark.ARK_ATTRIB_MEMO)
    system_type.AddAttribute("Date",pyark.ARK_ATTRIB_DATE)
    #Checking attributes have been added
    for attribute in system_type.GetAttributeList():
        print attribute.GetName()

    #Description and Date have now been created, they can be used and added to any
types
    # Adding them to type "Function"
    for attribute in system_type.GetAttributeList():
        function_type.AddAttribute(attribute)
        function_type.AddAttribute(attribute)
```

Result:

```
Date
Description
===={ Script Execution Terminated }==========================
```

We can see that the attributes have been created by a script.



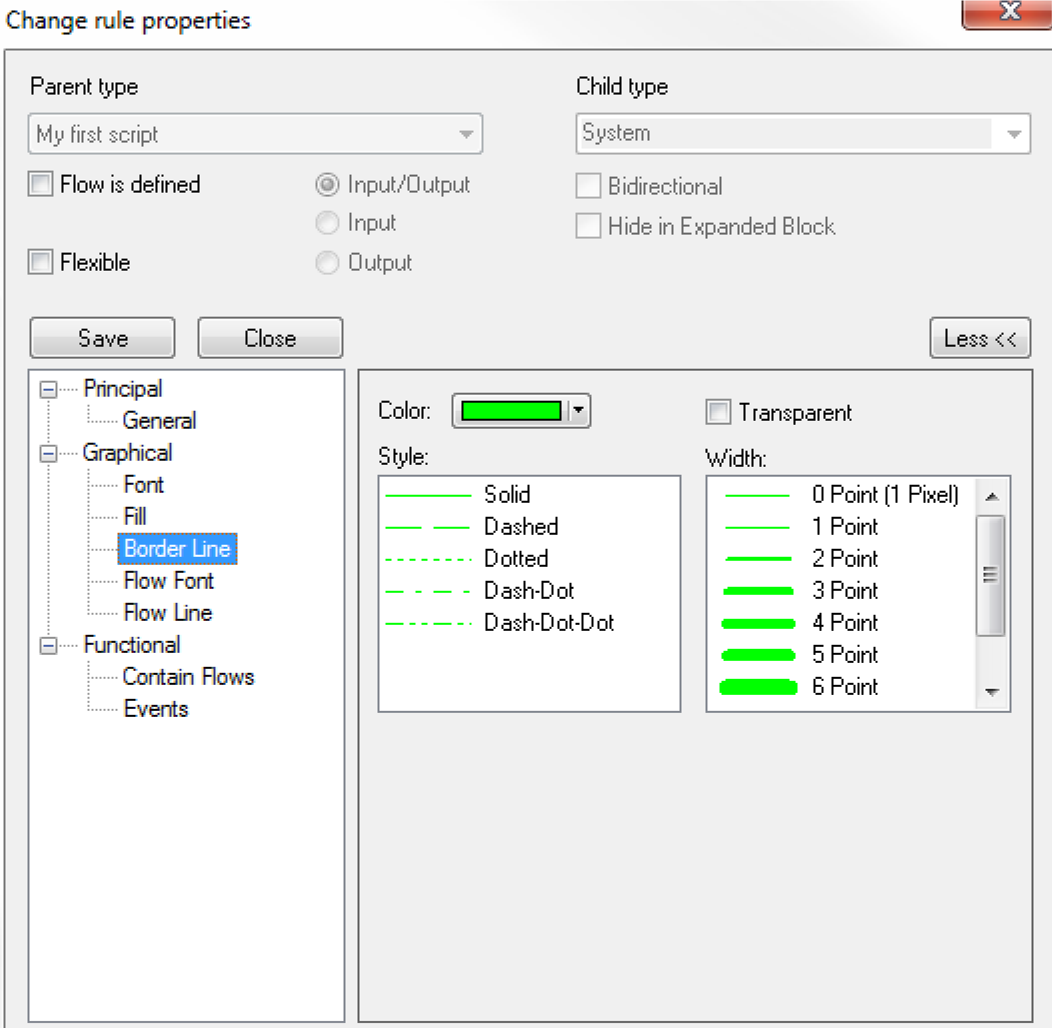
Change the properties of a given type

```
import pyark
def CreateTypes (self):
    #Get Reference of rule "System" and "Function" previously created
    system_type=pyark.GetArkMatrixType("System")
    function_type=pyark.GetArkMatrixType("Function")

    #Getting properties of "System"
    print system_type.GetProperty(pyark.ARK_ATYPE_CLRLINE)
    print system_type.GetProperty(pyark.ARK_ATYPE_OBJSHAPE)

    #Changing properties of "System"
    system_type.SetProperty(pyark.ARK_ATYPE_OBJSHAPE,pyark.ARK_OBJSHAPE_RECT_RE)
    system_type.SetProperty(pyark.ARK_ATYPE_CLRLINE,(0,255,0))
    system_type.FlashProperties() #Write definitly properties before this line

    #Changing properties of "Function"
    function_type.SetProperty(pyark.ARK_ATYPE_CLRFILL,(255,0,255))
    function_type.FlashProperties() #Write definitly properties before this line

    #Checking properties of "System"
    print system_type.GetProperty(pyark.ARK_ATYPE_CLRLINE)
    print system_type.GetProperty(pyark.ARK_ATYPE_OBJSHAPE)
```

Result:

```
(0, 0, 0)
Rectangle
(0, 255, 0)
Rounded Rectangle
Description
===={ Script Execution Terminated }=========================
```

The roperties have been changed.

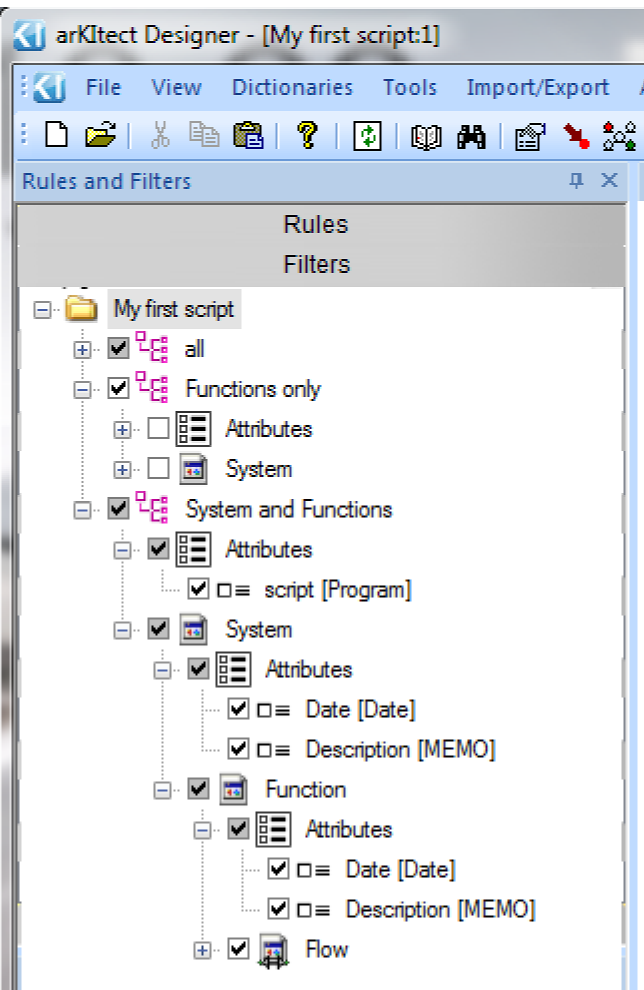


Create filters

```
import pyark
def CreateTypes (self):
    #Getting the root tree
    rootTree=pyark.GetRootTreeViewObj()
    rootTreeChildren=rootTree.GetChildList()
    #Get actual filters
    for view in rootTreeChildren:
        print view.GetName()

    rootTree.NewArkTreeViewObj("System and Functions",True) #Create a new filter and
check all rules
    rootTree.NewArkTreeViewObj("Functions only",False) #Create a new filter but any
rules are checked

    #Check new filters have been created
    for view in rootTree.GetChildList():
        print view.GetName()
```

Result:

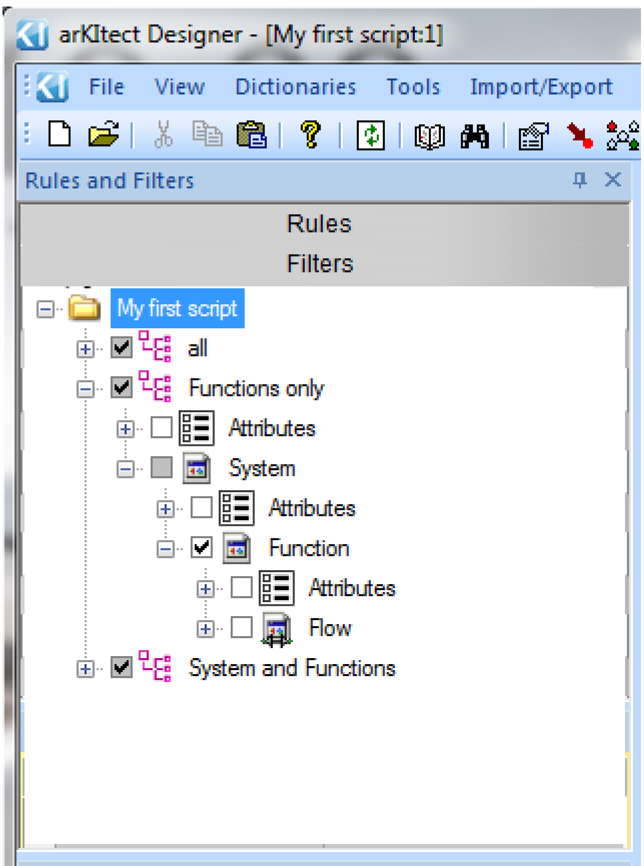
```
all
Functions only
all
System and Functions
===={ Script Execution Terminated }=========================
```



Check rules

In the filter *Functions*, no rules are checked, so we need check the rule Function in this filter.

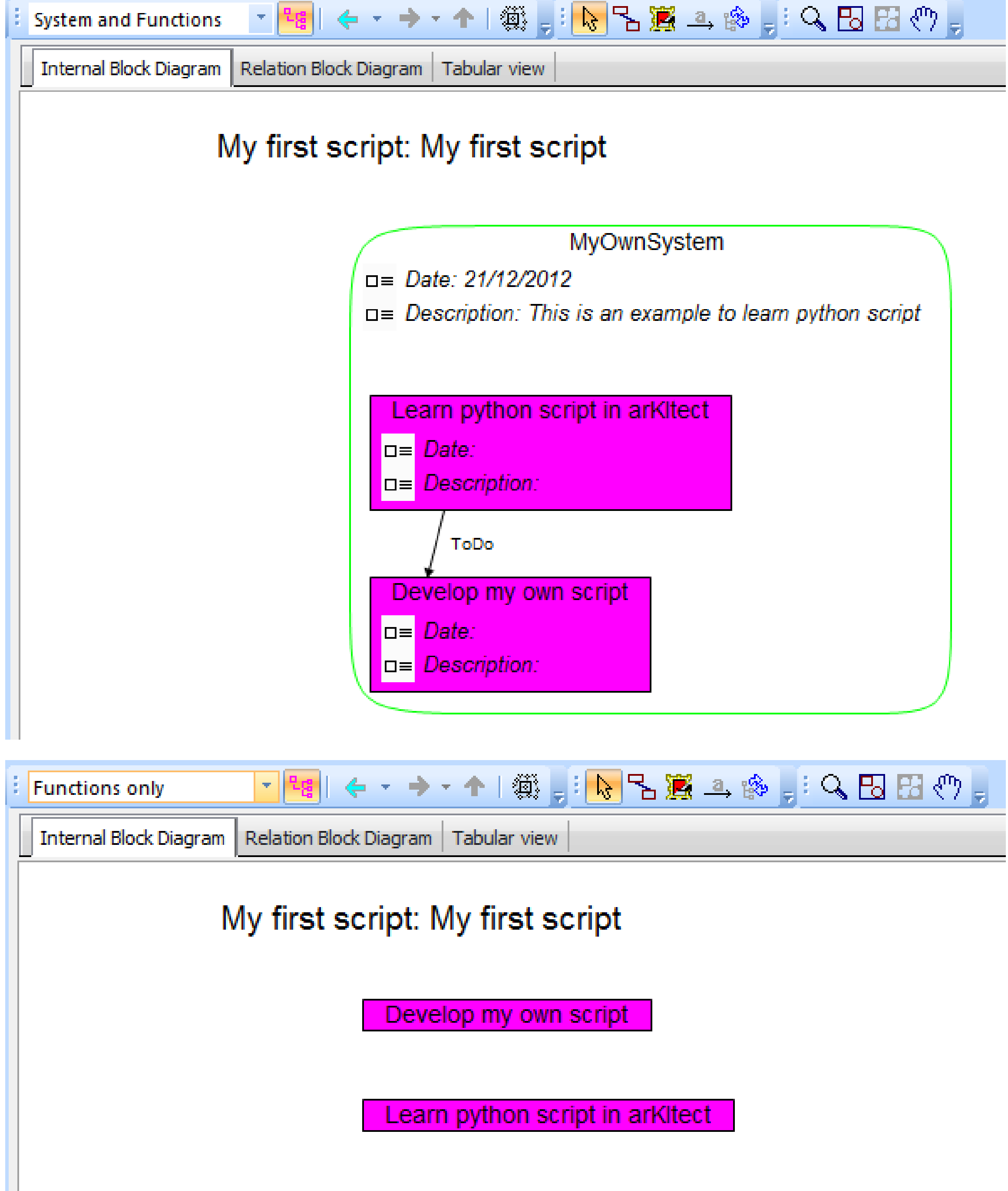
```
import pyark
def CreateTypes (self):
    #Get Reference of rule "System" and "Function" previously created
    system_type=pyark.GetArkMatrixType("System")
    function_type=pyark.GetArkMatrixType("Function")
    #Get Reference of filter "Functions only"
    functionsOnlyFilter=pyark.GetArkTreeViewObj("Functions only")
    #Check the rule "Function"

functionsOnlyFilter.CheckRule([pyark.GetArkMatrixType(self.GetArkType()),system_type,f
unction_type],True)
```



Add data in the project

We will try to add some data to check if our meta-model works fine.

```
import pyark
def CreateTypes (self):
    #Get root in view "System and Functions"
    systemAndFunctions=pyark.GetRoot("System and Functions")

    #Create a system and set attributes values
    system=systemAndFunctions.AddChildObject("System", "MyOwnSystem")
    system.GetAttribute("Date").SetValue("21/12/2012")
    system.GetAttribute("Description").SetValue("This is an example to learn python
script")
    #Create two functions under system
    f1=system.AddChildObject("Function", "Learn python script in arKItect")
    f2=system.AddChildObject("Function", "Develop my own script")
    #Create a flow to link both functions
    f1.AddChildObject("Flow","ToDo","output")
    f2.AddChildObject("Flow","ToDo","input")
```



Do the same actions in only one script

You can write all previous scripts in only one script:

```
import pyark
def CreateTypes (self):
    # creating base type called System above current Type
    #in this examples current type = top level project
    current_object = self
    current_type=pyark.GetArkMatrixType(current_object.GetArkType())
    system_type=current_type.AddRule("System")
    #Creating Type "Function" above type "System"
    function_type=system_type.AddRule("Function")
    #Creating type "Flow" flow exchanged by instance of "Functions"
    function_type.AddRule("Flow",pyark.ARK_RULE_FLOW)

    #Adding attributes to "System"
    system_type.AddAttribute("Description",pyark.ARK_ATTRIB_MEMO)
    system_type.AddAttribute("Date",pyark.ARK_ATTRIB_DATE)

    #Description and Date have now been created, they can be used and added to any
types
    # Adding them to type "Function"
    for attribute in system_type.GetAttributeList():
        function_type.AddAttribute(attribute)
        function_type.AddAttribute(attribute)

    #Changing properties of "System"
    system_type.SetProperty(pyark.ARK_ATYPE_OBJSHAPE,pyark.ARK_OBJSHAPE_RECT_RE)
    system_type.SetProperty(pyark.ARK_ATYPE_CLRLINE,(0,255,0))
    system_type.FlashProperties() #Write definitly properties before this line

    #Changing properties of "Function"
    function_type.SetProperty(pyark.ARK_ATYPE_CLRFILL,(255,0,255))
    function_type.FlashProperties()

    #Getting the root tree
    rootTree=pyark.GetRootTreeViewObj()

    rootTree.NewArkTreeViewObj("System and Functions",True) #Create a new filter and
check all rules
    functionsOnlyFilter=rootTree.NewArkTreeViewObj("Functions only",False) #Create a
new filter but any rules are checked

    #Check the rule "Function"
    functionsOnlyFilter.CheckRule([current_type,system_type,function_type],True)

    #Get root in view "System and Functions"
    systemAndFunctions=pyark.GetRoot("System and Functions")

    #Create a system and set attributes values
    system=systemAndFunctions.AddChildObject("System", "MyOwnSystem")
    system.GetAttribute("Date").SetValue("21/12/2012")
    system.GetAttribute("Description").SetValue("This is an example to learn python
script")
    #Create two functions under system
    f1=system.AddChildObject("Function", "Learn python script in arKItect")
    f2=system.AddChildObject("Function", "Develop my own script")
    #Create a flow to link both functions
    f1.AddChildObject("Flow","ToDo","output")
    f2.AddChildObject("Flow","ToDo","input")
```

schedule automatic scripts launching

Together with the Windows scheduler, unattended mode in arKItect can be used for running some tasks periodically. The usage is straightforward. Below is the sample,

Example: Periodically export architecture data

This example shows how to use scheduler to periodically export some specific architecture to an ARKZ file.

Prepare the script

First, prepare the script that would be executed by a scheduler. Below is a sample script, that exports current architecture in a file, named as `<Architecture Name>-<current date>.arkz`. Save this script to some folder, in this tutorial the folder `d:\arKItect\ScheduledScripts` is assumed. Any folder can be used.

export_architecture_data.py

```
from datetime import date
from os.path import join
from arki.features.impexp import arkz
import pyark

### Configuration section ###
output_folder = r"d:\arKItect\archives"

### Script section ###
arch_name = pyark.GetArkMatrixType().GetName()
file_name = "%s-%s.arkz"%(arch_name, date.today().isoformat())
arkz.export(join(output_folder, file_name))
```

Scheduling the script

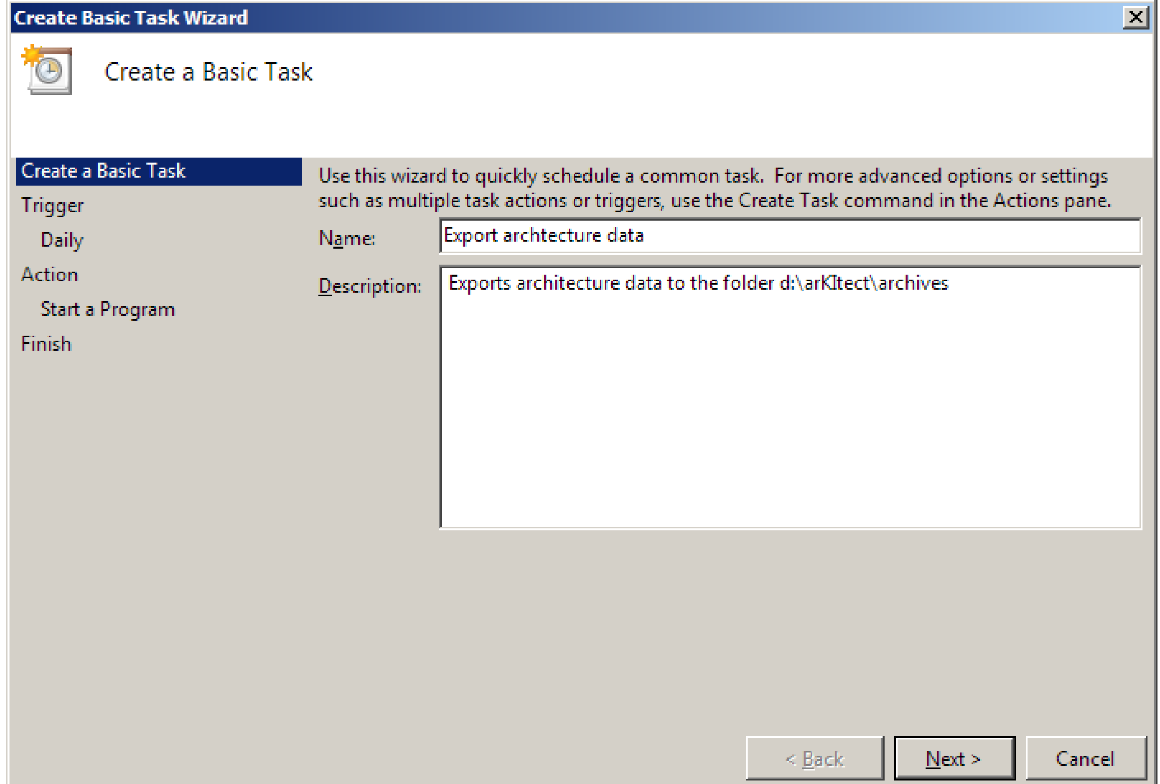
In the Windows scheduler, create a new task, that executes this script in unattended mode. The command line for running the script is:

Comman line

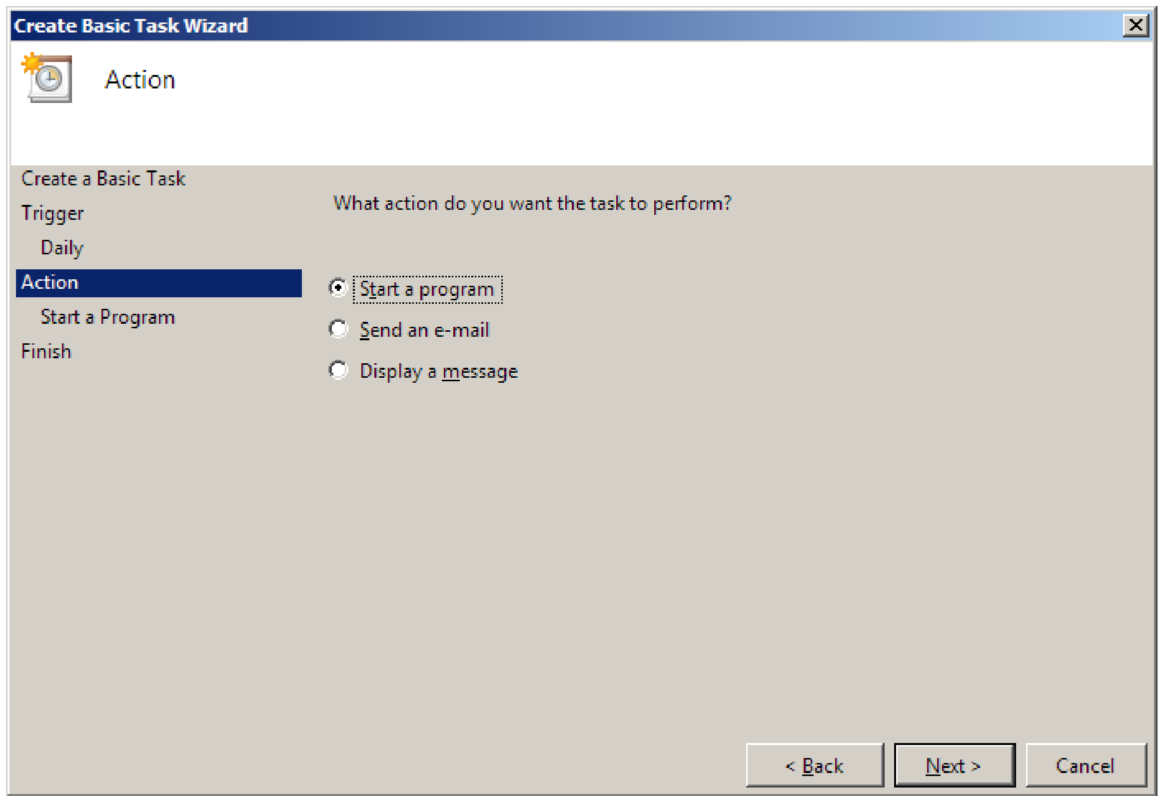
```
"C:\Program Files\arKItect\arKItect.exe" --login=user --password=password
--workspace="WS Name" --open-arch="arch name"
--run=d:\arKItect\ScheduledScripts\export_architecture_data.py
--log=d:\arKItect\ScheduledScripts\export_architecture_data.log
```

This command line runs script, saved to `d:\arKItect\ScheduledScripts\export_architecture_data.py`, and writes output to the file `d:\arKItect\ScheduledScripts\export_architecture_data.log`. The example uses "Basic Task" wizard.

Create a new task



Select an action: "Start a Program"



Configure unattended mode parameters

Specify arKItect executable in the "Program/Script" field (double quotes are only required, if path contains spaces)

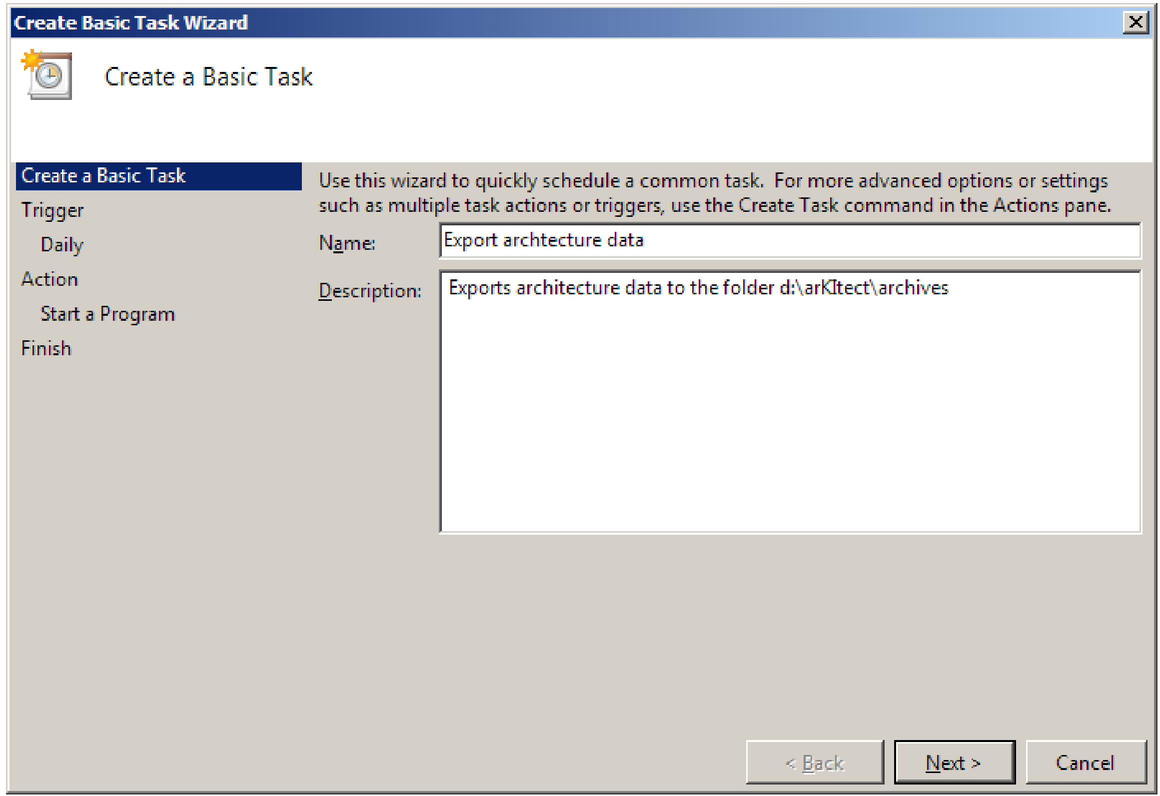
Program/Script field

```
"C:\Program Files\arKItect\arKItect.exe"
```

And put arguments to the "Add arguments" field ()

Arguments

```
--login=user --password=password --workspace="WS Name" --open-arch="arch name"
--run=d:\arKItect\ScheduledScripts\export_architecture_data.py
--log=d:\arKItect\ScheduledScripts\export_architecture_data.log
```



Configuration done.

Troubleshooting

Ensure that parameters with spaces (logins, workspace and project names) are enclosed in double quotes. In the provided example, a log file `d:\arKItect\ScheduledScripts\export_architecture_data.log` is configured. arKitect will write error messages and messages, printed during the script execution to this file.

Using External Python Libraries

This is a short tutorial which demonstrate how you can install and use external python libraries in **arKItect**.

Compatibility issue

All **arKItect** versions starting from 4.0.1 are compiled using MS Visual Studio 2015 compiler (vc14), the same is true for Python, boost, Stingray and other libraries shipped with **arKItect**.

If Python 2.7 is installed on your PC (independently from **arKItect**) it is highly probable that it is installed pre-compiled - using MS Visual Studio 2008 compiler (vc9). All libraries downloaded and installed by you under Python will also be either pre-compiled (as Python itself) or compiled in-place using the pre-compiled Python installed before.

These two compilations (embedded Python in **arKItect** and stand-alone Python at your PC) are incompatible and cannot be mixed. The same is true with respect to any external python libraries.

If the external library needed by you is not shipped with **arKItect** then it should be compiled using the same compiler (vc14 or later). There is one exception - no need to compile libraries which don't contain any source code in C/C++ (see 2 examples below)

Let's study 2 examples:

1. numpy-1.16.0-cp27-cp27m-win32.whl - this is a package for **numpy**
   a. cp27-cp27m - means that the package is for CPython 2.7
   b. win32 - means that the library is used by 32 bit Python (and this means that recompilation is needed - there is some code in C/C++)
2. pyparsing-2.4.6-py2.py3-none-any - this is a package for **pyparsing**
   a. py2.py3 - means that the package is compatible with CPython 2 and CPython 3
   b. none-any - means that there is no dependancy on the plateform so no need to recompile the library

To summarize, when you need an external Python library to be used with **arKItect**

- first you verify whether the library requires re-compilation (usually the presence of pyd files directly tells about it as well as the presence of any *.h, *.c or *.cpp file)
  - if yes - you recompile the library (or ask us to do that)
  - if no - you may use the library directly with arKitect (if it's Python27 compatible)

Deprecated

Library installation - can be used only with arKItect prior to v4.0.1

The simplest way to install new Python libraries is to install Python interpreter separately and then install the libs on this interpreter. As **arKItect** currently uses the version 2.7 of Python, you have to install the same major version otherwise **arKItect** will not be able to use the libraries you will install. Check this page to retrieve the latest 2.7.x version.

When retrieving the libraries you want to install, be careful to download the version for Python 2.7 (most libraries will be available for several Python versions). With the Python interpreter installed, you can use the normal installer of the libraries and **arKItect** Python scripts will automatically get access to them.

Known issues

Graphic interface problems

The default graphic interface which is provided with Python (TkInter) is known to have major issues when used within **arKItect**. Thus if possible we recommend to use a different graphic interface framework such as PyQt.

Let's take the example of matplotlib . By default the matplotlib library uses TkInter as its back-end which unfortunately results in crashing or hanging **arKItect**. To avoid those problem you can install the PyQt4 library and tell matplotlib to use it as its back-end with the following code:

```
import matplotlib
matplotlib.use('Qt4Agg')
```

Python API

Using Python scripts in arKItect

arKItect has support for scripting in Python (version 2.7). Python programs are stored in the attributes of type "Program". Such attribute must contain either simple script, or exactly one definition of a global function with one parameter, for example:

```
import pyark
#importing arKItect-specific  definitions
def run(self):
    print "Hello, script object is:", self
```

or

```
import pyark
print "Hello"
```

In the above example, function **run** is defined. When the script in the attribute is executed, this function receives a pointer (of type **CArkiObjPtr**) to the object it belongs to. Please note that importing functions from other modules also creates function definition, which can cause problems. In the script attributes, either import whole modules, or import functions inside

One script can contain only one function, but you can define inner functions inside it:

```
import pyark
#importing arKItect-specific  definitions
def run(self):
    def func1(a,b):
        pass

    def func2(c,d):
        pass

    pass
```

Functions, defined in the different attributes are not visible to each other.

What can you do with scripts ?

Almost any action available through the user interface is available through scripts. You can work with data, project, types and treeviews. Anyway, what you can really do relies on what you are allowed to do. For example, if you access a project in read only, any API to modify the project won't work. If you use **arKItect Developer**, all the modifications of the types or filters won't work, etc.

Using external libraries

If the script is too big to be conveniently used inside attribute, its code can be moved to the external Python library (file with *.py extension). This file later can be imported, using **import** directive. Such library should be placed to one of the folders **ArkiScripts** or **PythonLib**, or any other folder, present in the Python module search path.

Example:

```
#script inside arKItect
import pyark
import my_library #importing functions from external file "my_library.py"
def run(self):
    my_library.ComplexProcessing(self) #call a function from the external library
```

Note: Because Python scripts in arKItect must contain only one function declaration, the syntax **from library import function** or **from library**

import is not supported outside of function declaration.

Using libraries from Python installation

By default, arKItect comes with whole Python standard library and several additional libraries, including PyWin32 and PyQt. If Python 2.7 is installed on the computer, then any installed library can also be used in arKItect.

Available APIs

- Global functions and constants
- ArkAttribute objects
- ArkAttributeType objects
- ArkChoice objects
- ArkImage objects
- ArkMatrixType objects
- ArkObj objects
- ArkObjRef objects
- ArkPhase objects
- ArkProject objects
- ArkTreeViewObj objects
- ArkSimulinkLib objects
- ArkVariant objects
- ArkVariantFilter objects
- ArkConnection Objects
- ArkWorkspace Objects
- ArkUser objects
- ArkGroup objects
- ArkTab objects
- ArkBusyDialog objects
- Custom View Filters

Global functions and constants

- String validation policies
- Object management functions
- Types management functions
- Filters (tree views) management functions
- Getting and setting current context
- History management functions
- Images management functions
- Variants management functions
- Projects management functions
- Querying and modifying current project information
- Connection management functions
- User interface management functions
- Send alert functions
- AUA management functions
- Querying user permissions functions
- Enable Reader Cache for heavy operations
- Setting requests grouping mode
- Logging and Reporting functions
- Shared files functions
- Other functions
- Global constants
    - Attributes
    - Diagrams
    - Type properties
    - Line shapes
    - Line styles

- Arrow styles
- Object shapes
- Simulink types
- Rule types
- Projection properties
- Projection master/secondary
- Object options/variants/phases status
- Diagram orientation
- Object parent relations
- Events
- User permissions
- Use access types
- String validation policies
- Message logging levels
- History action types

To make a call of a global function/constant user needs to:

- import a module where global function/constant is defined
- use the following syntax: ModuleName.GlobalFunction or ModuleName.GlobalConstant

Example:

```
import pyark
def run(self):
    pyark.GetRoot( "The Tree" ) #Using global function to get the tree root
```

String validation policies

arKItect imposes some limitations on names of objects, types, filters and some other string values. If script passes incorrect string, default behavior is to raise an exception. This can be adjusted, using the following functions:

- **GetStringValidationPolicy()**
  Gets currently active string validation policy. See **SetStringValidationPolicy** for list of possible values.
- **SetStringValidationPolicy(policy)**
  Sets current string validation policy. Argument can be on of the following:
    - **pyark.SVP_STRICT** - strict policy; exception is raised on invalid strings.
    - **pyark.SVP_IGNORE** - ignore invalid characters, no exception is raised.
    - **pyark.SVP_REPLACE** - replace invalid characters with "?".

Object management functions

- **GetRoot(tree_name)**
  Returns the root for the specified tree. Root is an instance of ArkObjRef object. Tree is specified by the parameter **tree_name** (**tree_name** maybe a key as well) which must be string. A ValueError exception is risen if a treeview folder is passed
- **GetArkObj(ark_type, obj_name)**
  Returns ArkObj instance, if object with such name and type is found in project. Otherwise, returns **None**.
- **GetArkObjById(obj_id)**
  Returns ArkObj instance by internal identifier, returned by the **GetID** method of ArkObj or ArkObjRef. If object with such identifier is not found, returns **None**.

- Advanced functions

Types management functions

- **GetArkMatrixType(name=None)**
  Returns the ArkMatrixType object identified by **name** (**name** maybe a key as well). If no parameter is passed, then the matrix root object is returned. If the object is not found return **None**.

- **GetArkAttributeType(name)**
  Returns the ArkAttributeType object identified by **name** (**name** maybe a key as well) or **None** if the object is not found.

- **GetArkMatrixTypesList()**
  Return a list of ArkMatrixType objects defined in the project.

- **GetArkAttributeTypesList()**
  Return a list of ArkAttributeType objects defined in the project.

Filters (tree views) management functions

- **GetArkTreeViewObj(name)**
  Returns the ArkTreeViewObj object identified by **name** (**name** maybe a key as well) or **None** if the object is not found.

- **GetRootTreeViewObj()**
  Returns the root ArkTreeViewObj object.

- **NewArkTreeViewObj(name, checkStatus)**

  Deprecated - use **NewArkTreeViewObj()** or **NewArkTreeViewFolder()** methods of ArkTreeViewObj instead

  Create a new ArkTreeViewObj object with the given **name** on the highest level of the treeview hierarchy. Set the visibility status of the all rules to **checkStatus**, where **checkStatus** is boolean.

- **GetTreeViewsList()**

  Deprecated - use **GetChildList()** of *ArkTreeViewObj* instead

  Return a list of ArkTreeViewObj objects available in the project.

Getting and setting current context

- **GetActiveView()**
  Get currently active object and view type.
  Returns tuple: (**ArkObjRef**, **GraphType**), where **ArkObjRef** is a currently selected object, and **GraphType** is identifier of the currently active graph (same as for functions **SaveViewAsImage**, **Get**/**SetGraphXml**). In case there is no active view (possible in some situations), **GetActiveView** returns **None**.

- **SetActiveView(object, graphType)**
  Set currently active object and view type; update window. Parameter **graphType** specivies type of the view, and can be one of:
  - ARK_GRAPH_INNER_VIEW: Internal block diagram.
  - ARK_GRAPH_PEER_VIEW: Relation block diagram.
  - ARK_GRAPH_MATDRAW_INNER_VIEW: Internal block diagram, shown in Matdraw view.
  - ARK_GRAPH_MATDRAW_PEER_VIEW: Relation block diagram, shown in Matdraw view
  - ARK_GRAPH_MATDRAW_VIEW: (deprecated) Current block diagram, shown in Matdraw view (inner or internal).

- **GetSelection()**
  Returns list of **ArkObjRef** objects, corresponding the current selection, if view type is Inner or Peer. For other views, returns empty list.

- **UpdateActiveView()**
  Updates currently active view. Modifications are shown in the view

- **SetActiveVariant(var)**
  Argument **var** is an ArkVariant object.
  Note that **SetActiveVariant** does not updates view immediately. To update view, when script is running, use **UpdateActiveView()** metho
  d.
- **SetActivePhase(ph)**
  Argument **ph** is an ArkPhase object.
  Note that **SetActivePhase** does not updates view immediately. To update view, when script is running, use **UpdateActiveView()** method
  .

- **GetActiveVariant()**
  Returns active variant. If not found, returns **None**.
- **GetActivePhase()**
  Returns active phase. If not found, returns **None**.

- **IsSnapToGrid()**
  Returns **True** if 'snap to grid' enabled

- **GetGridParameters()**
  Returns dictionary of grid parameters:

Key	Value type	Explanation
"visible"	boolean	True, if grid is visible
"snap"	boolean	True, if snapping is enabled
"angle_snap"	boolean	True, if angular (rotation) snapping is enabled
"spacing_x"	float	Horizontal snapping distance
"spacing_y"	float	Vertical snapping distance
"color"	(int, int, int)	Grid color in the **RGB** format
"show_margins"	boolean	True, if document margins are shown

History management functions

- **GetHistory(dateFrom, dateTo [, userNameFilter, hpIdFrom, hpIdTo])**
  This function retrieves history of objects in the project. The arguments are:
  - **dateFrom**, **dateTo** - date time objects that define the part of history being retrieved. These two arguments are mandatory. In order to use date time objects, you need to import **datetime** module (see example below) in your script.

    Other arguments are optional, they specify additional conditions for filtering history records. All these arguments are strings.
  - **userNameFilter**- If specified, returns only records, related to the specified user.
  - **hpIdFrom**, **hpIdTo** - if specified, returns only records within the range

  Return value is a list of tuples, each tuple representing separate history record, as visible in the *History dialog*. Each tuple consists of following 11 elements: (**hpId**, **userName**, **date**, **objectType**, **objectName**, **treeViewName**, **action**, **actionClass, historyValue, objectId, sessionId**), where **date** is a datetime type, and other elements are strings.

GetHistory returns not more than **1000 latest records** in the specified range. To get whole history, call **GetHistory** several times, setting **hpidTo** parameter to the hpId of the latest record.

This is implemented in the utility function **arki.utils.arkiutils.IterHistory**

Example calling *GetHistory*:

```
import pyark
import datetime
d1 =datetime.date(2015,12,1) #from 1 December 2015
d2 = datetime.datetime.now() #until now
user = 'user@k.i'
hpIdFrom = '230000'
hpIdTo = '240000'
hist = pyark.GetHistory(d1, d2)  #request history for the given period
#hist2 = pyark.GetHistory(None, d2, user, hpIdFrom, hpIdTo)  #request history for the
given user within given history points range
for h in hist:
    print h #printing all found history records
```

Images management functions

- **NewArkImage(name)**
  Returns new ArkImage object, referencing specified image. **name** must be string, identifying image in library. Use **GetAllImages()** global function. It is important to note that images in ArKItect are lazy-loading, i.e. actual image data are downloaded only when they are needed. Thus, call **NewArkImage(url)** newer fails and always returns object. To explicitly load image, use **Load()**.

- **GetAllImages()**
  Returns list of *ArkImage* objects, available in images library. Note that this function does not cause image data to download, only list of image names is retrieved.

- **UploadImage(path)**
  Uploads image from local file and returns corresponding ArkImage objects

Variants management functions

- **GetRootChoice()**
  Returns the root category for all choices. Returned object is of the type *ArkChoice*.

- **GetRootVariant()**
  Returns the root category for all variants. Returned object is of the type *ArkVariant*.
- **GetRootPhase()**
  Returns the root category for all phases. Returned object is of the type *ArkPhase*.

- **GetChoiceByName(name)**

  Deprecated - use **GetChoice(name)** method of *ArkChoice*

  Search for choice by its **name**. If not found, returns **None**.

- **GetVariantByName(name)**
  Search for variant by its **name**. Returned object is of the type ArkVariant. If not found, returns **None**.
- **GetPhaseByName(name)**
  Search for phase by its **name**. Returned object is of the type ArkPhase. If not found, returns **None**.

- **GetDefaultVariant()**
  Returns default variant as ArkVariant object. If not found, returns **None**.
- **GetDefaultPhase()**
  Returns default phase as ArkPhase object. If not found, returns **None**.

- **GetVariantList()**
  Returns a list of the ArkVariant objects, representing variants, defined for the project.
- **GetPhaseList()**
  Returns a list of the ArkPhase objects, representing phases, defined for the project.

- **DeleteVariant(name)**
  Deletes variant by its **name**.
- **DeletePhase(name)**
  Deletes phase by its **name**.

- **DeleteChoice(name)**

  Deprecated - use **DeleteChoice(name)** method of *ArkChoice*

  Deletes choice by its **name**.
- **VariantFilter(variant, phase = None)**
  Creates ArkVariantFilter object, that can be used for enabling or disabling variant or/and phase filters. Argument **variant** must be a valid (non-folder) ArkVariant object or None if no active variant is needed, argument **phase** must be a valid (non-folder) ArkPhase object or None if no active phase is needed.

Projects management functions

These functions allow script to query, create, delete and copy projects in the current workspace. For working with other workspaces, see

documentation for **pyark.OpenConnection** in the Connection management functions section.

- **GetProject(name)**
  Search for project by its **name**. If not found, raises a **ValueError**. Returned value is an instance of ArkProject object.

- **GetProjectsList( )**
  Returns a list of the ArkProject object, representing projects, defined in the current workspace.

- **DeleteProject(name)**
  Deletes project specified by its **name**.

- **CopyProject(name, newname = None, variant=None)**
  Copies project **name** to a new project in the same workspace. If specified, **newname** defines the name of the copy.
  Optional argument **variant** must be an instance of **ArkVariant** object. If specified, only part of the project, visible in the specified variant, is copied (currently implemented via copying whole project and removing invisible part)

- **NewProject(name, template_name = None)**
  Creates a new project **name** in the same workspace. If specified, **template_name** points to the template which should be used.
- **CreateRevision()**
  Create copy of the current project into **REVISIONS** Workspace.
- **RestoreRevision(revName, newName)**
  Create new project with name **newName** as a copy of the project **revName** from **REVISIONS** Workspace.
- **SetProjectDeletable(status)**
  Enables/disables the Project protection from deletion. Set status to False to enable the protection from deletion or True to disable it.
- **IsProjectDeletable()**
  Returns True if the Project can be deleted (protection is not set), False otherwise.

Querying and modifying current project information

Functions that return information about the state of the currently open project and allow to modify it.

- **GetProjectName()**
  Returns name of the current project as string. New in 4.0.3.
- **SetProjectName(newName)**
  Renames current project. If new name is invalid (empty or contains forbidden characters); or such project already exists in the workspace, ValueError exception is raised. If new name is the same as current, method does nothing. New in 4.0.3.

- Advanced functions

Connection management functions

These functions allow to open connection, accessing projects in other servers and workspaces.

- **OpenConnection(login, password, httpsOnly=False)**
  Returns new ArkConnection object, corresponding to the given login and password, or raises a **ValueError**, if login or password are invalid. By default, attempt to establish HTTPS connection is made first. if it is failed, then HTTP connection attempt is done. If optional flag **httpsOnly** is True, HTTP connection attempt is not done and exception is raised immediately.
- **GetCurrentConnection(httpsOnly=False)**
  Same as **OpenConnection** with login and password of the current user.

User interface management functions

These functions allow the user to customize the graphical interface of the arKItect client. It relies on the PyQt integration technology. An example of PyQt integration within the arKItect client can be found here
The following functions are defined in a submodule of pyark named 'ui'. This means that in order to use them, you have to use the prefix 'pyark.ui' and not just 'pyark'.

- **NewArkTab(name)**
  Creates a new tab in the arKItect's main frame with the caption **name** and set it as the current active tab. A new ArkTab object is returned to be able to manipulate the newly created tab.
  If a tab with the given **name** already exists, a **ValueError** will be raised.

- **RemoveTab(tab)**
  Removes a tab from the the arKItect's main frame. **tab** can either be an ArkTab object or the name of an existing tab.

- **GetTabList()**
  Returns a list of every ArkTab object present in the main frame.

- **GetTab(tabName)**
  Returns the ArkTab object with the given **tabName** if it exists, returns None otherwise.
- **MessageBox(text)**
  Shows simple popup message with text and OK button.

Other functions to control arKItect GUI from scripts:

- **ui.ShowMultipleRevisionsDialog()**
  Opens a dialog, showing existing object revisions (available from GUI menu as Tools / Show Objects Under Revisions).

- Advanced functions

Send alert functions

- **GetUserName()**
  Returns the name of the current user. Can be used in the send-to address.

- **GetUsers()**
  Return a list of names of all users, who have access to the project.

- **SendAlert(message, send_to = None, subject = "arKItect alert", alert_url)**
  Send an alert message.
  Arguments:
  - **message**: message text.
  - **send_to**: Name (not e-mail) of the user to receive the alert. Several names can be specified in a list. If None is passed, message is sent to all users having access to the current project, i.e. the resulting list of **GetUsers()** method is used.
  - **subject**: Subject of the alert message. Default is "arKItect alert"
  - **alert_url**: Url of the alert. If message is sent in the text format, this parameter is ignored. If HTML message is sent, this url is attached to the message. To generate this parameter properly use **ArkObjRef.GetURL(graphType)** method.

AUA management functions

- **CreateGroup(name)**
  Creates a user group identified by **name**. Raises **ValueError**, if group name is incorrect, or already used. Returns ArkGroup object.
- **DeleteGroup(name)**
  Removes the group identified by **name**. Returns **None**. Raises **ValueError** if given name is empty string, or user has no right to delete group.
- **InitGroup(name)**
  Returns the ArkGroup object identified by **name**. Raises **ValueError** if group with given doesn't exist or name is empty.
- **InitUser(name)**
  Returns the ArkUser object identified by **name**. Raises **ValueError** if user with given doesn't exist or name is empty.
- **GetUsers( )**
  Returns list of user names of users that are registered in this workspace.

Querying user permissions functions

- **CanWriteMatrix()**
  Returns True if Matrix has write access.
- **CanWriteTreeview()**
  Returns True if Treeview has write access.
- **HasSimulinkRights()**
  Returns True if current user has license for Simulink module, False otherwise
- **GetUserPrivileges(userName = <current user name>)**
  Returns status of the current user if no parameter is passed or status of user with **userName.** Returns one of **pyark.UserPrivilege (READONLY, USER, COORDINATOR, SUPERCORE, GROUP, TESTER)**. If user with such name not found, returns value **USER**. User name must end with "@k.i".

Enable Reader Cache for heavy operations

To improve performance on different reading actions when they are done multiple times - e.g. getting all objects and then checking for each object whether it is a flow or getting attributes for each object - it is possible to enable data reader cache in **arKItect**.

- **_EnableDataReaderCache(enable)**
  Enables / disables data reader cache. User do not need to call this function, it is used internally in **DataReaderCacheContext** context manager

DataReaderCacheContext should be used to parse the model (walk_objects, GetAllChildren, IsChildFlow, GetInputFlowList / GetOutputFlowList ...)

Setting requests grouping mode

To improve performance of scripts which create a lot of objects, query packaging mode can be enabled in **arKItect**. In this mode, object creation and modification queries are grouped into single packets, which significantly improves performance. However, functionality of **arKItect** is seriously limited in this mode: basically, only object creation and modification methods are working. The following functions can be used to control this mode:

- **EnablePacketNetworkInterface()**
  Enables grouping of queries. Currently, user do not need to call this function, it is used internally in the **BulkModeContext** context manager.
- **DisablePacketNetworkInterface()**
  Disables special network interface. Currently, user do not need to call this function, it is used internally in the **BulkModeContext** context manager.
- **IsBulkModeEnabled()**
  Returns **True**, if grouping of queries (bulk mode) is currently active. Returns **False** otherwise.
- **FlushAccumulatedData()**
  When network packeting mode is enabled (see **arki.utils.arkinternal.BulkModeContext**), forces arKitect to commit all currently accumulated requests to server without interrupting packeting mode. Causes all modifications to be written in the database and all possible conflicts accounted. Currently, user do not need to call this function, it is used internally in the **BulkModeContext** context manager.

BulkModeContext should be used when updating several object

A good practice is to split reader and setter task when it is possible and use **DataReaderCacheContext** for reading tasks and **BulkModeContext** for writing tasks

Sometimes when a script is being executed, it is needed to visualize intermediate results (update diagrams) - to do so a special method is introduced:

- **Flush()**
  Updates GUI during script execution without interrupting the script

Logging and Reporting functions

Scripts might need to give user some information about details of the execution: information messages, warnings, recoverable errors and other. It is possible to print this information to the "Program Trace" window, using Python's "print" operator. However, their use is discouraged in production scripts because of the limited features of this approach. Instead, developers are encouraged to use "Pythons Events" system.

There are 2 API functions that control it:

- **LogEvent(level, code, module, title_description_callback)**
  Adds new event to the log.
- **SignalLogWindow()**
  Checks conditions for showing "Python Events" window, and shows it, if conditions were satisfied. In the latter case, the method returns only when the window is closed. Normally, shows this window at the end of script execution, this method allows to show it before terminating script. "Python Events" window shows a log of events, generated by **pyark.LogEvent** method and uncaught exceptions.

Each event is characterized by 6 parameters:

- Time: when the event was reported.
- Level: one of **pyark.LOG_XXX** constants. Specifies importance of the message for the user, from lowest (**LOG_DEBUG**) to the highest (**LOG_CRITICAL**). Possible values are:
  1. **LOG_DEBUG:** use it for debug messages that are not intended to be viewed by the regular users.

2. **LOG_INFO**: messages for the users that do no bear any critical information (script started / script finished).
3. **LOG_WARNING**: messages that warn user about some possible problems.
4. **LOG_ERROR**: information about recoverable errors (something failed, but script can continue work)
5. **LOG_EXCEPTION**: information about Python exceptions. These events are are added automatically, if uncaught exception leaked to arKitect.
6. **LOG_CRITICAL**: critical errors, that prevent further script execution.

- Error code: arbitrary integer value. Can be used to distinguish between different errors types, its value is up to developer.
- Module: string, indicating what module reported the event.
- Title: short, one-line description of the problem. Shown in the event list.
- Description: long, multiline description of the problem details (stack trace, list of files, etc).

User can define, what levels of events to ignore and what to store in the memory; it is also possible to configure, when to show pop-up window with event log messages.

Generally, generating lots of long descriptions may cause significant performance degradation. To avoid this problem, title and description generation are **delayed**. Instead of providing 2 strings, developer must provide a callback function that produces a tuple of 2 strings: **(title, description)**. This callable is executed only if the event is not ignored by the current logging settings. Thus, ignored messages are never generated, and don't waste resources.

Example of a code, that adds a message:

```
pyark.LogEvent( pyark.LOG_WARNING, 0, "important script",
                lambda: ("Warning title", "Warning description") )
```

This code uses lambda expression to make a callback function, but any other kind of callable is appropriate.

Shared files functions

Shared files are stored together on the server side in the project shared storage, under the SVN ontrol. The storage is unique for each project. Shared files are commonly associated to objects.

Note about revision numbers: current latest revision is incremented each time someone updates file data on server.

- **UploadSharedFile(file_path, comment (optional))**
  Uploads a file to the "shared" storage of the project. Returns resulting revision number of the uploaded file.
  Arguments:
  - **file_path**: string, path to the file. If file is not readable or does not exists, method raises **ValueError**.
  - **comment**: string, optional. If not specified, default comment will be written.

  File update rules, applied when shared storage already have file with such name:
  - If uploaded file is the same as existing, function does nothing. Last revision of the existing file is returned, comment is ignored.
  - If uploaded file differs, new revision is created and new comment is written.
- **DownloadSharedFile(file_name, [save_path], [revision])**
  Downloads a file with given name from the project shared storage. Raises **ValueError**, if file_name is incorrect.
  Arguments:
  - **file_name**: string, name of the file in shared storage.
  - **save_path**: string, optional. Path where to save file.
  - **revision**: string, optional. Revision to download. Default is latest revision.

  Return value: tuple **(file_path, revision)**. Path to the file on local computer and actual revision number. If prior to function call file with such path already existed, it is overwritten.
- **ListSharedFolderFiles( )**
  Returns a list of files in the project shared storage in XML format (string). Returns empty string, if shared storage is not initialized yet for this project (should be treated as empty list). XML format is svn XML output.
- **RemoveSharedFile(file_name)**
  Removes the file with given name from the project shared storage. Raises **ValueError**, if there is no such file. Returns **None**.

Other functions

- **GetAppDataFolder()**
  Returns path to Aplication Data Directory. If failed, raises a RuntimeError.
- **GetProjectAddress()**
  Returns technical data about connection: workspace ID, business space ID, architecture ID, data server address.

- **LoadAcceptedProjectState()**
  in context of RACI feature, loads all accepted data, all arKItect actions related to 'not accepted requests' became undone in memory.
- **GetRevisionImpactingObjectList([ArkObj])**
  Returns the list of objects with stable revision state impacted by any of provided objects (impact - provided objects are used in some wiki attribute). Data is prepared on the server side.
- **GetRevisionPossibleImpactingObjectList([ArkObj])**
  Returns the list of objects impacted by any of provided objects (impact - provided objects are used in some wiki attribute) in arKItect memory.
- **GetActiveUsers()**
  Returns the list of active users working in opened project (active **arKItect**sessions). Returns the list of pairs : user name + arkObjRef . arkObjRef object allows to understand the treeview and 'view path' for active user.

- **GetVersion()**
  Returns version of the **arKItect** as a 4-tuple of type **(int, int, int, string)**.
  Numeric part of this tuple represents first 3 version numbers from the version string. All characters after the first 3 numbers, or any non-numeric characters are returned in the 4th element, as string. If version string has only 1 or 2 numbers, zeros are added.

Examples:

arKItect version	**pyark.GetVersion()**
"2.1"	(2,1,0,"")
"2.1alpha"	(2,1,0,"alpha")
"2.1.1"	(2,1,1,"")
"2.1.1.2009"	(2,1,1,"2009")
"2.alpha2.3"	(2,0,0,"alpha2.3")

Global constants

Attributes

ARK_ATTRIB_APP, ARK_ATTRIB_AUTHOR, ARK_ATTRIB_SIMULINK_CONFIG, ARK_ATTRIB_BOOL, ARK_ATTRIB_DATE, ARK_ATTRIB_DOUBLE, ARK_ATTRIB_ENUM, ARK_ATTRIB_ENUM_VALUE, ARK_ATTRIB_FILE, ARK_ATTRIB_GROUP, ARK_ATTRIB_HYPERLINK, ARK_ATTRIB_INT, ARK_ATTRIB_MEMO, ARK_ATTRIB_RICHTEXT, ARK_ATTRIB_POINTERS, ARK_ATTRIB_PROGRAM, ARK_ATTRIB_SAVEDFILE, ARK_ATTRIB_SINGLE, ARK_ATTRIB_TEXT, ARK_ATTRIB_TEXTLARGE, ARK_ATTRIB_MARKUP

Diagrams

ARK_GRAPH_INNER_VIEW, ARK_GRAPH_PEER_VIEW, ARK_GRAPH_MATDRAW_VIEW, ARK_GRAPH_MATDRAW_INNER_VIEW, ARK_GRAPH_MATDRAW_PEER_VIEW, ARK_GRAPH_TABULAR_VIEW

Type properties

ARK_ATYPE_CLRTEXT, ARK_ATYPE_CLRLINE, ARK_ATYPE_CLRFILL, ARK_ATYPE_CLRFLOW, ARK_ATYPE_CLRFLOW_TEXT, ARK_ATYPE_CLRFILLBG, ARK_ATYPE_CLRTEXT_TRANSP, ARK_ATYPE_CLRTEXTFLOW_TRANSP, ARK_ATYPE_CLRLINE_TRANSP, ARK_ATYPE_CLRFILL_TRANSP, ARK_ATYPE_CLRFLOW_TRANSP, ARK_ATYPE_CLRFILLBG_TRANSP, ARK_ATYPE_LINESHAPE, ARK_ATYPE_LINESTYLE, ARK_ATYPE_LINEWIDTH, ARK_ATYPE_BORDERSTYLE, ARK_ATYPE_BORDERWIDTH, ARK_ATYPE_ARROWSTYLE, ARK_ATYPE_OBJSHAPE, ARK_ATYPE_SIMTYPE, ARK_ATYPE_ICON, ARK_ATYPE_DISPLAY_ATTRIB_NAME, ARK_ATYPE_DISPLAY_ATTRIB_ICON, ARK_ATYPE_DISPLAY_ATTRIB_VALUE, ARK_ATYPE_FILLHATCH, ARK_ATYPE_KEEP_RATIO, ARK_ATYPE_DEFAULT_SIZE, ARK_ATYPE_READ_ONLYARK_ATYPE_EXCLUDE_FROM_REVMNG, ARK_ATYPE_UNIQUE, ARK_ATYPE_FLEXIBLE

ATYPE_GEN_TAG_ATRRIB_BGIMAGE, ATYPE_GEN_TAG_ATRRIB_FGIMAGE, ATYPE_GEN_TAG_ATRRIB_FONT, ATYPE_GEN_TAG_ATRRIB_FONT_FLOW

Line shapes

ARK_LINESHAPE_CURVE, ARK_LINESHAPE_STRAIGHT, ARK_LINESHAPE_ORTHOGONAL, ARK_LINESHAPE_CUSTOMIZABLE

Line styles

ARK_LINESTYLE_SOLID, ARK_LINESTYLE_DASH, ARK_LINESTYLE_DOTARK_LINESTYLE_DASHDOT, ARK_LINESTYLE_DASHDOTDOT

Arrow styles

ARK_ARROWSTYLE_NOARROW, ARK_ARROWSTYLE_ARROW, ARK_ARROWSTYLE_OPEN,
ARK_ARROWSTYLE_STEALTH, ARK_ARROWSTYLE_DIAMOND, ARK_ARROWSTYLE_DIAMOND_WHITEARK_ARROWSTYLE_CIRCLE

Object shapes

ARK_OBJSHAPE_AND, ARK_OBJSHAPE_COMMENT, ARK_OBJSHAPE_CROSS, ARK_OBJSHAPE_CUBE, ARK_OBJSHAPE_DIAMOND,
ARK_OBJSHAPE_DOC, ARK_OBJSHAPE_ELLIPSE, ARK_OBJSHAPE_FLAG, ARK_OBJSHAPE_METEOR,ARK_OBJSHAPE_OR,
ARK_OBJSHAPE_PLAQUE, ARK_OBJSHAPE_PRLGRM, ARK_OBJSHAPE_RECT, ARK_OBJSHAPE_RECT_RE, ARK_OBJSHAPE_SAIL,
ARK_OBJSHAPE_SIGN_LEFT, ARK_OBJSHAPE_SIGN_RIGHT, ARK_OBJSHAPE_TEARDROP, ARK_OBJSHAPE_TRAPEZ_UP,
ARK_OBJSHAPE_TRAPEZ_DOWN, ARK_OBJSHAPE_TRIANGLE_UP, ARK_OBJSHAPE_TRIANGLE_DOWN, ARK_OBJSHAPE_WAVE,
ARK_OBJSHAPE_PENTAGON, ARK_OBJSHAPE_HEXAGON, ARK_OBJSHAPE_HEPTAGON, ARK_OBJSHAPE_OCTAGON,
ARK_OBJSHAPE_DECAGON, ARK_OBJSHAPE_DODECAGON, ARK_OBJSHAPE_CAN, ARK_OBJSHAPE_CAN_LYING,
ARK_OBJSHAPE_END_NODE, ARK_OBJSHAPE_INIT_NODE, ARK_OBJSHAPE_CROSS_NODE, ARK_OBJSHAPE_JOIN,
ARK_OBJSHAPE_BIFURCATION, ARK_OBJSHAPE_BIFURCATIONV, ARK_OBJSHAPE_PRIORITY_AND_GATE,
ARK_OBJSHAPE_HOUSE_EVENT, ARK_OBJSHAPE_XOR, ARK_OBJSHAPE_INHIBIT_GATE, ARK_OBJSHAPE_SIMULINK

Simulink types

ARK_SIMTYPE_ENABLE, ARK_SIMTYPE_FCALL, ARK_SIMTYPE_SM, ARK_SIMTYPE_SUBSYSTEM, ARK_SIMTYPE_STATE,
ARK_SIMTYPE_SIGNAL, ARK_SIMTYPE_TRANSITION, ARK_SIMTYPE_TRIGGER, ARK_SIMTYPE_VECTOR

Rule types

ARK_RULE_SIMPLE, ARK_RULE_FLOW, ARK_RULE_BIFLOW, ARK_RULE_INPUT, ARK_RULE_OUTPUT

Projection properties

ARK_TREEVIEW_SIMTYPE, ARK_TREEVIEW_DISPNAME, ARK_TREEVIEW_MASTERSECONDARY, ARK_TREEVIEW_GRAPHORIENT

Projection master/secondary

ARK_TREEVIEW_MASTER, ARK_TREEVIEW_SECONDARY

Object options/variants/phases status

ARK_OBJVAR_UNDEFINED, ARK_OBJVAR_MANUAL, ARK_OBJVAR_PROPAGATED, ARK_OBJVAR_PROPAGATED_UP

Diagram orientation

ARK_GRAPHORIENT_STANDARD, ARK_GRAPHORIENT_TB, ARK_GRAPHORIENT_LR

Object parent relations

RELATION_NONE, RELATION_DIRECT_CHILD, RELATION_INDIRECT_CHILD, RELATION_ATTRIBUTE

Events

EVENT_ONADDCHILD, EVENT_ONADDEXISTING, EVENT_ONADDNEW, EVENT_ONCHANGEATTRIBVALUE,
EVENT_ONCHANGEDIRECTION, EVENT_ONCLOSE, EVENT_ONCUSTOMFILTER, EVENT_ONDELETE, EVENT_ONOPEN,
EVENT_ONREMOVECHILD, EVENT_ONRENAME, EVENT_ONACTIVATEIBD, EVENT_ONCHANGEPROPERTIES,
EVENT_ONCHANGEPROJECTION, EVENT_ONADDCHILD_GLOBAL, EVENT_ONREMOVECHILD_GLOBAL

User permissions

STATUS_DEVELOPER, STATUS_DEVELOPER_ADMIN, STATUS_DESIGNER, STATUS_READONLY

Use access types

RIGHTS_NOT_VISIBLE, RIGHTS_READ_ONLY, RIGHTS_FULL_ACCESS

String validation policies

SVP_STRICT, SVP_IGNORE, SVP_REPLACE

Message logging levels

LOG_INFO, LOG_DEBUG, LOG_WARNING, LOG_ERROR, LOG_EXCEPTION, LOG_CRITICAL

History action types

HISTORY_OBJECT_ATTRIBUTES, HISTORY_OBJECT_ATTRIBUTES_REVISION_EXCLUDED, HISTORY_OBJECT_PARENTSTRUCTURE, HISTORY_OBJECT_OPTIONS, HISTORY_OBJECT_RENAME, HISTORY_ADD_NEW_OBJECT, HISTORY_DELETE_REFERENCE, HISTORY_DELETE_OBJECT_GLOBALLY, HISTORY_LINK_DIRECTION, HISTORY_NEW_OBJECT_REFERENCE, HISTORY_PROJECT_ACTION, HISTORY_OBJECT_CREATE_REVISION, HISTORY_OBJECT_SWITCH_REVISION,HISTORY_SCRIPT

Advanced functions

Object management functions

- **ReloadObjects()**
  Reloads project's data. Useful after exiting LocalScriptContext so that memory model corresponds to the real project data. Could not be called under Bulk Mode / Local script context, the exception raised otherwise.
- **GetAllArkObjs()**
  Returns a list of all objects (as ArkObj) defined in the project

- **SubstituteObjects(substitutions, copyChildren=True)**
  where arguments are:
  - **substitutions**: list of tuples of ArkObjRef: [(original, substitution), ...]. Can be empty (method does nothing in this case)
  - **copyChildren**: boolean, optional; if **True**, children of the "original" object are copied to the "substitution" object. Otherwise, they are lost.

  Conditions:
  - Original and substitution objects must have the same type.
  - The sets of original and substitution objects must not intersect.
  - User must have write access to the objects.

  If these conditions are not fulfilled, exception is raised.
  Additional notes:
  - This API method does not check for possible cycle formation.
  - View graph information is not updated; new objects will have different positions.
- **RemoveObjectsGlobally(objects)**
  objects: list of ArkObjRef. The function performs deletion of objects given with ArkObjRef reference. Deletion is possible if the reference comes from Master/default filter or it is the last reference in the Secondary filter. If any object could not be deleted - whole deletion operation refused with the error report. It is recommended to gather all objects to be deleted and run this function once to increase performance.

Querying and modifying current project information

- **GetBSInformation()**
  Returns business space information as XML string
- **GetMetaProjectInfo()**
  Returns meta data - information about revisions
- **SetMetaProjectInfo()**
  Set meta data - information about revisions
- **SVNCommit(fullFilePath, SVNRepo, SVNLogin, SVNPassword, comment = '')**
  Uploads the file specified by fullFilePath to the Data server and commits it to the specified SVN Repository with given credentials. Optionally can set SVN comment for the file. Returns resulting SVN revision number. Argument **SVNRepo** must be a correct URL to the SVN repository folder (this implies %-escaping of unsafe characters like spaces).
- **SVNDirList(SVNRepo, SVNLogin, SVNPassword)**
  Returns xml containing list of directories and files in the given SVN Repository. Connects to the SVN server with given credentials. Argument **SVNRepo** must be a correct URL to the SVN repository folder (this implies %-escaping of unsafe characters like spaces).
- **SVNGet(fileName, SVNRepo, SVNLogin, SVNPassword, folder = '', revisionNumber = '')**
  Download chosen file to specified or default folder (if empty folder path provided) from the given SVN Repository path. Connects to the

SVN server with given credentials. Optionally can define desired SVN revision number for the file. Argument **SVNRepo** must be a correct URL to the SVN repository folder (this implies %-escaping of unsafe characters like spaces). **File name must be URL-escaped too** (spa ces replaced with %20). If the file with the specified name already exists in the folder, it will be overwritten.

- **SVNLog(SVNRepo, SVNLogin, SVNPassword)**
  Returns the SVN log for the given SVN Repository (can include folder and file name) as a list of dicts {user, time, description, file, revision}. Connects to the SVN server with given credentials. Argument **SVNRepo** must be a correct URL to the SVN repository folder (this implies %-escaping of unsafe characters like spaces).

Simulink interface functions

- **OpenMatlabInstance()**

Creates an ArkSimulinkLib object (opens Matlab engine). It is recommended first to start Matlab instance using the following code:

```
import os
os.system('matlab -automation - desktop')
```

Other functions

- **SideRunScript(workspace, project, script)**
  Starts new **arKItect** instance with the same login and password as the current instance, logs into specified workspace and project, and executed specified Python script in it. Arguments:
  - **workspace**: name of the workspace, string
  - **project**: name of the project, string
  - **script**: path to the script to be executed, string.
- **arki_log(text)**
  Prints string in the **arKItect** output windows (script output, program trace). Used internally to redefine sys.stdout.

ArkAttribute objects

ArkAttribute objects support the following methods:

- **GetName()**
  Return the name of the attribute.

- **GetType()**
  Return the attribute type as it is used in the arKitect ('bool', 'int', 'str', 'pyscript', ...)

- **GetValue()**
  Return the value of the attribute. The type of the returned object depends on the attribute type (Integer, Boolean, ...).

- **SetValue(value)**
  Set the **value** of the attribute. The type of **value** must match the same rules used by **GetValue()**.

Saved File
This method will raise a ValueError exception for Saved File attribute. For Saved File attributes use special methods given below

- **Execute()**
  For the attributes of type 'Program': executes the code in the attribute. Method returns the value, returned by the code. For other attribute types, method raises a ValueError exceptio
- **IsModifiable()**
  Returns True if attribute value can be modified, False otherwise.
- **GetArkObj()**
  Returns ArkObj owner object.

Methods for "SavedFiles" attributes:

- **DownloadFileSVN(fileName, folder=default files folder, revisionNumber=None)**
  For the attributes of type 'Saved files': Download chosen file to specified or default folder.

(revisionNumber parameter type : integer) if "revisionNumber" is not specified the latest is returned. Returns the path to the downloaded file

- **UploadFileSVN(path, fileComment=None, revisionComment=None)**
  For the attributes of type 'Saved files': Upload chosen file on server side and add this file to attribute value. Optionally can set comment for the file.
  If a standard file with the same name already exists, the file you are uploading will be added as a new version of that file. Third argument allows specifying revision message. Upload failed if a shared file with same name already exist. Returns the resulting revision number from SVN server.

- **GetFileRevisionListSVN(fileName)**
  Returns a list of dictionaries with the following keys ['user', 'time', 'description', 'file', 'revision'].
  (Values are of type "String" except the last one: "revison" of type "Integer").

- **RemoveFileSVN(fileName)**
  Remove the saved file from SVN using its "fileName". Deleted file also removed from the attribute attached file list. Can not be used with shared files, use **DetachSharedFile** for them.

- **AttachSharedFile(fileName)**
  Attaches named shared saved file to the object. Shared file can't be attached if a standard file already exists in attribute with the same name. Doesn't check existence

- **DetachSharedFile(fileName)**
  Detaches named shared saved file from the object. Doesn't check existence

Examples:

```
import pyark
import os
def run(self):

    filePathList = r"C:\Users\SomeUserName\Desktop\Nouveau dossier"
    fileToUpdate = r"C:\Users\SomeUserName\Desktop\text.txt"

    # Upload files from "filePathList" to SVN
    for elt in os.listdir(filePathList):
        self.GetAttribute("SF").UploadFileSVN(os.path.join(filePathList, elt))

    # Upload a new version of the file "text.txt"
    self.GetAttribute("SF").UploadFileSVN(fileToUpdate)

 # Share file "text.txt"
    self.GetAttribute("SF").AttachSharedFile(os.path.basename(fileToUpdate))

 # Download all files from SVN
    for elt in self.GetAttribute("SF").GetValue():
        self.GetAttribute("SF").DownloadFileSVN(elt[0],
r"C:\Users\SomeUserName\SomeFolder")

    # Download specific version of the file "text.txt"
    self.GetAttribute("SF").DownloadFileSVN("text.txt",
r"C:\Users\SomeUserName\SomeFolder", "6079")

    # Displays revision list of all file from the repository
    for elt in os.listdir(filePathList):
        for item in self.GetAttribute("SF").GetFileRevisionListSVN(elt):
            print item
```

ArkAttribute - Advanced methods

- **GetFileListSVN()**
  Returns xml containing list of directories and files in the SVN Repository assigned to the attribute. The resulting list contains SVN revision numbers and commit dates

ArkAttributeType objects

ArkAttributeType is a python object representing an **arKItect** Attribute Type.

Standard interface methods

- **GetName()**
  Returns the name of the attribute type.

- **SetName(name)**
  Works only with* ***arKItect* *Designer**
  Changes the name of the attribute type. **name** cannot contain the following symbols : '\ / : * ? " < > |' - in this case the method raises a **ValueError** exception.

- **GetType()**
  Returns type of attribute value (see constants ARK_ATTRIB_xxxx)
- **SetType(attribtype)**
  Works only with* ***arKItect* *Designer**
  Valid only for ARK_ATTRIB_MEMO and ARK_ATTRIB_TEXT
  Changes type of attribute value to **attribtype**: ARK_ATTRIB_MEMO can be promoted to ARK_ATTRIB_MARKUP and ARK_ATTRIB_TEXT can be promoted to ARK_ATTRIB_TEXTLARGE.

  Method is only available in **arKItect 4.4.2** and following

- **GetDefaultValue()**
  Returns the default value of the attribute type.

- **SetDefaultValue(defaultvalue, isModifiable=True)**
  Works only with* ***arKItect* *Designer**
  Sets the default value of the attribute type and defines whether it can be modified later for objects. If **isModifiable** is False - the default value will be displayed for all objects (even if individual attribute value was defined for the object before) (excluding Date and Author attributes)

- **ResetDefaultValue()**
  Works only with* ***arKItect* *Designer**
  Removes default value.

- **IsModifiable()**
  Returns True if attribute value can be modified, False otherwise. If returned value is False, then any attribute of this type will retain its default value forever. Use **SetDefaultValue()** method to modify this flag.
- **SetModifiable(isModifiable)**
  Works only with* ***arKItect* *Designer**
  Defines whether default value of the attribute type can be modified. If default value is not yet defined, the method raises an error - **SetDefaultValue** method should be used instead.
  To apply changes, **FlushProperties**() method must be called after.
  Note that the modification does not changes behavior of ArkAttribute objects, requested before the call to **SetModifiable**.

- **GetAttribute(name)**
  Returns the ArkAttributeType object identified by **name** or **None** if it does not exist.

- **GetAttributeList()**
  Returns the list of child ArkAttributeType objects. The list will be non empty only for the attribute types ARK_ATTRIB_ENUM and ARK_ATTRIB_GROUP.
- **GetParentsList()**
  Returns list of all ArkMatrixType and ArkAttributeType objects, that represent rules and attribute groups, containing given attribute.

- **AddAttribute(arkattrib, attribtype=None, defaultvalue=None)**
  Works only with* ***arKItect* *Designer**

Valid only for ARK_ATTRIB_GROUP and ARK_ATTRIB_ENUM. In the case of ARK_ATTRIB_ENUM only attributes of type ARK_ATTRIB_ENUM_VALUE can be added. For further description refer to the *ArkMatrixType* method.

- **DeleteAttribute(arkattrib)**
  Works only with **arKItect Designer**
  Removes the given **arkattrib** from the parent object. **arkattrib** can be a name or an ArkAttributeType object.

- **SetKeys(key_list)**
  Works only with **arKItect Designer**
  Sets object's keys.

- **GetKeys()**
  Returns list of keys (strings)
- **GetIconHandle()**
  Returns the objt`s Icon handle

Properties management methods

- **SetProperty(PropEntry, PropValue)**
  Works only with **arKItect Designer**
  See the same method in the ArkMatrixType object.
  Additional *PropEntry* specific to **ArkAttributeType**:
  - ARK_ATYPE_READ_ONLY: a flag specifying whether attribute can be modified through GUI (in views and properties) - not available for attributes of type ARK_ATTRIB_GROUP and ARK_ATTRIB_ENUM_VALUE
  - ARK_ATYPE_UNIQUE: a flag specifying whether attribute value should be unique in the project - available only for attributes of type ARK_ATTRIB_TEXT
  - ARK_ATYPE_EXCLUDE_FROM_REVMNG: a flag specifying whether changes of object attribute values of this Attribute Type are excluded from object revision management
  - ARK_ATYPE_DISPLAY_ATTRIB_NAME: a flag specifying whether attribute name is displayed in views
  - ARK_ATYPE_DISPLAY_ATTRIB_ICON: a flag specifying whether attribute icon is displayed in views
  - ARK_ATYPE_DISPLAY_ATTRIB_VALUE: a flag specifying whether attribute value is displayed in views

- **GetProperty(PropEntry)**
  See the same method in the ArkMatrixType object.

- **FlashProperties()**
  Works only with **arKItect Designer**
  See the same method in the ArkMatrixType object.

- **RestoreProperties()**
  Works only with **arKItect Designer**
  See the same method in the ArkMatrixType object.

- **GetObjBGImage()**
  Returns an *ArkImage* holding the *ArkObjRef* background image, or **None** if the background image does not exist.

- **GetObjFGImage()**
  Returns an *ArkImage* holding the *ArkObjRef* foreground image, or **None** if the foreground image does not exist.

- **SetObjBGImage(image)**
  Works only with **arKItect Designer**
  Sets the object background image. The parameter **image** should be an ArkImage object.

- **SetObjFGImage(image)**
  Works only with **arKItect Designer**
  Sets the object foreground image. The parameter **image** should be an ArkImage object.

ArkChoice objects

Objects of type ArkChoice are used for choice management. Each such object represents choice or choice category. These objects are returned by the global function **GetChoiceByName()**. To get a root choice, use global function **GetRootChoice()** declared in the **pyark** module.

Methods

- **GetName()**
  Returns the name of the choice or category. For the root choice, name is None.

- **GetId()**
  Returns the identifier of the choice or category as string.

- **GetParent()**
  Returns parent choice as ArkChoice object.

- **GetChildList()**
  Returns list of child ArkChoice objects.

- **GetChoice(name)**
  Returns child ArkChoice object with the given name. If choice was not found returns **None**.

- **IsFolder()**
  Returns **True** if choice is a folder, **False** otherwise.

- **str{_}_()**
  String conversion method (usually should not be called directly). Returns string representation of a choice, effectively the same as **GetNa me()**.

- **SetName(name)**
  Sets choice name. If **name** is incorrect (it cannot contain the following symbols : '\ / : * ? \ " < > |'), raises **ValueError** exception. Requires write access to the matrix.

- **NewChoice(name, is_folder = false)**
  Creates a new ArkChoice object, adds it under current choice and returns it. Argument **name** must be a string, **is_folder** must be boolean. If **name** is incorrect, raises ValueError exception. Requires write access to the matrix.

- **MoveChoice(ark_choice)**
  Moves **ark_choice** under the current choice object. Argument **ark_choice** must be a validArkChoice object instance. Requires write access to the matrix.

- **DeleteChoice(name)**
  Removes child choice with the given name. Requires write access to the matrix.

ArkImage objects

ArkImage objects provide Python interface to manipulating images, used as backgrounds for objects. ArkImage objects support following methods:

- **GetName()**
  For local images, returns path to image file. For network images, returns URL of image.

- **GetTempFilePath()**
  Returns path to temporary file, image stored in. Path is guaranteed to be valid, only while object exists. For local images, no temporary file is created, and **GetTempFilePath()** function returns path to the original image.

- **Save(path)**
  Saves image to the specified local path. Image format is determined by file extension.

- **str{_}_()**
  String conversion. Returns string representation of image object in form [Image:url]

ArkMatrixType objects

ArkMatrixType is a python object representing an **arKItect** Matrix Type.

Standard interface methods

- **GetName()**
  Returns the name of the type.

- **SetName(name)**
  Works only with **arKItect Designer**
  Changes the name of the type. **name** cannot contain the following symbols : '\ / : * ? \ " < > |' - in this case the method raises a **ValueError** exception.

- **GetChild(name)**
  Returns the ArkMatrixType object identified by **name** or **None** if it does not exist.

- **GetAttribute(name)**
  Returns the ArkAttributeType object identified by **name** or **None** if it does not exist.

- **GetChildList()**
  Returns the list of child ArkMatrixType objects.

- **GetAttributeList()**
  Returns the list of child ArkAttributeType objects.

- **GetParentsList()**
  Returns a list of parent ArkMatrixType objects.

- **SetKeys(key_list)**
  Works only with **arKItect Designer**
  Sets object's keys.

- **GetKeys()**
  Returns list of keys (strings)
- **GetIconHandle()**
  Returns the objt`s Icon handle

Rule management methods

- **AddRule(arktype, RuleType=pyark.ARK_RULE_SIMPLE, bHideOnExpand=False)**
  Works only with **arKItect Designer**
  Adds the given arKItect type to the parent type as a child.
    - **RuleType** defines the type of link: pyark.ARK_RULE_FLOW, pyark.ARK_RULE_INPUT, pyark.ARK_RULE_OUTPUT, pyark.ARK_RULE_BIFLOW or pyark.ARK_RULE_SIMPLE.
    - **arktype** can be a name or an *ArkMatrixType* object. If it is a name which is supplied, and if the corresponding ArkMatrixType object does not exist it will be created.
    - Parameter **bHideOnExpand** is actual only if **RuleType** is ARK_RULE_FLOW, ARK_RULE_INPUT, ARK_RULE_OUTPUT, ARK_RULE_BIFLOW and defines whether flow objects of **arktype** will be hide or displayed in expanded graphs of the parent object.

  Returns the child ArkMatrixType in case of success and **None** otherwise.

- **DeleteRule(arktype)**
  Works only with **arKItect Designer**
  Removes the given **arktype** from the parent object. **arktype** can be a name or an ArkMatrixType object.

- **SetRuleType(arktype, ruletype)**
  Works only with **arKItect Designer**
  Sets the Rule type for the given pair of matrix objects. **arktype** can be a name or an ArkMatrixType object. Valid rule type values are:
    - ARK_RULE_SIMPLE: **arktype** is simply included.
    - ARK_RULE_FLOW: **arktype** is a flow allowing input and output.
    - ARK_RULE_INPUT: **arktype** is a flow allowing only input.
    - ARK_RULE_OUTPUT: **arktype** is a flow allowing only output.
    - ARK_RULE_BIFLOW: bit flag that marks **arktype** as a bidirectional flow. It must be added to other flow-type links, giving 3 values:
        - ARK_RULE_BIFLOW : general bidirectional flow
        - ARK_RULE_INPUT | ARK_RULE_BIFLOW : input-only bidirectional flow
        - ARK_RULE_OUTPUT | ARK_RULE_BIFLOW : output-only bidirectional flow

- **GetRuleType(arktype)**
  Return the Rule type for the given pair of matrix objects (see **SetRuleType**). **arktype** can be a name or an ArkMatrixType object.
    - returned value is one of constants: ARK_RULE_SIMPLE, ARK_RULE_FLOW, ARK_RULE_INPUT, ARK_RULE_OUTPUT, ARK_RULE_BIFLOW, ARK_RULE_INPUT | ARK_RULE_BIFLOW, ARK_RULE_OUTPUT | ARK_RULE_BIFLOW

- **SetHideOnExpand(arktype, bHideOnExpand)**
  **Works only with arKItect Designer*
  Defines whether flow objects of **arktype** will be hidden or displayed in expanded graphs of the parent object. Setting this value is actual only for flow child types

- **IsHideOnExpand(arktype)**
  Returns the HideOnExpand flag for the pair of matrix objects. This value is actual only for flow child types

Attribute management methods

- **AddAttribute(arkattrib, attribtype=None, defaultvalue=None)**
  **Works only with arKItect Designer*
  Add the given attribute to the parent type. In case of success the function returns an ArkAttributeType object, if **arkattrib** already exists but has a different **attribtype** or in case of error it returns **None**. The method can be used in two different ways:
  - If **arkattrib** is ArkAttributeType object then the attribute with all its properties is simply copied. In that case, **attribtype** and default value are ignored.
  - If **arkattrib** is a name, then a new attribute of type **attribtype** is created. Note that a default Value can optionally be supplied. The valid **attribtype** for this method are: ARK_ATTRIB_BOOL, ARK_ATTRIB_DATE, ARK_ATTRIB_DOUBLE, ARK_ATTRIB_FILE, ARK_ATTRIB_HYPERLINK, ARK_ATTRIB_INT, ARK_ATTRIB_MEMO, ARK_ATTRIB_POINTERS, ARK_ATTRIB_PROGRAM, ARK_ATTRIB_SAVEDFILE, ARK_ATTRIB_SINGLE, ARK_ATTRIB_TEXT, ARK_ATTRIB_TEXTLARGE, ARK_ATTRIB_ENUM, ARK_ATTRIB_GROUP, ARK_ATTRIB_APP, ARK_ATTRIB_MARKUP.

- **DeleteAttribute(arkattrib)**
  **Works only with arKItect Designer*
  Remove the given **arkattrib** from the parent object. **arkattrib** can be a name or anArkAttributeType object.

Properties management methods

- **SetProperty(PropEntry, PropValue)**
  **Works only with arKItect Designer*
  Assign property a new value.

  SetProperty modifies the property in the script's memory but does not propagate the new value to **arKItect**. You need to call **F lushProperties()** if you want your changes to be effective.

  Valid PropEntryvalues:
  - ARK_ATYPE_CLRTEXT: color of the Text,
  - ARK_ATYPE_CLRLINE: color of the Border,
  - ARK_ATYPE_CLRFILL: color of the Filling,
  - ARK_ATYPE_CLRFILLBG: background color of the Filling (used with patterns),
  - ARK_ATYPE_FILLHATCH: code of pattern, used for filling block background,
  - ARK_ATYPE_CLRFLOW: color of the Flow,
  - ARK_ATYPE_CLRFLOW_TEXT: color of the Flow Text,
  - ARK_ATYPE_CLRFLOW_TRANSP: a flag specifying whether Flow color is transparent,
  - ARK_ATYPE_CLRTEXT_TRANSP: a flag specifying whether Text color is transparent,
  - ARK_ATYPE_CLRTEXTFLOW_TRANSP: a flag specifying whether Flow Text color is transparent,
  - ARK_ATYPE_CLRLINE_TRANSP: a flag specifying whether Border color is transparent,
  - ARK_ATYPE_CLRFILL_TRANSP: a flag specifying whether Filling color is transparent,
  - ARK_ATYPE_CLRFILLBG_TRANSP: a flag specifying whether background filling color is transparent,
  - ARK_ATYPE_LINEWIDTH: width of link,
  - ARK_ATYPE_BORDERSTYLE: style of the border line of block (possible values are ARK_LINESTYLE_SOLID, ARK_LINESTYLE_DASH, ARK_LINESTYLE_DOT, ARK_LINESTYLE_DASHDOT, ARK_LINESTYLE_DASHDOTDOT),
  - ARK_ATYPE_BORDERWIDTH: width of border line of block,
  - ARK_ATYPE_LINESHAPE: corresponding Line Shape (possible values are ARK_LINESHAPE_CURVE, ARK_LINESHAPE_STRAIGHT, ARK_LINESHAPE_ORTHOGONAL),
  - ARK_ATYPE_LINESTYLE: corresponding Line Style (possible values are ARK_LINESTYLE_SOLID, ARK_LINESTYLE_DASH, ARK_LINESTYLE_DOT, ARK_LINESTYLE_DASHDOT, ARK_LINESTYLE_DASHDOTDOT),
  - ARK_ATYPE_ARROWSTYLE: corresponding Arrow Style (possible values are ARK_ARROWSTYLE_ARROW, ARK_ARROWSTYLE_OPEN, ARK_ARROWSTYLE_STEALTH, ARK_ARROWSTYLE_DIAMOND, ARK_ARROWSTYLE_DIAMOND_WHITE, ARK_ARROWSTYLE_CIRCLE),

    - ARK_ATYPE_OBJSHAPE: corresponding Object Shape (possible PropValue values are: ARK_OBJSHAPE_AND, ARK_OBJSHAPE_COMMENT, ARK_OBJSHAPE_CROSS, ARK_OBJSHAPE_CUBE, ARK_OBJSHAPE_DIAMOND, ARK_OBJSHAPE_DOC, ARK_OBJSHAPE_ELLIPSE, ARK_OBJSHAPE_FLAG, ARK_OBJSHAPE_METEOR, ARK_OBJSHAPE_OR, ARK_OBJSHAPE_PLAQUE, ARK_OBJSHAPE_PRLGRM, ARK_OBJSHAPE_RECT, ARK_OBJSHAPE_RECT_RE, ARK_OBJSHAPE_SAIL, ARK_OBJSHAPE_TEARDROP, ARK_OBJSHAPE_TRAPEZ_UP, ARK_OBJSHAPE_TRAPEZ_DOWN, ARK_OBJSHAPE_TRIANGLE_UP, ARK_OBJSHAPE_TRIANGLE_DOWN, ARK_OBJSHAPE_WAVE, ARK_OBJSHAPE_PENTAGON, ARK_OBJSHAPE_HEXAGON, ARK_OBJSHAPE_HEPTAGON, ARK_OBJSHAPE_OCTAGON, ARK_OBJSHAPE_DECAGON, ARK_OBJSHAPE_DODECAGON, ARK_OBJSHAPE_CAN, ARK_OBJSHAPE_CAN_LYING, ARK_OBJSHAPE_SIGN_LEFT, ARK_OBJSHAPE_SIGN_RIGHT, ARK_OBJSHAPE_END_NODE, ARK_OBJSHAPE_INIT_NODE, ARK_OBJSHAPE_CROSS_NODE, ARK_OBJSHAPE_JOIN, ARK_OBJSHAPE_BIFURCATION ARK_OBJSHAPE_SIGN_LEFT, ARK_OBJSHAPE_SIGN_RIGHT, ARK_OBJSHAPE_END_NODE, ARK_OBJSHAPE_INIT_NODE, ARK_OBJSHAPE_CROSS_NODE, ARK_OBJSHAPE_JOIN, ARK_OBJSHAPE_BIFURCATION),
    - ARK_ATYPE_ICON: Icon of the type (as string),
    - ARK_ATYPE_KEEP_RATIO: a flag specifying whether ratio should be kept for graphical objects while scaling,
    - ARK_ATYPE_DEFAULT_SIZE: a tuple of two integers defining default size of graphical objects of the Matrix type,
    - ARK_ATYPE_FLEXIBLE: a flag specifying whether graphical properties of objects implementing the Matrix type are modifiable,
    - ATYPE_GEN_TAG_ATRRIB_BGIMAGE: background image of graphical object,
    - ATYPE_GEN_TAG_ATRRIB_FGIMAGE: foreground image of graphical object,
    - ATYPE_GEN_TAG_ATRRIB_FONT: font of graphical object,
    - ATYPE_GEN_TAG_ATRRIB_FONT_FLOW: font of flow label.
    - ArkMatrixType - Advanced properties

- **GetProperty(PropEntry)**
  Return property value for the given entry. See **SetProperty()** for valid **PropEntry** values.

- **FlushProperties()**
  Works only with **arKItect Designer**

  Be careful, use this method only when you have finished setting properties and want to save modifications. After use of this method it will be impossible to restore previous values

  Write accumulated property changes into memory and database.

- **RestoreProperties()**
  Works only with **arKItect Designer**
  Restore all properties from the memory.

- **GetObjBGImage()**
  Return an *ArkImage* holding the *ArkObjRef* background image, or **None** if the background image does not exist.

- **GetObjFGImage()**
  Return an *ArkImage* holding theArkObjRef foreground image, or **None** if the foreground image does not exist.

- **SetObjBGImage(image)**
  Works only with **arKItect Designer**
  Set the object background **image**. The parameter image should be an ArkImage object.

- **SetObjFGImage(image)**
  Works only with **arKItect Designer**
  Set the object foreground **image**. The parameter image should be an ArkImage object.

Abstraction management methods

Remember to call **FlushProperties**() so that modifications come into force

- **GetAbstraction()**
  Returns list of names of types, that were unchecked in the "Abstraction" properties of the rule.

- **ClearAbstraction()**
  Works only with* ***arKItect* *Designer**
  Reset any abstraction set for the object.

- **SetAbstraction(arktype, status)**
  Works only with* ***arKItect* *Designer**
  Sets the abstraction for given type on the parent object. **status** can be **True** or **False**. **arktype** can be a name or an ArkMatrixType object .

Events management methods

Remember to call **FlushProperties**() so that modifications come into force

- **SetAttributeForEvent(eventType, arkattrib)**
  Works only with* ***arKItect* *Designer**
  Defines a linked attribute for the given eventType.
  - **eventType** defines the type of event: pyark.EVENT_ONADDCHILD, pyark.EVENT_ONADDEXISTING, pyark.EVENT_ONADDNEW, pyark.EVENT_ONCHANGEATTRIBVALUE, pyark.EVENT_ONCHANGEDIRECTION, pyark.EVENT_ONCLOSE, pyark.EVENT_ONDELETE, pyark.EVENT_ONOPEN, pyark.EVENT_ONREMOVECHILD, pyark.EVENT_ONRENAME, pyark.EVENT_ONACTIVATEIBD, pyark.EVENT_ONCHANGEPROPERTIES, pyark.EVENT_ONSERVERINFO or pyark.EVENT_ONCHANGEPROJECTION
  - **arkattrib** can be a name or an ArkAttributeType object. In any case it should result in ArkAttributeType object of pyark.ARK_ATTRIB_PROGRAM type. This ArkAttributeType object should be a child for the current ArkMatrixType object.

- **GetAttributeForEvent(eventType)**
  Returns a linked attribute (ArkAttributeType object) for the event specified.
  - **eventType** defines the type of event: pyark.EVENT_ONADDCHILD, pyark.EVENT_ONADDEXISTING, pyark.EVENT_ONADDNEW, pyark.EVENT_ONCHANGEATTRIBVALUE, pyark.EVENT_ONCHANGEDIRECTION, pyark.EVENT_ONCLOSE, pyark.EVENT_ONDELETE, pyark.EVENT_ONOPEN, pyark.EVENT_ONREMOVECHILD, pyark.EVENT_ONRENAME, pyark.EVENT_ONACTIVATEIBD, pyark.EVENT_ONCHANGEPROPERTIES, pyark.EVENT_ONSERVERINFO or pyark.EVENT_ONCHANGEPROJECTION.

ArkMatrixType - Advanced properties

- **SetProperty(PropEntry, PropValue)**

ARK_ATYPE_SIMTYPE: corresponding Simulink Type (possible PropValue values are: ARK_SIMTYPE_ENABLE, ARK_SIMTYPE_FCALL, ARK_SIMTYPE_SM, ARK_SIMTYPE_SUBSYSTEM, ARK_SIMTYPE_STATE, ARK_SIMTYPE_SIGNAL, ARK_SIMTYPE_TRANSITION, ARK_SIMTYPE_TRIGGER, ARK_SIMTYPE_VECTOR),

ArkObj objects

ArkObj object represents a specific reference of an arkitect object as an instance. This may be useful when we need to know whether an object of the given type and name exists at all without specifying a treeview.
ArkObj objects support the following methods:

- **GetName()**
  Returns the name of the object as string.

- **GetArkType()**
  Returns the type of the object as string.
- **GetID()**
  Returns string with internal ID of the object. IDs of different objects are persistent and unique inside a project. Same value as **ArkObjRef. GetID()**.

- ArkObj objects - Advanced methods

ArkObj objects - Advanced methods

- **GetChildList()**
  Returns list of all direct ArkObj children: flows and regular objects.
- **GetChildRelationList([arktypefilter])**
  Same as **GetChildList()** but returns list of children pairs (*ArkObj*, *relType*) matching the **arktypefilter**. *relType* is one of ARK_RULE_INPUT, ARK_RULE_OUTPUT or ARK_RULE_SIMPLE

- **GetParentList()**
  Returns list of all direct ArkObj parents of this object, including flow producers and consumers.

- **GetAttribute(name)**
  Returns the ArkAttribute object identified by name. If there is no such attribute, the method raises **ValueError**. The method never returns **None**.

- **GetAttributeList()**
  Return the ArkAttribute object list, representing all attributes of the object.

Revision management

Methods are available for any ArkObj object except for the root object.

- **GetRevisionList()**
  Return list of object`s revisions description as a list of dictionaries ('hpoint', 'time', 'user', 'comment', 'name', 'svnrevision').
  Depends on pyark VariantFilter if variant revisions exist for object

- **GetLastRevision()**
  Return last object revision description as a dictionary ('hpoint', 'time', 'user', 'comment', 'name', 'modified', 'svnrevision').
  Depends on pyark VariantFilter if varian revisions exist for object

- **GetCurrentRevision()**
  Return current revision working status as a string:
  "last revision name*" - if there are changes after the last revision
  "last revision name" - if there are no changes after the last revisionDepends on pyark VariantFilter if varian revisions exist for object

- **GetRevision(historyPoint)**
  Return object revision as a read-only ArkObj object . An exception will raise if you try to set an attribute from this object.

- **CreateRevision(name, comment)**
  Create a new revision of object, where **name** is revision number, **comment** is revision description.
  Depends on pyark VariantFilter if varian revisions exist for object

- **RestoreRevision(historyPoint)**
  Restore object to revision specified by **historyPoint**.

- **HaveVariantRevisions()**
  Returns True if any variant revisions exists for the object

Variant management methods

Python interface for "Variants" feature. See also description of the ArkChoice objects, ArkVariant objects, ArkPhase objects and global functions **GetRootChoice()**, **GetRootVariant()**, **GetRootPhase()**, **GetActiveVariant()**, **GetActivePhase()**, **GetDefaultVariant()**, **GetDefaultPhase()**.

- **GetChoice(choice, index = 0)**
  Returns a boolean value, describing whether given **choice** is set for the object or not.
  The argument **choice** must be an ArkChoice object. Root choice category can not be used as argument (if root choice is passed as argument, exception is raised). **index** (>= 0) may be specified if the composite choices are used.

- **GetVariant(variant)**
  Returns a boolean value, describing whether given **variant** is set for the object or not.
  The argument **variant** must be an ArkVariant object. Root variant category can not be used as argument (if root variant is passed as argument, exception is raised). Unlike choices, no composite variants are allowed.

- **GetPhase(phase)**
  Returns a boolean value, describing whether given **phase** is set for the object or not.

The argument **phase** must be an ArkPhase object. Root phase category can not be used as argument (if root phase is passed as argument, exception is raised). Unlike choices, no composite phases are allowed.

- **SetChoice(choice, value, index = 0)**
  Set or reset choice flag for the object.
  choice must be an ArkChoice object. **value** must be a boolean. **index** (>= 0) may be specified if the composite choices are used; if the specified index does not exist, all missing composite choices from 1 to *index* will be created and set to **False**. If **choice** has children then **value** is set for them too.
  Choice object must not be root choice, choice folder or choice category without children.
  In case of incorrect arguments, **ValueError** exception is raised.
  When multiple choices (options) are to be set consider using **SetChoiceList()** method

- **SetVariant(variant, value)**
  Set or reset variant flag for the object.
  Variant must be an ArkVariant object. **value** must be a boolean. If **variant** has children then **value** is set for them too. Variant object must not be root variant nor variant folder.
  In case of incorrect arguments, **ValueError** exception is raised.
  When multiple variants are to be set consider using **SetVariantList()** method

- **SetPhase(phase, value)**
  Set or reset phase flag for the object.
  Phase must be an ArkPhase object. **value** must be a boolean. If **phase** has children then **value** is set for them too.
  Phase object must not be root phase nor phase folder.In case of incorrect arguments, **ValueError** exception is raised.
  When multiple phases are to be set consider using **SetPhaseList()** method

- **GetChoiceList(choices)**
  choices - list (or any iterable object) of ArkChoice objects.
  Returned value is list of the same length as **choices**.
  Each element of the returned list is another list of boolean values, representing "checked" status of a choice in each composite choice column. Number of elements of each inner list is equal to value, returned by **ArkObj.GetNumberOfCompositeChoices()**. In simplest case (number of composite choices is 1), each inner list contains 1 boolean value.

- **GetVariantList(variants)**
  variants - list (or any iterable object) of ArkVariant objects.
  Returned value is list of the same length as **variants**. Each element of the returned list is a boolean value, representing "checked" status of a variant.

- **GetPhaseList(phases)**
  phases - list (or any iterable object) of ArkPhase objects.
  Returned value is list of the same length as **phases**. Each element of the returned list is a boolean value, representing "checked" status of a phase.

- **SetChoiceList(args[, choiceStatus])**
  Allows to set multiple choices at once. Much faster than calling **SetChoice** in cycle.
  args - list (or any iterable container) of choice descriptions.
  Choice description is roughly a tuple of arguments, passed to the **ArkObj.SetChoice**. It can be one of:
  - Single ArkChoice object
    Corresponding choice is set to "checked".
  - Tuple of 2 elements: (ArkChoice, boolean)
    Corresponding choice is set to "checked" or "unchecked" according to the boolean flag
  - Tuple of 3 elements: (ArkChoice, boolean, int)
    Sets value of a composite choice, see **SetChoice** for details. Third argument is number of composite choice column.

Different formats can be used in the same list.
choiceStatus - optional parameter, can be only ARK_OBJVAR_MANUAL.

- **SetVariantList(args[, variantStatus])**
  Allows to set multiple variants at once. Much faster than calling **SetVariant** in cycle.
  args - list (or any iterable container) of choice descriptions.
  Variant description is roughly a tuple of arguments, passed to the **ArkObj.SetVariant**. It can be one of:

  - Single ArkVariant object
    Corresponding variant is set to "checked".
  - Tuple of 2 elements: (ArkVariant, boolean)

Corresponding variant is set to "checked" or "unchecked" according to the boolean flag

Different formats can be used in the same list.
variantStatus - optional parameter, can be one of ARK_OBJVAR_MANUAL, ARK_OBJVAR_PROPAGATED.

- **SetPhaseList(args[, phaseStatus])**
  Allows to set multiple phases at once. Much faster than calling **SetPhase** in cycle.
  args - list (or any iterable container) of phase descriptions.
  Phase description is roughly a tuple of arguments, passed to the **ArkObj.SetPhase**. It can be one of:

  - Single ArkPhase object
    Corresponding variant is set to "checked".
  - Tuple of 2 elements: (ArkPhase, boolean)
    Corresponding variant is set to "checked" or "unchecked" according to the boolean flag

Different formats can be used in the same list.
phaseStatus - optional parameter, can be one of ARK_OBJVAR_MANUAL, ARK_OBJVAR_PROPAGATED_UP.

- **CopyOptionsFrom(object)**
  Copies choices from the specified **ArkObj** instance.

- **CopyVariantsFrom(object)**
  Copies variants from the specified **ArkObj** instance.

- **CopyPhasesFrom(object)**
  Copies phases from the specified **ArkObj** instance.

- **IsInVariant (variant, phase = None, onlyOptions = False)**
  Returns **True** or **False**, depending on whether the object is visible under the **variant** or/and under the **phase**.
  variant must be an ArkVariant object or None.
  phase must be an ArkPhase object or None.
  onlyOptions is used only with **variant** not None and means that object belonging to variant will be decided basing on options and not variants information.

- **GetNumberOfCompositeChoices()**
  Return the number of composite choices (choice columns) of the ArkObj objects. Default non-composite choices are considered being of size 1.

- **RemoveCompositeChoice(index)**
  Remove one of the columns of a composite choice. **index** must be a nonnegative integer. Does nothing if there's no choice at *index*, nor if there's only one remaining choice. Since indexes are strictly continuous, all choices at *i* where *i*>*index* are shifted to *i*-1.

- **ResetChoices(onlyPropagation = False)**
  Removes any choice definition from the current object if **onlyPropagation** is not specified or is True, otherwise removes choice definition from the current object only if its status is ARK_OBJ_PROPAGATED.

- **ResetVariants(onlyPropagation = False)**
  Removes any variant definition from the current object if **onlyPropagation** is not specified or is True, otherwise removes variant definition from the current object only if its status is ARK_OBJ_PROPAGATED.

- **ResetPhases(onlyPropagation = False)**
  Removes any phase definition from the current object if **onlyPropagation** is not specified or True, otherwise removes phase definition from the current object only if its status is ARK_OBJ_PROPAGATED.

- **GetOptionsStatus()**
  Returns options' status: ARK_OBJVAR_UNDEFINED or ARK_OBJVAR_MANUAL or ARK_OBJVAR_PROPAGATED.

- **GetVariantsStatus()**
  Returns variants' status: ARK_OBJVAR_UNDEFINED or ARK_OBJVAR_MANUAL or ARK_OBJVAR_PROPAGATED.

- **GetPhasesStatus()**
  Returns phases' status: ARK_OBJVAR_UNDEFINED or ARK_OBJVAR_MANUAL or ARK_OBJVAR_PROPAGATED or ARK_OBJVAR_PROPAGATED_UP.

- **GetOptions()**
  Returns a pair:

optionsStatus, [[list of checked ArkChoice objects in column 1], [list of checked ArkChoice objects in column 2], ...] - in case **optionsStatus** = ARK_OBJVAR_MANUAL
or
optionsStatus, [list of referenced ArkObj objects] - in case **optionsStatus** = ARK_OBJVAR_PROPAGATED

- **GetVariants()**
  Returns a pair:
  variantsStatus, [list of checked ArkVariant objects, may contain **None** meaning all is checked]
  or
  variantsStatus, None - in case nothing is checked

- **GetPhases()**
  Returns a pair:
  phasesStatus, [list of checked ArkPhase objects, may contain **None** meaning all is checked] - in case **phasesStatus** = ARK_OBJVAR_MANUAL
  or
  phasesStatus, [list of referenced ArkObj objects] - in case **phasesStatus** = ARK_OBJVAR_PROPAGATED_UP
  or
  phasesStatus, None - in case nothing is checked

- **PropagateOptions(refObjects, updateOnly =false**)
  Propagates options of referenced ArkObj objects passed in **refObjects** list.
  If **updateOnly** is not set or set to **False** it will rewrite existing options propagation, otherwise it will append to existing propagation set.

- **PropagateVariants**(**refVariants, resetPropagation =false**)
  Propagates variants using referenced ArkVariant objects passed in **refVariants** list.
  If **resetPropagation** is set to **True** it will simply remove existing variants propagation ignoring other parameters.

- **PropagatePhases**(**refObjects, propagationType = ARK_OBJVAR_PROPAGATED, resetPropagation =False**)
  Propagates phases of referenced ArkObj objects passed in **refObjects** list in case **propagationType** = ARK_OBJVAR_PRPAGATED.
  Propagates phases using referenced ArkPhase objects passed in **refObjects** list in case **propagationType** = ARK_OBJVAR_PRPAGATED_UP
  If **resetPropagation** is set to **True** it will simply remove existing phases propagation ignoring other parameters.***

ArkObjRef objects

- Standard interface methods
- Objects management methods
- Graphical interface methods
- View graph management methods
- Variant management methods
- Properties management methods
- SendAlert management methods
- Revision management
- Custom view filter support
- History management methods

ArkObjRef object represents a specific reference of an **arKItect** object in a specific treeview.

This means that the ArkObjRef object is attached to a treeview and holds the path of the reference in this treeview. The reference path uniquely identifies the reference in the treeview, e.g. it can consist of the list of the reference's parents in the treeview, starting from the root to the reference itself. The reference full path consist of the treeview and the reference path in the treeview, it uniquely identifies the reference in the system
ArkObjRef objects support the following methods:

Standard interface methods

- **GetName()**
  Returns the arKItect object name string.

- **GetDisplayedName()**
  Returns the arKItect object name string as displayed in the current treeview.

- **SetName(name)**
Renames the object. **name** cannot contain the following symbols : '\ / : * ? \ " < > |' - in this case the method raises a **ValueError** exceptio n.

- **GetArkType()**
Returns the arKItect object type string.

- **GetChildList([arktypefilter])**
Returns a list of children matching the **arktypefilter**. **arktypefilter** consist of a list of allowed types string, if no parameter is passed then no filter is applied. In case the number of allowed types equals one, it may be passed as a string - no need to put it into a list.
- **GetChildRelationList([arktypefilter])**
Same as **GetChildList()** but returns list of children pairs (*ArkObjRef*, *relType*) matching the **arktypefilter**. *relType* is one of ARK_RULE_INPUT, ARK_RULE_OUTPUT or ARK_RULE_SIMPLE.

- **GetChild(name, [type])**
Returns a child object by its name. To distinguish between several objects with the same name, type name may be specified. Both **name** and **type** name must be strings. If there is no such child, returns **None**.

- **GetParent()**
Returns a reference to the parent object. If called for the root object, returns **None**. Note that in the arKItect, each object can have several instances and thus have several parents. Each *ArkObjRef* instance represents exactly one object instance, and thus have exactly one parent. If you need to get the list of all parents, use arkiutils library.

- **GetParentRelation()**
Returns an integer constant, indicating the relation between object and its parent.Can be one of the following:
    - RELATION_NONE- returned, if the object is root (and therefore has no parent).
    - RELATION_DIRECT_CHILD- returned if the object is direct child of its parent.
    - RELATION_INDIRECT_CHILD- the object is an indirect child, i.e. some of its parent are hidden (unchecked) by the tree.

- **GetInputFlowList([arktypefilter])**
Same as **GetChildList()** but returns only input flows.

- **GetOutputFlowList([arktypefilter])**
Same as **GetChildList()** but returns only output flows.

- **GetAttribute(name)**
Returns the ArkAttribute object identified by name.If there is no such attribute, or attribute is not visible (hidden by filter), the method raises **ValueError**. The method never returns **None**.

> **GetAttribute(*enum*)** will raise **ValueError** when the object is filtered out because of the enumerator *enum* in the current treeview. It may happen when the script is run at the object creation.

- **GetAttributeList()**
Return the ArkAttribute object list, representing all attributes of the object.

> **GetAttributeList()** will not return some enumerators if the object is filtered out because of these enumerators in the current treeview. It may happen when the script is run at the object creation.

- **IsChildFlow(arktype, name, [flowdir])**
Return a boolean flag, indicating if the child object given by its **name** and **type** is a flow or not. If the **flowdir** parameter is specified, flow direction is also checked. **flowdir** must be either "input" or "output".

- **GetTree()**
Returns the *ArkTreeViewObj* where the current object belongs to

- **IsChildPossible(ark_type_name)**
Returns True, if both matrix definition and tree definition allow creating child of the specified type. If it is not possible, returns False. If object if not accessible or frozen, raises an exception.

- **GetID()**
Returns string, containing unique object identifier. Identifier is guaranteed to be the same for all instances of one object in every view, and guaranteed to be different for any different objects.

- **__str__()**
  String conversion function, usually should not be called directly. Returns string representation of the object in the form **Object_Name(Object_Type)**.This function is called automatically, when you pass the ArkObjRef object somewhere, where the string is required.

Objects management methods

- **AddChildObject(arktype, name=None, flowdir='output', variant='empty')**
  Add a child object of type **arktype** and return this child object. If **name** is **None** then a dummy object (i.e. with a automatically generated name) is created. Otherwise if the arKItect object (**name**, **arktype**) already exists then a reference to this object is added to the child list and in the case the object (**name**, **arktype**) does not yet exist it is created and added to the child list. The argument **flowdir** is used only when **arktype** is a flow, it specifies if the flow is an 'input' flow or an 'output' flow. If **arktype** is not a valid type then the function returns **None**. The argument **variant** specifies whether variant settings are to be set for the newly added object - possible values are 'empty' and 'current': 'empty' will apply no variant settings, while **'current'** will force application of currently active variant settings

- **AddChildReference(arkobjref, flowdir='output')**
  Add a reference to the arKItect object **arkobjref** to the child list (no effect if **arkobjref** is already in the child list) and return corresponding ArkObjRef object.

- **RemoveChild(arktype, name, [flowdir])**
  Remove the object of **arktype** and **name** from the child list. If no other reference to the object exists then it is destroyed.

- **SetChildFlowDirection(child, flowdir)**
  Set the direction of the child flow. direction can be either 'input' or 'output'. **child** is the a child *ArkObjRef*.

- **GetParentsOfType(treeName, [typeName])**
  This function returns list of all parents of object, having specified type. If **typeName** is **None** of not specified, all parents are returned. Search is performed according to tree **treeName** from its root.This function gives "Get location" functionality to Python scripts.

Graphical interface methods

- **GetGraphChildPos(child, view_id)**
  Return the child coordinates in the view specified: (x, y, width, height). **child** can either be the child name string or the corresponding ArkObjRef object. Valid *view_id* are: ARK_GRAPH_INNER_VIEW, ARK_GRAPH_PEER_VIEW.

- **SetGraphChildPos(child, view_id, x, y, width=0, height=0)**
  Change the child coordinates in the view specified. A value of zero for **width** or **height** means that this parameter should not be modified. **child** can either be the child name string or the corresponding ArkObjRef object. Valid **view_id** are: ARK_GRAPH_INNER_VIEW, ARK_GRAPH_PEER_VIEW. This method works only if the graph exists (if the graph was changed manually at least once). This method will not create the graph is it doesn`t exist. It is better to use "NewGraphParser"

- **GetObjBGImage(onlyObject = *False*)**
  If background image was specified for the object, this method returns the image, regardless to argument. If object has no own background image, and the argument has its default value **False**, method tries to get background image of abstract object, associated with the object.If the argument has value of **True**, then method does not try to refer to abstract object and returns **None**, if object has no own background. Thus, when the argument is **False**(default), the method returns visible image, shown in the view.

- **GetObjFGImage(onlyObject = *False* ***)
  If foreground image was specified for the object, this method returns the image, regardless to argument. If object has no own foreground image, and the argument has its default value ***False**, method tries to get foreground image of abstract object, associated with the object.If the argument has value of **True**, then method does not try to refer to abstract object and returns **None**, if object has no own foreground. Thus, when the argument is **False**(default), the method returns visible image, shown in the view.

- **SetObjBGImage(image)**
  Set the object background image. The parameter should be an ArkImage object.

- **SetObjFGImage(image)**
  Set the object foreground image. The parameter should be an ArkImage object.

- **GetRefInTree(target_tree)**
  Return the corresponding *ArkObjRef* in the treeview **target_tree** or raise an exception if no unique corresponding reference could be found. Refer to the description of the reference mapping mechanism.

- **SaveViewAsImage(file_name, view_id = *ARK_GRAPH_INNER_VIEW*, show_title = *True*)**
  Selects object in a Internal view and saves graph to an image file. Image format is determined by extension. Supported formats are JPG,

BMP, TIFF..Valid **view_id** are: ARK_GRAPH_INNER_VIEW, ARK_GRAPH_PEER_VIEW, ARK_GRAPH_MATDRAW_INNER_VIEW, ARK_GRAPH_MATDRAW_PEER_VIEW.
If one of Matdraw views is specified as a **view_id**. With this **view_id** :
    - The **show_title** parameter is ignored
    - Additional image formats are supported (PNG, ICO, XPM, ...)
    - ValueError is raised with a human readable error string in case of failure.

- **ResetOuterflowPorts(view_id)**
  Recalculates positions and grouping of outerflow ports and redraws the view. Valid **view_id** are: ARK_GRAPH_INNER_VIEW, ARK_GRAPH_PEER_VIEW

View graph management methods

The following two methods are intended for saving and restoring view graph configuration. View graph configuration describes geometrical positions of the objects in the view, their expansion state and the form of the inks between objects. In arKItect, graph is described by XML file. Using the following two methods, user can get or set the XML graph description.

- **GetGraphXML(graph_id, bForceNew = False, bOnlyVisible = False)**
  Returns string, containing XML description of a graph. The **graph_id** argument specifies, which view configuration is returned. Valid **view _id** are: ARK_GRAPH_INNER_VIEW, ARK_GRAPH_PEER_VIEW. Set **bForceNew** to **True** if graph should be regenerated.
  Set **bOnlyVisible** to **True** to get the elements actually present in the diagram.

bOnlyVisible is only available in **arKItect 2.1.2** and following

- **SetGraphXML(graph_id, xml, update_subtrees=False)**
  Sets the XML for graph. Arguments:
    - **xml** XML string, having the same format, as returned by **GetGraphXML**.
    - **graph_id** Type of graph to return. Valid view types are: ARK_GRAPH_INNER_VIEW, ARK_GRAPH_PEER_VIEW.
    - **update_subtrees** (optional) if True, then all corresponding graphs in the sub-trees will be updated, using standard graph synchronization mechanism.

- **SynchronizeViews()**
  Synchronize Peer and Inner Views. Peer View of object become the same, as its parent Inner View. Note that the exact format of the XML string is the same as internal arKItect format, which is subject to change in the future versions. Therefore, it is only guaranteed that XML, returned by **GetGraphXML** will be accepted by the **SetGraphXML** method.

Variant management methods

Python interface for "Variants" feature. See also description of the ArkChoice objects, ArkVariant objects, ArkPhase objects and global functions **GetRootChoice()**, **GetRootVariant()**, **GetRootPhase()**, **GetActiveVariant()**, **GetActivePhase()**, **GetDefaultVariant()**, **GetDefaultPhase()**.

- **GetChoice(choice, index = 0)**
  Returns boolean value, describing whether given **choice** is set for the object or not. The argument **choice** must be an ArkChoice object. Root choice category can not be used as argument (if root choice is passed as argument, exception is raised). **index** (>= 0) may be specified if the composite choices are used.
- **GetVariant(variant)**
  Returns boolean value, describing whether given **variant** is set for the object or not. The argument **variant** must be an ArkVariant object. Root variant category can not be used as argument (if root variant is passed as argument, exception is raised). Unlike choices, no composite variants are allowes.
- **GetPhase(phase)**
  Returns boolean value, describing whether given **phase** is set for the object or not. The argument **phase** must be an ArkPhase object. Root phase category can not be used as argument (if root phase is passed as argument, exception is raised). Unlike choices, no composite phases are allowes.
- **SetChoice(choice, value, index = 0)**
  Set or reset choice flag for the object. **choice** must be an ArkChoice object. **value** must be a boolean. **index** (>= 0) may be specified if the composite choices are used; if the specified index does not exist, all missing composite choices from 1 to *index* will be created and set to **False**. If **choice** has children then **value** is set for them too.
  Choice object must not be root choice, choice folder or choice category without children.
  In case of incorrect arguments, **ValueError** exception is raised.
- **SetVariant(variant, value)**
  Set or reset variant flag for the object. **Variant** must be an ArkVariant object. **value** must be a boolean. If **variant** has children then **value**

is set for them too.
Variant object must not be root variant nor variant folder.
In case of incorrect arguments, **ValueError** exception is raised.

- **SetPhase(phase, value)**
  Set or reset phase flag for the object. **Phase** must be an ArkPhase object. **value** must be a boolean. If **phase** has children then **value** is set for them too.
  Phase object must not be root phase nor phase folder.
  In case of incorrect arguments, **ValueError** exception is raised.
- **GetChoiceList(choices)**
  choices - list (or any iterable object) of ArkChoice objects.
  Returned value is list of the same length as **choices**. Each element of the returned list is another list of boolean values, representing "checked" status of a choice in each composite choice column. Number of elements of each inner list is equal to value, returned by **ArkObjRef.GetNumberOfCompositeChoices()**. In simplest case (number of composite choices is 1), each inner list contains 1 boolean value.
- **GetVariantList(variants)**
  variants - list (or any iterable object) of ArkVariant objects.
  Returned value is list of the same length as **variants**. Each element of the returned list is a boolean value, representing "checked" status of a variant.
- **GetPhaseList(phases)**
  phases - list (or any iterable object) of ArkPhase objects.
  Returned value is list of the same length as **phases**. Each element of the returned list is a boolean value, representing "checked" status of a phase.

- **SetChoiceList(args[, choiceStatus])**
  Allows to set multiple choices at once. Much faster than calling **SetChoice** in cycle.
  args - list (or any iterable container) of choice descriptions.
  Choice description is roughly a tuple of arguments, passed to the **ArkObjRef.SetChoice**. It can be one of:
    - Single ArkChoice object
      Corresponding choice is set to "checked".
    - Tuple of 2 elements: (ArkChoice, boolean)
      Corresponding choice is set to "checked" or "unchecked" according to the boolean flag
    - Tuple of 3 elements: (ArkChoice, boolean, int)
      Sets value of a composite choice, see **SetChoice** for details. Third argument is number of composite choice column.

  Different formats can be used in the same list.
  choiceStatus - optional parameter, can be one of ARK_OBJVAR_UNDEFINED, ARK_OBJVAR_MANUAL, ARK_OBJVAR_PROPAGATED.
- **SetVariantList(args[, variantStatus])**
  Allows to set multiple variants at once. Much faster than calling **SetVariant** in cycle.
  args - list (or any iterable container) of choice descriptions.
  Variant description is roughly a tuple of arguments, passed to the **ArkObjRef.SetVariant**. It can be one of:
    - Single ArkVariant object
      Corresponding variant is set to "checked".
    - Tuple of 2 elements: (ArkVariant, boolean)
      Corresponding variant is set to "checked" or "unchecked" according to the boolean flag

  Different formats can be used in the same list.
  variantStatus - optional parameter, can be one of ARK_OBJVAR_UNDEFINED, ARK_OBJVAR_MANUAL, ARK_OBJVAR_PROPAGATED.
- **SetPhaseList(args[, phaseStatus])**
  Allows to set multiple phases at once. Much faster than calling **SetPhase** in cycle.
  args - list (or any iterable container) of phase descriptions.
  Phase description is roughly a tuple of arguments, passed to the **ArkObjRef.SetPhase**. It can be one of:
    - Single ArkPhase object
      Corresponding variant is set to "checked".
    - Tuple of 2 elements: (ArkPhase, boolean)
      Corresponding variant is set to "checked" or "unchecked" according to the boolean flag

  Different formats can be used in the same list.
  phaseStatus - optional parameter, can be one of ARK_OBJVAR_UNDEFINED, ARK_OBJVAR_MANUAL, ARK_OBJVAR_PROPAGATED, ARK_OBJVAR_PROPAGATED_UP.

- **CopyOptionsFrom(object)**
  Exactly copies choices from the specified **ArkObjRef** instance.

- **CopyVariantsFrom(object)**
  Exactly copies variants from the specified **ArkObjRef** instance.
- **CopyPhasesFrom(object)**
  Exactly copies phases from the specified **ArkObjRef** instance.

- **IsInVariant (variant, [phase])**
  Returns **True** or **False**, depending on whether the object specified is visible under the variant and phase, if phase is specified. The argument **variant** must be an ArkVariant object, **phase**, if specified - must be an instance of ArkPhase object

- **ApplyVariant (variant)**

  Deprecated - does nothing and returns None.

  Apply Variant to the current object. The argument **variant** must be an ArkVariant object or **None**. If **None** is passed variant information is removed from the current object.

- **GetNumberOfCompositeChoices()**
  Return the number of composite choices (choice columns) of the ArkObjRef object. Default non-composite choices are considered being of size 1.

- **RemoveCompositeChoice(index)**
  Remove one of the columns of a composite choice. **index** must be a nonnegative integer. Does nothing if there's no choice at *index*, nor if there's only one remaining choice. Since indexes are strictly continuous, all choices at *i* where *i*>*index* are shifted to *i*-1.
- **ResetChoices(onlyPropagation = False)**
  Removes any choice definition from the current object if **onlyPropagation** is not specified or is True, otherwise removes choice definition from the current object only if its status is ARK_OBJ_PROPAGATED.
- **ResetVariants(onlyPropagation = False)**
  Removes any variant definition from the current object if **onlyPropagation** is not specified or is True, otherwise removes variant definition from the current object only if its status is ARK_OBJ_PROPAGATED.
- **ResetPhases(onlyPropagation = False)**
  Removes any phase definition from the current object if **onlyPropagation** is not specified or True, otherwise removes phase definition from the current object only if its status is ARK_OBJ_PROPAGATED.
- **GetOptionsStatus()**
  Returns options' status: ARK_OBJVAR_UNDEFINED or ARK_OBJVAR_MANUAL or ARK_OBJVAR_PROPAGATED.
- **GetVariantsStatus()**
  Returns variants' status: ARK_OBJVAR_UNDEFINED or ARK_OBJVAR_MANUAL or ARK_OBJVAR_PROPAGATED.
- **GetPhasesStatus()**
  Returns phases' status: ARK_OBJVAR_UNDEFINED or ARK_OBJVAR_MANUAL or ARK_OBJVAR_PROPAGATED or ARK_OBJVAR_PROPAGATED_UP.
- **GetOptions()**
  Returns a pair:
  optionsStatus, [[list of checked ArkChoice objects in column 1], [list of checked ArkChoice objects in column 2], ...] - in case **optionsStatus** = ARK_OBJVAR_MANUAL
  or
  optionsStatus, [list of referenced ArkObjRef objects] - in case **optionsStatus** = ARK_OBJVAR_PROPAGATED
- **GetVariants()**
  Returns a pair:
  variantsStatus, [list of checked ArkVariant objects, may contain **None** meaning all is checked]
  or
  variantsStatus, None - in case nothing is checked
- **GetPhases()**
  Returns a pair:
  phasesStatus, [list of checked ArkPhase objects, may contain **None** meaning all is checked] - in case **phasesStatus** = ARK_OBJVAR_MANUAL
  or
  phasesStatus, [list of referenced ArkObjRef objects] - in case **phasesStatus** = ARK_OBJVAR_PROPAGATED_UP
  or
  phasesStatus, None - in case nothing is checked

- ArkObjRef objects - Advanced methods

Properties management methods

In case abstract type of the object allows properties modification(**ArkMatrixType.GetProperty(pyark.ARK_ATYPE_FLEXIBLE)** returns **True**) it is possible to rewrite default object's properties (inherited from the abstract type) with new values.

- **SetProperty(PropEntry,PropValue)**
  Assign property a new value. For valid **PropEntry** values see ArkMatrixType . **SetProperty()**:**FlashProperties()** should be called to apply changes when all needed properties are set.

- **GetProperty(PropEntry)**
  Return property value for the given entry. See **SetProperty()** for valid **PropEntry** values.

- **FlashProperties()**

  Be careful, use this method only when you have finished setting properties and want to save modifications. After use of this method it will be impossible to restore previous values

  Write accumulated property changes into memory and database.

- **RestoreProperties()**
  Abandon accumulated changes to properties and restore all properties from the memory.

- **ResetProperties()**
  Returns object properties to original state, resetting all modifications.

SendAlert management methods

- **GetURL(view_type = ARK_GRAPH_INNER_VIEW, [user_name])**
  Get arkitect:// URL, pointing to the object.
  Arguments:
  view_type - specifies, which view is pointed by the URL. Can be one of ARK_GRAPH_INNER_VIEW, ARK_GRAPH_PEER_VIEW, ARK_GRAPH_MATDRAW_INNER_VIEW, ARK_GRAPH_MATDRAW_PEER_VIEW.
  user_name (optional) - If specified, embeds user login into URL. See functions pyark.GetUserName(), pyark.GetUsers(). If this parameter is not specified, URL is generated without embedded login.

Revision management

Methods are available for any ArkObjRef object except for the root object.

- **CreateRevision(name, comment, replaceOld = False)**
  Create a new revision of object, where **name** is revision number, **comment** is revision description and **replaceOld** is a flag indicating whether new revision should replace the old one if there is already a revision with the given **name**.
  Depends on pyark VariantFilter if varian revisions exist for object
- **RestoreRevision(historyPoint)**
  Restore object to revision specified by **historyPoint**.

- **GetRevisionList()**
  Return list of object`s revisions description as a list of dictionaries ('hpoint', 'time', 'user', 'comment', 'name', 'svnrevision')
  Depends on pyark VariantFilter if varian revisions exist for object

- **GetLastRevision()**
  Return last object revision description as a dictionary ('hpoint', 'time', 'user', 'comment', 'name', 'modified', 'svnrevision')
  Depends on pyark VariantFilter if varian revisions exist for object
  Depends on pyark VariantFilter if varian revisions exist for object
- **GetCurrentRevision()**
  Return current revision working status as a string:
  "last revision name*" - if there are changes after the last revision
  "last revision name" - if there are no changes after the last revision
  Depends on pyark VariantFilter if varian revisions exist for object

- **GetRevision(historyPoint)**

Return object revision as a read-only ArkObjRef object. An exception will raise if you try to set an attribute from this object.

- **HaveVariantRevisions()**
  Returns True if any variant revisions exists for the object

Custom view filter support

These methods support the "Custom view filter" functionality, allowing Python script to customize visibility of object children in arKitect. In order to define custom filter, object must have **CustoVirewFilter** event handler.

- **ReloadCustomFilter()**
  Clears currently loaded custom view filter data for this object. On next access to object children in a view, where custom view filter is active, it will be loaded again. Always returns **None**.

History management methods

- **GetHistory([dateFrom, dateTo, userNameFilter, hpIdFrom, hpIdTo])**
  This function retrieves history of the object in the project. All arguments are optional and allow to filter retrieved history more precisely:
  - **dateFrom**, **dateTo** - datetime objects that define the part of history being retrieved. In order to use date time objects, you need to import **datetime** module (see example below) in your script.
  - **userNameFilter**- If specified, returns only records, related to the specified user.
  - **hpIdFrom**, **hpIdTo** - if specified, returns only records within the range

  Return value is a list of tuples, each tuple representing separate history record, as visible in *History dialog*. Each tuple consists of following 8 elements: (**hpId**, **userName**, **date**, **treeViewName**, **action**, **actionClass, historyValue, sessionId**), where **date** is a **datetim e** type, and other elements are strings.

  Be careful, there may be a lot of history information in the project, so properly set the filters to make retrieval last not so long

Example calling *GetHistory*:

```
import datetime
def func(self):
    dateFrom = datetime.date(2015,12,1) #from 1 December 2015
    #dateTo = datetime.datetime.now() #until now
 #user = 'user@k.i'
 #hpIdFrom = '450000'
 #hpIdTo = '450000'
 hist = self.GetHistory(dateFrom)  #request history for the given period
 #hist2 = self.GetHistory(None, dateTo, user, hpIdFrom, hpIdTo)  #request history for
the given user within given history points range
 for h in hist:
        print h #printing all found history records
```

- **GetActiveUsers()**
  Returns the list of names of ctive users for the view path given by ArkObjRef.

ArkObjRef object.

ArkObjRef objects - Advanced methods

- **PropagateOptions(refObjects, updateOnly = false**)
  Propagates options of referenced ArkObjRef objects passed in **refObjects** list.
  If **updateOnly** is not set or set to **False** it will rewrite existing options propagation, otherwise it will append to existing propagation set.
- **PropagateVariants**(**refVariants, resetPropagation = false**)
  Propagates variants using referenced ArkVariant objects passed in **refVariants** list.
  If **resetPropagation** is set to **True** it will simply remove existing variants propagation ignoring other parameters.
- **PropagatePhases**(**refObjects, propagationType = ARK_OBJVAR_PROPAGATED, resetPropagation = False**)
  Propagates phases of referenced ArkObjRef objects passed in **refObjects** list in case **propagationType** = ARK_OBJVAR_PRPAGATED.
  Propagates phases using referenced ArkPhase objects passed in **refObjects** list in case **propagationType** =

ARK_OBJVAR_PRPAGATED_UP
If **resetPropagation** is set to **True** it will simply remove existing phases propagation ignoring other parameters.

ArkPhase objects

ArkPhase objects represent *phases*, defined in the project. These objects are returned by the global functions **GetPhaseList()**, **GetPhaseByName()**, **GetActivePhase()**, **GetDefaultPhase()**. To get a root phase object, use global function **GetRootPhase()**.

Methods

- **GetName()**
  Returns name of the phase as string.

- **GetId()**
  Returns unique identifier of the phase as string.

- **IsFolder()**
  Returns **True** if phase is folder, **False** otherwise.

- **IsDefault()**
  Returns **True** if phase is set as default, **False** otherwise.

- **GetParent()**
  Returns parent phase as ArkPhase object.

- **GetChildList()**
  Returns list of child ArkPhase objects.

- **SetName(name)**
  Sets phase name. **name** cannot contain the following symbols : '\ / : * ? " < > |' - in this case the method raises a **ValueError** exception.

- **SetDefault(is_default)**
  Sets phase as default. Argument **is_default** must be boolean.

- **NewPhase(name, is_folder = false)**
  Creates a new ArkPhase object, adds it under current phase and returns it. Argument **name** must be string, **is_folder** must be boolean.

- **MovePhase(ark_phase)**
  Moves **ark_phase** under the current phase object. Argument **ark_phase** must be valid ArkPhase object instance.

ArkProject objects

ArkProject is a python object representing an arKItect Project.
Standard interface methods

- **GetName()**
  Return the name of the project.

- **GetWorkspace()**
  Returns an **ArkWorkspace**, the project belongs to, or **None** for local projects (returned by the method **pyark.GetProjectList**)
- **GetFiltersList()**
  Return the list of existing Filters.

ArkTreeViewObj objects

ArkTreeViewObj is a python object representing an **arKItect** TreeView object.

Standard interface methods

- **GetName()**
  Returns the name of the projection.

- **SetName(name)**
  Works only with* ***arKItect* *Designer**
  Change the name of the projection. **name** cannot contain the following symbols : '\ / : * ? \ " < > |' - in this case the method raises a **Value**

Error exception.

- **GetHiddenDiagrams()**

Method is only available in **arKItect 4.4.2** and following

Returns list of hidden diagrams for projection, possible values are ARK_GRAPH_INNER_VIEW, ARK_GRAPH_PEER_VIEW, ARK_GRAPH_MATHDRAW_VIEW or ARK_GRAPH_TABULAR_VIEW.

- **GetDefaultDiagram()**
  Works only with* ***arKItect* *Designer**
  Returns default diagram for projection. Returned value can be one of the following : ARK_GRAPH_INNER_VIEW or ARK_GRAPH_PEER_VIEW.
- **IsDefault()**
  Returns **True** if the projection is marked as default, **False** otherwise.
- **IsVisible()**
  Returns **True** if the projection is available for display, **False** otherwise
- **IsFolder()**
  Returns **True** if the projection is a folder, **False** otherwise.
- **IsOuterportsDisplayed()**
  Returns **True** if outerflow ports are displayed in the projection, **False** otherwise.
- **IsAttributesDisplayed()**
  Returns **True** if attributes are displayed in the projection, **False** otherwise.
- **GetKeys()**
  Returns list of keys (strings)
- **IsSnapToGrid()**
  Return **True** if objects in all diagrams of the projection always snap to grid, **False** otherwise.
- **IsMetaModelPart()**
  Returns **True** if projection is a metamodel part, **False** otherwise
- **SetDefault(bDefault)**
  Works only with* ***arKItect* *Designer**
  Sets default status of the projection according to **bDefault**. **bDefault** is a boolean.
- **SetHiddenDiagrams(hiddenDiagrams)**
  Works only with* ***arKItect* *Designer**
  Defines diagrams hidden for projection. **hiddenDiagrams** is a list of ARK_GRAPH_INNER_VIEW, ARK_GRAPH_PEER_VIEW, ARK_GRAPH_MATHDRAW_VIEW or ARK_GRAPH_TABULAR_VIEW.

Method is only available in **arKItect 4.4.2** and following

SetDefaultDiagram(defaultDiagram)

- **Works only with* ***arKItect*** *Designer**
  Sets default diagram for projection according to **defaultDiagram**. **defaultDiagram** can be one of the following : ARK_GRAPH_INNER_VIEW or ARK_GRAPH_PEER_VIEW.
- **SetVisible(visibility)**
  Works only with* ***arKItect* *Designer**
  Sets the visibility of a projection
- **SetOuterportsDisplayed(bDisplayed)**
  Works only with* ***arKItect* *Designer**
  Sets outerflow ports display status of the projection according to **bDisplayed**. **bDisplayed** is a boolean.
- **SetAttributesDisplayed(bDisplayed)**
  Works only with* ***arKItect* *Designer**
  Set attributes display status of the projection according to **bDisplayed**. **bDisplayed** is a boolean.

- **SetKeys(key_list)**
  Works only with arKItect Designer
  Sets object's keys.

- **SetSnapToGrid(snapToGrid)**
  Works only with arKItect Designer
  Makes objects in all diagrams of the projection always snap or not to grid according to **snapToGrid**. **snapToGrid** is a boolean. Method is not available for sub-filters.

- **SetMetaModelPart(bMetaModelPart)**
  Works only with arKItect Designer
  Set metamodel status of the projection according to **bMetaModelPart. bMetaModelPart** is a boolean.

Properties management methods

- **SetProperty(arktype, PropEntry, PropValue)**
  Works only with arKItect Designer
  Assign projection property for the given **arktype** a new value. **arktype** is either a name, an *ArkMatrixType* or an ArkAttributeType object.
  Valid **PropEntry** values:
  - ARK_TREEVIEW_DISPNAME: defines the name should be displayed as the name of the passed **arktype** (**PropValue** is either a valid *ArkAttributeType* or **None**, if it is **None** then regular name will be used),
  - ARK_TREEVIEW_MASTERSECONDARY: defines whether filter is Master or Secondary for the given type *ArkMatrixType* (possible **PropValue** values are: ARK_TREEVIEW_MASTER, ARK_TREEVIEW_SECONDARY, empty or **None**)
  - ARK_TREEVIEW_GRAPHORIENT: corresponding orientation for graphs of type *ArkMatrixType* (possible **PropValue** values are: ARK_GRAPHORIENT_STANDARD, ARK_GRAPHORIENT_LR, ARK_GRAPHORIENT_TB)

- ArkTreeViewObj - Advanced properties

- **FlashProperties()**
  Works only with arKItect Designer

  Be careful, use this method only when you have finished setting properties and want to save modifications. After use of this method it will be impossible to restore previous values

  Write accumulated property changes into memory and DB.

- **RestoreProperties()**
  Works only with arKItect Designer
  Restore all properties from the memory.

- **GetProperty(arktype, PropEntry)**
  Return property value for the given entry of the given **arktype** which can be an *ArkMatrixType* or an ArkAttributeType object. See **SetProperty** for valid **PropEntry** and values. This method returns **None**, if specific value is not set for tree.

Abstraction management methods

- **GetAbstraction(arkmatrixtype)**
  For the given **arkmatrixtype** it returns pairs (names of children types, abstraction flag).

- **ClearAbstraction(arkmatrixtype)**
  For the given **arkmatrixtype** reset any abstraction set. **arkmatrixtype** can be a name or an *ArkMatrixType*. If **arkmatrixtype** is Null - abstraction for all types is cleaned. **FlashProperties**() is to be called to apply changes

- **SetAbstraction(arkmatrixtype, arktype, status)**
  Works only with arKItect Designer
  For the given **arkmatrixtype** it sets the abstraction for **arktype**. status can be **True** or **False**. **arkmatrixtype** can be a name or an *ArkMatrixType*. **arktype** can be a name or a ArkMatrixType object. **FlashProperties**() is to be called to apply changes.

Rules management methods

- **IsRuleChecked(matrixObjList)**
  Returns **True** if rule, specified by path is checked in the tree and **False** otherwise.
  matrixObjList is a list of *ArkMatrixType* and ArkAttributeType objects, starting from root.

- **CheckRule(matrixObjList, checkState)**
  Works only with* ***arKItect* *Designer**
  Check/uncheck rule at the given path in the projection. **matrixObjList** is a list of *ArkMatrixType* and ArkAttributeType objects, starting from root.

- **CheckRuleSubRules(matrixObjList, checkState)**
  Works only with* ***arKItect* *Designer**
  Check/uncheck all rules starting from the given path in the projection. **matrixObjListis** a list of *ArkMatrixType* and ArkAttributeType objects. Passing empty list as **matrixObjList** will check/remove all rules depending on **checkState**.

- **CheckRulesSameType(matrixObj, checkState)**
  Works only with* ***arKItect* *Designer**
  Check/uncheck given rule everywhere in the projection. **matrixObj** is an ArkMatrixType object.

- **CheckRuleSubRulesSameType(matrixObj, checkState)**
  Works only with* ***arKItect* *Designer**
  Check/uncheck given rule and its subrules everywhere in the projection. **matrixObj** is an ArkMatrixType object.

- **CheckAttributes(matrixObj, checkState)**
  Works only with* ***arKItect* *Designer**
  Check/uncheck all attributes for the given rule in the projection. **matrixObj** is an ArkMatrixType object.

Projection hierarchy management methods

- **GetParent()**
  Returns a reference to the parent **ArkTreeViewObj** object.

- **GetChildList()**
  Returns list of child ArkTreeViewObj objects

- **NewArkTreeViewObj(name, bCheckRules = True)**
  Works only with* ***arKItect* *Designer**
  Creates a new ArkTreeViewObj object with the given name under the current object. Check rules in it if **bCheckRules** is **True**.

- **NewArkTreeViewFolder(name)**
  Works only with* ***arKItect* *Designer**
  Creates a new **ArkTreeViewObj** object with **Folder** flag set to **True** with the given **name** under the current object.

- **Copy(name)**
  Works only with* ***arKItect* *Designer**
  Creates a copy of the current projection. **name** is the name of the new projection. The new projection is placed under the same parent as the source.

- **Move(ArkTreeViewObj)**
  Works only with* ***arKItect* *Designer**
  Moves current projection under another parent specified as *ArkTreeViewObj*.

- **DeleteArkTreeViewObj(ArkTreeViewObj)**
  Works only with* ***arKItect* *Designer**
  Deletes child **ArkTreeViewObj** from the current projection.

Custom view filter support

These methods support the "Custom view filter" functionality, allowing Python script to customize visibility of object children in arKitect. In order to define custom filter, object must have **CustomViewFilter** event handler.

- **_EnableCustomFilterEnabled(isEnabled::bool)**
  Allows script to temporarily disable (False) and enable back custom filter in a specific tree.
- **_IsCustomFilterEnabled()**
  Returns activity state of the custom filter. Default is True, could be set to False by the above method.

Direct use of these methods is discouraged. Instead, use a context manager **arkiext.ki.chains.generic_chains.disableChains(arktreeviewobj)**, that disables costom filter on enter, and enables it back on exit and supports nesting.

ArkTreeViewObj - Advanced properties

- **SetProperty(arktype, PropEntry, PropValue)**

ARK_TREEVIEW_SIMTYPE: corresponding Simulink Type (possible PropValue values are: ARK_SIMTYPE_ENABLE, ARK_SIMTYPE_FCALL, ARK_SIMTYPE_SM, ARK_SIMTYPE_SUBSYSTEM, ARK_SIMTYPE_STATE, ARK_SIMTYPE_SIGNAL, ARK_SIMTYPE_TRANSITION, ARK_SIMTYPE_TRIGGER, ARK_SIMTYPE_VECTOR),

ArkSimulinkLib objects

ArkSimulinkLib object represents a reference to a matlab instance which allows to import/export models and perform specific actions on models.

ArkSimulinkLib methods:

- **ExecString (str)**
  Executes a given string in matlab engine.

- **OpenModel (path)**
  Opens a model at the given path. This method should be called before any import is done.

- **CloseModel(name)**
  Closes the referenced model.

- **CloseEngine()**
  Closes matlab engine (call it only at the end of work).

- **ImportModel(name, import_properties = "'FollowLinks','on','LookUnderMasks','on'")**
  Imports model hierarchy with appropriate parameters (model is referenced by name). Those parameters will be added to MATLAB Simulink **find_system** function (user can manage import process with those parameters). Returns top *ArkSimulinkObject*

- **CreateModel(name)**
  Creates a new model with a given name

- **CreateRootObject(name)**
  Creates a root hierarchy ArkSimulinkObject object (needed for export) with a given name. When constructing a new hierarchy this object should be at the top.

- **GetRoot()**
  Returns the reference to the top *ArkSimulinkObject* in the hierarchy.

Helper classes for Matlab Simulink model interface

- ArkSimulinkConnection objects
- ArkSimulinkObject objects

ArkSimulinkConnection objects

ArkSimulinkConnection object represents a link between two MATLAB Simulink blocks. On Import, connections are referenced by source/destination objects handle.

ArkSimulinkConnection methods:

- **GetSrcHandle()**
  Returns the source object unique simulink object ID.

- **GetDstHandle()**
  Returns the destination object unique simulink object ID.

- **GetSrcPort()**
  Returns the source port number.

- **GetDstPort()**
  Returns the destination port number.

- **GetLineName()**
  Returns the line caption if exists.

- **ExportLink**
  Exports a link to a Simulink model (source/target object ArkSimulinkObject.fullNames are used as references).

ArkSimulinkObject objects

ArkSimulinkObject object represents a reference to a block in MATLAB Simulink model. This block has some properties (which can be read/written) and may have children. *ArkSimulinkObject* represents an hierarchical structure of MATLAB Simulink model. Each *ArkSimulinkObject* has its name and Simulink type.

***ArkSimulinkObject* methods:**

- **GetName()**
  Returns the name of the object.

- **SetName()**
  Sets the name of the object.

- **GetSimType()**
  Returns object's Simulink type.

- **GetParentObject()**
  Returns the reference to the parent **ArkSimulinkObject** object.

- **GetChildList()**
  Returns the list of children **ArkSimulinkObject** objects.

- **GetInputSignals()**
  Returns the list of input signals (Matlab simulink InPort blocks will be referenced there).

- **GetOutputSignals()**
  Returns the list of output signals (Matlab simulink OutPort blocks will be referenced there).

- **AddChild(Name)**
  Adds a child _ArkSimulinkObject_with a given name (Simulink FullPath will be filled automatically).

- **GetParamValue(paramName)**
  Returns the value of **paramName** parameter in Simulink block as string.

- **SetParamValue(paramName, strValue)**
  Sets the value for the given parameter.

- **GetParamList()**
  Returns the list of existing parameters for the current Simulink Block.

- **GetFullName()**
  Returns full hierarchy path of the **ArkSimulinkObject** object.

- **SetFullName(fullPath)**
  Sets the full simulink name (full hierarchy path) of the **ArkSimulinkObject** object.

- **SetSimRef(str)**
  Sets the Simulink object type (can be used for referencing objects in other models for future export).

- **GetSimHandle()**
  Returns Simulink reference handle as Double.

- **GetLines()**
  Retuns the list of existing lines as **ArkSimulinkConnection** objects.

- **ExportBlock()**
  Exports a block hierarchy to a model which reference is stored in FullName. It is important that FullName should be provided before calling this method.

- **AddLink(SrcArkSimulinkObj, DstArkSimulinkObj, SrcPort (string), DstPort (string), name (string))**
  Adds and returns **ArkSimulinkConnection** object which is a link between **SrcArkSimulinkObj** and **DstArkSimulinkObject** child objects. **SrcPort** and **DstPort** should refer to Simulink Port definition rules.

- **ImportChildren(import_properties = "'FollowLinks', 'on', 'LookUnderMasks', 'on'")**
  Imports children for a given object, referenced by FullName

ArkVariant objects

ArkVariant objects represent *variants*, defined in the system. These objects are returned by the global functions **GetVariantList()**, **GetVariantByName()**, **GetActiveVariant()**, **GetDefaultVariant()**. To get a root variant object, use global function **GetRootVariant()**.

Methods

- **GetName()**
  Returns name of the variant as string.

- **GetId()**
  Returns unique identifier of the variant as string.

- **IsFolder()**
  Returns **True** if variant is folder, **False** otherwise.

- **IsDefault()**
  Returns **True** if variant is set as default, **False** otherwise.

- **GetParent()**
  Returns parent variant as ArkVariant object.

- **GetChildList()**
  Returns list of child ArkVariant objects.

- **SetName(name)**
  Sets variant name. **name** cannot contain the following symbols : '\ / : * ? " < > |' - in this case the method raises a **ValueError** exception.

- **SetDefault(is_default)**
  Sets variant as default. Argument **is_default** must be boolean.

- **NewVariant(name, is_folder = false)**
  Creates a new ArkVariant object, adds it under current variant and returns it. Argument **name** must be string, **is_folder** must be boolean.

- **MoveVariant(ark_variant)**
  Moves **ark_variant** under the current variant object. Argument **ark_variant** must be valid ArkVariant object instance.

- **GetChoice(ark_choice)**
  Returns checked status of the choice. Argument **ark_choice** must be valid ArkChoice object instance. Returned value is **True** or **False**. Requires read access to the matrix.

- **SetChoice(ark_choice, is_checked)**
  Sets the "checked" status of the given choice in the variant. Argument **ark_choice** must be valid ArkChoice object instance ,**is_checked** must be boolean. Requires write access to the matrix.
- **CopyVariant(name, ref_variant)**
  Creates a new variant as a copy of the variant **ref_variant**, puts it under the current variant object and returns it. Argument **name** must be string, argument **ref_variant** must be a valid non-folder ArkVariant object instance.
- **MergeVariants(name, ref_variants_list)**
  Creates a new variant as a merge of the variants definition of **ref_variants_list**, puts it under the current variant object and returns it. Argument **name** must be string, argument **ref_variants_list** must be iterable container of valid non-folder ArkVariant object instances.

ArkVariantFilter objects

ArkVariantFilter objects can be used for transparent for scripts, scoped filtering of the **arKItect** objects by variant and/or phase. Each ArkVariantFilter object represents an instance of the variant/phase filter, and can be in 2 states: enabled or disabled. When variant/phase filter is disabled, it

does not affects anything. When filter is enabled, then Python API methods will only return objects, visible in the corresponding variant and/or phase. New filter objects can be created with the global function **pyark.VariantFilter(variant, phase = None)**. Several filters can be active in the same time, their effect is combined by the logical AND.

Despite ArkVariantFilter objects support manual enabling or disabling, this approach is discouraged. Python **with** operator should be used instead. Typical usage pattern is:

```
...
#This code is not affected by any variant or phase
...
with pyark.VariantFilter( variant ):
   ...
   #This code only sees objects, visible in the variant
   ...
...
#This code is not affected by any variant or phase again
...
with pyark.VariantFilter( None, phase ):
   ...
   #This code only sees objects, visible in the phase
   ...
...
#This code is not affected by any variant or phase again
...
with pyark.VariantFilter( variant, phase ):
   ...
   #This code only sees objects, visible in the variant and in the phase
   ...
...
#This code is not affected by any variant or phase again
...
```

The `with` statement automatically enables filter (unlike simple creation of ArkVariantFilter object when user needs to enable it explicitly), when control reaches the with block, and guarantees that the filter would be disabled, when execution of this block finished. Filter will be correctly disabled even if execution was finished with exception.

Methods, supporting with statement

These methods usually should not be called manually, the `with` operator calls them automatically.

- **__enter__()**
  Support for "with" operator. Enables filter. Attempt to enable same filter twice raises **ValueError**. Returns self.
- **__exit__()**
  Support for "with" operator. Disables filter. If filter is already disabled, then **ValueError** is raised.

Methods for controlling filter state

Use of these methods is discouraged. The `with` statement should be used instead.

- **Enable()**
  Enables filter. Does nothing, if filter is already enabled.
- **Disable()**
  Disable filter. Does nothing, if filter is already disabled.

Methods to get filter information

- **GetId()**
  Returns either a tuple of two Ids: (ID of the variant, ID of the phase) used by the filter or None if filters are not set.
- **IsEnabled()**

Returns filter enabled status (**True** or **False**).

- **__str__()**
  Printable information about filter.

ArkConnection Objects

AnArkConnection represents a software connection to arKItect server. It has no constructor and can only be retrieved from global functions **pyark. OpenConnection(...), pyark.GetCurrentConnection().**

- **GetWorkspacesList()**
  Returns list of workspace names (strings) of all workspaces visible to the user. Use **OpenWorkspace(name)** to access them.

- **OpenWorkspace(name)**
  Returns an **ArkWorkspace** by name. If there is no such workspace, or workspace is not visible to the user, **ValueError** is raised.
- **OpenRevisionsWorkspace()**
  Returns an **ArkWorkspace** object for the special "REVISIONS" workspace. This object supports additional methods and is intended for storing project revisions (stored as copies). This object is excluded from the list, returned by the **GetWorkspacesList**() method.
- **IsHttps()**
  Returns True, if connection is established via secure HTTPS connection. See 3rd argument to the **pyark.OpenConnection()**.
- **GetLogin()**
  Returns login name for the given connection, as string.
- **GetBusinessSpaceId()**
  Returns internal business space identifier for the connection. Guaranteed to be non-empty string.
- **GetCurrentWorkspace()**
  Must be called for the **ArkConnecton** object, returned by the **pyark.GetCurrentConnection()** method. Returns **ArkWorkspace** object for the workspace, where currently open project resides. Raises exception if failed.

ArkWorkspace Objects

An ArkWorkspace represents an arKItect workspace.

- **GetName()**
  Returns name of the workspace (string).

- **GetID()**
  Returns internal identifier of the workspace (string).

- **GetUserStatus()**
  Returns value, equal to one of the following constants:
  - pyark.STATUS_DEVELOPER
  - pyark.STATUS_DEVELOPER_ADMIN
  - pyark.STATUS_DESIGNER
  - pyark.STATUS_READONLY

- **GetProject(name)**
  Returns ArkProject objects, representing the project with given name. Raises **ValueError**, if no such project found.

- **NewProject(name)**
  Creates new project and returns corresponding ArkProject object. Raises a **ValueError** exception, if such project already exists, or the name is invalid. Raises **pyark.ArkAccessRightViolation** exception, if project creation is not allowed. Name or value cannot cannot exceed 50 symbols or contain any of the following characters: \/:*?"<>|

- **GetProjects()**
  Returns list of the ArkProject objects, for each project in the workspace.

- **DeleteProject(project)**
  Tries to delete a project in the specified workspace. Argument may be either a string (name of the project to delete), or an **ArkProject** ins tance, belonging to the specified workspace. Raises **pyark.ArkAccessRightViolation** if user has no permission to delete.

- **CopyCurrentProject(remote_name, variant=None)**
  Copies *currenlty active* project (the project, currently opened in the arKItect), to the workspace. Raises **pyark.ArkAccessRightViolation** if user has no permission to create or copy projects. Returns list of warning messages received during copy creation, each message as a string.

Optional argument **variant** must be an instance of **ArkVariant** object. If specified, only part of the project, visible in the specified variant, is copied (currently implemented via copying whole project and removing invisible part).

- **LightCopyCurrentProject(remote_name)**
  Creates revision (copy) of the *currently active* project. This method must be called for the REVISIONS workspace object, returned by **OpenRevisionsWorkspace** method. Any other use is obsolete. Returns list of warning messages received during copy creation, each message as a string.
- **RestoreRevision(revision_name, new_name)**
  Restore revision of the project, stored in the special "REVISIONS" workspace. This method must only be called for **ArkWorkspace** objects, returned by the method **OpenRevisionsWorkspace** of the **ArkConnection** object. For any other **ArkWorkspace**, **ValueError** is raised.
  Argument **revision_name** must be a name of the revision (project) in the REVISIONS workspace, argument **new_name** is the name of the restored project.
  Revisions are created by the **LightCopyCurrentProject** method, called for REVISIONS ArkWorkspace object.

ArkUser objects

ArkUser object represents a reference to a **arKItect** user, allowing to change permissions.

ArkUser methods:

General methods

- **GetName**()
  Returns user name.
- **GetType ()**
  Returns the user type by WAM. Returns one of: STATUS_DEVELOPER; STATUS_DEVELOPER_ADMIN; STATUS_DESIGNER; STATUS_READONLY
- **FlushInfo ()**
  This function stores made permission changes to DB. Must be called permission modification.
- **GetGroup ()**
  Returns the name of assotiated group.

Working with permissions

- **GetAUAPermission ()**
  Returns the AUA permission type (True/False).
- **SetAUAPermission(bool)**
  Sets the AUA permission type (allow/disallow).
- **GetDeleteProjectPermission ()**
  Returns the Delete Project permission type (True/False).
- **SetDeleteProjectPermission (bool)**
  Sets the Delete Project permission type (allow/disallow).
- **GetNewProjectPermission ()**
  Returns the NewProject permission type (True/False).
- **SetNewProjectPermission (bool)**
  Sets the New Project permission type (allow/disallow).

Working with rules

- **AddRule** (name)
  Creates an permission association object for the named Project. Returns ArkRule object

Available APIs

- ArkRule objects

ArkRule objects

ArkRule object allows to set permissions for appropriate pair user-Project.

***ArkRule* methods:**

- **GetProjectName ()**
  Returns the name of assotiated project

- **SetCopyPermission (bool)**
  Sets the Copy permission for a project (allow/disallow).
- **GetCopyPermission ( )**
  Returns the Copy permission for a project.
- **SetAccessType (type)**
  Sets the Access Type for a project. Possible values: RIGHTS_NOT_VISIBLE; RIGHTS_READ_ONLY; RIGHTS_FULL_ACCESS
- **GetAccessType ( )**
  Returns the Access Type permission for a project.
- **SetFilterAccess (FilterName , type)**
  Sets the permission for a named filter. Possible values for type: RIGHTS_NOT_VISIBLE; RIGHTS_READ_ONLY; RIGHTS_FULL_ACCESS
- **GetFilterAccess ( )**
  Returns the permission for a given filter.

ArkGroup objects

ArkGroup object represents a reference to a **arKItect** user group, allowing to manage it and change group permissions.

ArkGroup methods:

General methods

- **GetName ()**
  Returns the group name
- **SetName (name)**
  Renames the group with a given name
- **FlushInfo ()**
  This function stores made permission changes to DB. Must be called permission modification.

User management methods

- **AddUser (name)**
  Add named userto the group.
- **RemoveUser (name)**
  Removes named user from the group.

Rule management methods

- **AddRule** (name)
  Creates an permission assotiation object for the named Project. Returns ArkRule object

ArkTab objects

ArkTab objects represent a handle to a graphical tab in the arKItect client's main frame. They support the following methods :

- **GetName()**
  Returns the current caption of the tab

- **SetName(name)**
  Sets the caption of the tab. If **name** is already the caption of another ArkTab object, a ValueError will be raised.

- **TopWidget()**
  Returns The PyQt widget shown in the tab. Use it as a parent for the windows you need to embed into the tab.

- **SetActionState(actionId, enabled, checked = False)**
  Defines the state of the arKItect's toolbar component identified by the **actionId**, whether it is **enabled** or not (True or False) and whether

it is **checked** or not (True or False).
Here are the following values possible for **actionId** :

Action ID	Matching Icon
ACTION_FILE_NEW	
ACTION_FILE_OPEN	
ACTION_EDIT_CUT	
ACTION_EDIT_COPY	
ACTION_EDIT_PASTE	
ACTION_APP_ABOUT	
ACTION_REFRESH	
ACTION_DICTIONARY	
ACTION_SEARCH	
ACTION_SHOW_PROPS	
ACTION_SHOW_LOCATION	
ACTION_SHOW_VARIANTS	
ACTION_VIEW_TASK_PANE	
ACTION_ADD_PANE	
ACTION_TREEVIEWS_COMBO	*Combobox*
ACTION_ACTIVATE_TREEVIEW	
ACTION_VIEW_PREV	
ACTION_VIEW_NEXT	
ACTION_VIEW_UP	
ACTION_LOCALIZE_OBJECTS	
ACTION_OD_DRAW_SELECT	
ACTION_OD_LINK_COMPONENTS	
ACTION_SAVE_AS_JPG	
ACTION_HIDE_LABELS	
ACTION_SYNCHRONIZE_VIEWS	
ACTION_AUTOLAYOUT_H	
ACTION_AUTOLAYOUT_V	

ACTION_AUTOLAYOUT_LEGACY	
ACTION_AUTOLAYOUT_HTREE	
ACTION_AUTOLAYOUT_VTREE	
ACTION_AUTOLAYOUT_OUTERPORTS	
ACTION_RESET_OUTERPORTS	
ACTION_ZOOM	
ACTION_ZOOM_FIT	
ACTION_ZOOM_SELECTION	
ACTION_PAN	
ACTION_GRID	
ACTION_SNAP_TO_GRID	
ACTION_PAGE_BOUNDS	
ACTION_NUDGE_DOWN	
ACTION_NUDGE_LEFT	
ACTION_NUDGE_RIGHT	
ACTION_NUDGE_UP	
ACTION_ALIGN_BOTTOM	
ACTION_ALIGN_CENTER	
ACTION_ALIGN_LEFT	
ACTION_ALIGN_MIDDLE	
ACTION_ALIGN_RIGHT	
ACTION_ALIGN_TOP	
ACTION_SPACE_ACROSS	
ACTION_SPACE_DOWN	
ACTION_SAME_WIDTH	
ACTION_SAME_HEIGHT	
ACTION_SAME_SIZE	
ACTION_CHOICE_MNGR	
ACTION_VARIANT_MNGR	

ACTION_VARIANT_SUM	
ACTION_VARIANTS_COMBO	*Combobox*

- **OnAction(actionId, functor)**
  Binds **functor** to the click of the arKItect's toolbar button identified by **actionId**. **functor** needs to be a callable with an arity of 1. This ArkTab object will be passed as a parameter when **functor** is called.
  actionId can be any of the values enumerated previously except ACTION_TREEVIEWS_COMBO and ACTION_VARIANTS_COMBO since they are not buttons.

- **OnFocus(functor)**
  Binds **functor** to the focus event of the tab. **functor** will therefore be called when the tab gets the focus (most likely due to a click on the tab).
  functor needs to be a callable with an arity of 2. This ArkTab object and the current ArkObjRef object will be passed as parameters when **functor** is called.
  Note that a focus event can be generated by a click on the tab, a change of projection, a change of variant or even a change of object in the projection.

- **OnDestroy(functor)**
  Binds **functor** to the destroy event of the tab. **functor** will therefore be called when the tab gets removed.
  functor needs to be a callable with an arity of 1. This ArkTab object will be passed as a parameter when **functor** is called.

- **OnActivate(functor)**
  Binds a callback (functor) that is called when tab is activated or deactivated. The **functor** must be a callable taking 2 arguments:
    - The ArkTab object, which was activated or de-activated
    - Boolean value, True if tab was activated, and False otherwise.

- **Activate()**
  Activates tab.

ArkBusyDialog objects

ArkBusyDialog is used to show a waiting dialog when executing long tasks. It is found in the "pyark.ui" module and it supports the following methods :

- **ArkBusyDialog(message = "")**
  Constructs a dialog object with the given message

- **Show()**
  Displays the dialog on top of every other displayed windows

- **Hide()**
  Hides the dialog

- **Message()**
  Returns the current message of the dialog

- **SetMessage(message)**
  Sets the message of the dialog

Some magic methods are also defined to make the usage of this class easy :

```
import pyark

with pyark.ui.ArkBusyDialog("doing some long task...") :
    doSomeLongTask()
```

or even :

```
import pyark

with pyark.ui.ArkBusyDialog() as dlg :
    for i, data in enumerate(dataContainer) :
        dlg.SetMessage("doing task %d of %d" % (i, len(dataContainer)))
        doSomeTask(data)
```

Custom View Filters

The "custom view filter" functionality provides possibility to modify visibility of child objects (direct and indirect) using Python callbacks (**CustomViewFilter** event handlers). This functionality is supported by the following features:

- **CustomViewFitler** event handler (**pyark.EVENT_ONCUSTOMFILTER**).
  Defines Python callback that modifies child object visibility.
- Several API methods:
  - **ArkObjRef.ReloadCustomFilter()**
    Clears currently loaded custom fitler data fom the object, causing it to be reloaded on the next access to the object's children.
  - **TreeViewObj._EnableCustomFilterEnabled(isEnabled::bool)**
    Allows script to temporarily disable (False) and enable back custom filter in a specific tree.
  - **TreeViewObj._IsCustomFilterEnabled()**
    Returns activity state of the custom filter. Default is True, could be set to False by the above method.
- Library functions
  - **arkiext.ki.chains.generic_chains.disableChains(arktreeviewobj)**
    A context manager, that should be used instead of calling methods **_EnableCustomFilterEnabled, _IsCustomFilterEnabled** directly. Guarantees that custom filter will be enabled back, supports nesting.

Custom View Filter basics

Custom view filters can hide child objects, that normally would be displayed in the view, thus the name "filter": they only filter existing data. This is done for the sake of consistency with meta-model. Hiding mechanics works in the same way as hiding by ENUM attributes. Particularly, in the hierarchy of objects of the same type, when parent is hidden then its children appear on its place.

Filtering is done by *whitelisting*: OnCustomFilter handler returns list of objects that are only allowed among children. Any object not in this list would be hidden.

OnCustomFilewFilter event handler

This event handler must be defined on the parent object of a hierarchy, where children are being hidden. It must define a single Python function, returning a dictionary with several fields. It **must not** modify any data in the architecture. Event filter handler is executed, when filtered subtree is accessed for the first time. For performance reasons, results are stored in arKitect session memory indefinitely, until **ArkObjRef.ReloadCustomFilter** is called.

Returned dictionary must have the following fields with string names:

- **"whitelist"** : required. Python list of **ArkObj** or **ArkObjRef** objects. Only objects in this list are visible in the hierarchy. This list can contain objects that are not present in the hierarchy at all, they will be ignored. This field os the only required field of the retured value. Other field are used to provide additional functionality.
- **"commands"**: list of "command descriptions": dictionaries that describe additional commands, shown in context menus for the objects in the filtered hierarchy. Each "command description" is a dictionary with the following fields:
  - **"name"** : string, required. Human-readable name of the command.
  - **"handler"** : Python callable with 1 required and 1 optional argument: same as "Program" attribute handler. When custom action is executed from context menu, this handler is called; with first argument receiving **ArkObjRef** reference to the object, and optional argument is either not present or a dicitonary with additional arguments.
  - **"flags"** : optional, string. Several characters, defining when the command is present in the menu. Supported flag characters are:
    - 'o' - command is shown for "box"objects
    - 'l' - command is shown for links. When clicked on GUI link, that can contain several flow objects, additional argument of the hander has "siblings" field, that contains list of **ArkObj** references to the flows in a links.
    - 'm' - command is shown for multiple selection. WHen handler accepts additional argument, it has "siblings" field, that

contains list of **ArkObj** references to all selected objects.

Example handler:

sample custom view filter

```
def run(self):
    def onObject(obj):
        print "CUstom command called for:", obj
    def onLink(obj, arg=None):
        print "Custom command called for:", obj, "additional args are:", arg
    return { 'whitelist' : [pyark.GetArkObj('A','A1'), pyark.GetARkObj('A', 'A2')],
             'commands'  : [
                 {'name' : "Custom Command for objects",
                  'handler' : onObject,
                  'flags': 'o'},
                 {'name' : "Custom Command for links",
                  'handler' : onLink,
                  'flags': 'l'}
             ] }
```

Available libraries

We developed some python libraries to answer some current needs. You might find useful to rely on it instead of redeveloping what we did.

- advanced - package containing advanced features that are managed by script
- arkiexport - library for exporting and importing parts of architecture to XML files.
- arkimatrixexport - library for exporting and importing parts of Matrix and TreeViews to xml files.
- arkivariantsexport - library for exporting and importing parts of choices and variants structure to xml files.
- graph - package with libraries for manipulating graphical objects in arKitect views.
- arkilayout - library for organizing graphs' layouts.
- projectsmerger - library for merging two projects in a new one.
- matrixupdater - library for updating matrix from a matrix export.
- MIC Model - package to manage Model Identity Card data
- utils - package containing library of additional Python functions for arKitect, simplifying routine tasks.
- ui - libraries for working with graphical interface.
- word_template - library to easily generate Microsoft Word documents from arKItect.

utils

utils is a Python package, containing miscellaneous Python extension module.

This package contain:

- arkiutils - library of additional Python functions for arKitect, simplifying routine tasks.
- arkiexcel - library of additional Python functions for Excel.
- xmltodict - library to transform XML file to python dictionary and vice versa

arkiexcel

arkiexcel.py is a Python extension module, containing miscellaneous utility Python functions for arKItect.

In order to use it in scripts, put the module to the arKItect folder and use **import arkiexcel** to access it's functions.

Functions in the arkiexcel module

Excel(filename)

Return an object which represent the excel file.

```
from arki.utils import arkiexcel

...

excelPath = ...
excel = arkiexcel.Excel(excelPath)
```

SheetCount

Return the number of sheets in the document

```
print excel.SheetCount()
```

SelSheet(sheetNumber)

Set the active sheet to the sheet corresponding to the sheetNumber. For example, with the following code, active sheet will be the 2nd one

```
excel.SelSheet(2)
```

GetActiveSheetName()

returns the name of the active sheet (defined by SetSheet)

GetCellValue(row, col)

Return the value of cell at row, column position. For example, the following code will print the value of cell B1

```
print excel.GetCellValue(1,2)
```

SetCellValue(row, col, value)

Set the value of cell at row, column position. For example, the following code will set the value of cell B1 to "new value"

```
excel.SetCellValue(1, 2, "new value")
```

GetLastRow

Return the last none empty row

```
print excel.GetLastRow()
```

GetLastCol

```
print excel.GetLastCol()
```

Save

Save active Excel document

```
excel.Save()
```

SaveAs(newName)

Save active Excel document under the new name. newName must be a valid path

```
excel.SaveAs(r'C:\newExcel.xlsx')
```

arkiutils

arkiutils.py is a Python extension module, containing miscellaneous utility Python functions for arKItect.

In order to use it in scripts, put the module to the arKItect folder and use **import arkiutils** to access it's functions.

Functions in the arkiutils module

CountChildren(object)

Returns number of children for given **object**. For example, the following code will return number of objects in the architecture, under tree named "all":

```
import pyark
from arki.utils import arkiutils
...
print arkiutils.CountChildren(pyark.GetRoot("all"))
```

FindObject(treeName, name, arkType=None)

This function retrieves reference to an object by its name. Object type may be specified to distinguish between several objects with same names, but different types (such situation is allowed by arKItect rules). If object type is not specified, function returns first matched object. This function searches whole object tree. Example:

```
import pyark
from arki.utils import arkiutils
...
obj = arkiutils.FindObject("all", "MyObject") #find object named "MyObject" in the
tree "all"
```

FixNames(activeTree, chars, replacements, flags='oa')

For all objects in **activeTree**, replaces characters (strings) from list **chars** with characters (strings) from list **replacements**.**chars** and **replaceme nts** must be lists of same length. **flag**' argument specifies, which parts of architecture should be processed. It must be an ASCII string, containing following characters:'a' - attributes will be fixed 'o' - object names will be fixed. Default is 'ao'

GetAllChildren(object, onlyTypes = None)

For the given root **object**, this function returns list of all underlying objects, including root itself. Only unique objects are counted.

onlyTypes : list of filtered types we want to get, if onlyTypes is None, all objects are returned
This is a generator function, use **list(GetAllChildren(obj))** to get the list

Example1 : get all object under a root

```
import pyark
from arki.utils import arkiutils
...
root = pyark.GetRoot(view)
for obj in arkiutils.GetAllChildren(root):
...
```

Example 2: get all Requirements, Functions under a root

```
import pyark
from arki.utils import arkiutils
...
root = pyark.GetRoot(view)
for obj in arkiutils.GetAllChildren(root, ['Requirement', 'Function']):
...
```

GetObjectByPath(root, path)

Walks object tree, starting from given **root**. **path** must be "/"-separated list of object names. For example, following code will put reference to "STM/state/flow" object into "obj" variable:

```
import pyark
from arki.utils import arkiutils...
root = pyark.GetRoot("all")
obj = arkiutils.GetObjectByPath(root, "STM/state/flow")
```

This function does not performs full tree search and therefore works faster than **FindObject**, especially for the large trees.

GetDirectParentsOfTypeAllTrees(objectRef)

Returns list of (treeview, parent, direction) for a given object with:

- **objectRef** - *ArkObjRef* of an object found in any treeview.
- **treeview** - *ArkTreeViewObj* where the parent is found.
- **parent** - *ArkObjRef* of parent.
- **direction** - the direction of the **objectRef**, can be "input", "output" or **None**.

```
from arki.utils import arkiutils...
obj = arkiutils.FindObject("all", "System_name", "System_type")
for item in arkiutils.GetDirectParentsOfTypeAllTrees( obj ):
    print item[0].GetName()
    print item[1].GetName()
    print item[2]
    print "---"
```

SimplifiedChangeType(objectRef, newType)

Change type for a given object with the **newType** parameter in all treeviews.

```
from arki.utils import arkiutils...
obj = arkiutils.FindObject("all", "System_name", "System_type")
arkiutils.SimplifiedChangeType( obj , "System_newtype" )
#System_name(System_type) change type in System_name(System_newtype)
```

CheckChildCompatibility(objectRef, checkType)

Check if a given object (**objectRef**) can have a child of given type (**checkType**)

```
from arki.utils import arkiutils...
obj = arkiutils.FindObject( "all", "System_name", "System_type" )
arkiutils.CheckChildCompatibility( obj, "s" )
#Check if obj can have a child of s type
```

CheckAllCompatibility(objectRef , typesToRemove)

Check if we can remove object of types in the list **typesToRemove** under the **objectRef** object and move children up with CheckChildCompatibility.
This function return **True** when it's ok else **False**.

```
from arki.utils import arkiutils

result =
arkiutils.CheckAllCompatibility(self,[arkiutils.FindObject("all","U").GetArkType(),ark
iutils.FindObject("all","T").GetArkType()])
print "The result of test is",result
```

This code check if we can remove all objects of same type as "**U**" object and "**T**" object under the **self** object.

makeParentsListGetter(rootArkObjRef)

Alleviates the inability to effectively get list of all parents of ArkObjRef. Creates a function, that takes ArkObjRef and return list of parent ArkObjRef's. Example usage:

```
#obj is and ArkObjRef instance
#if possible, use some deeper object instead of root, this would reduce memory usage
and improve performance.
root = pyark.GetRoot( ... )
GetParents = makeParentListGetter( root )
print "Parents of", obj
for parent in GetParents(obj):
    print parent
```

Font management

Font properties are stored in the following format:

```
[face]Arial[/face][size]-15[/size][italic][underline][bold][s]
```

To simplify manipulation of these strings, 2 helper functions are available:

- **MakeFontStr(name, size, bold=False, italic=False, underline=False, strike_out=False)**
  Creates font string in the format, accepted by arKItect properties. Use in conjunction with SetProperty method
- **ParseFontStr(font_string)**
  Parses font string, returns dictionary with the following keys:

Key (string)	Value type	Meaning
name	str	Font face
size	int	Font size
italic	bool	**Italic** font (may be absent)
bold	bool	*Bold* font (may be absent)
underline	bool	Underlined font (may be absent)
strike_out	bool	~~Strike~~ font (may be absent)

These key names correspond to the argument names of **MakeFontStr** function.
The call **MakeFontStr(**ParseFontStr(font_string))** will return the same font_string.
In case of parsing failure, raises **ValueError**.

Sample usage of these functions, setting font property to the font "Parchment" with size 15 and other properties set to default.

```
font = MakeFontStr("Parchment", 15)
   func.SetProperty(pyark.ATYPE_GEN_TAG_ATRRIB_FONT, font)
```

RenderRichTextToImage(filename, rtfText, width [, height])

Rich Text attributes are available only with **arKItect** 2.2 and above

Render Rich Text data to a picture and save it. The file format is detected from the file extension. Valid extensions are:

- **bmp**
- **png**
- **gif**
- **jpg** (or **jpeg**)

The width has to be specified. If the height is omitted, it will be automatically adjusted to render the full data. Otherwise, if the height is specified, the image will be either truncated or extended with a white background.
Width and height are in pixels, and must be strictly positive.
If the file already exists, it will be overwritten.
If the data is not valid RTF, the result is unspecified, but will probably provide a blank image.
Returns True on success, False otherwise.

GetPlainTextFromRichText(richtext)

GetRichTextFromPlainText(plaintext)

Rich Text attributes are available only with **arKItect** 2.2 and above

The first function converts richtext data to plain unformatted text. The fonts, color, alignment, images and so on are dropped. You get the same result as if you had pasted the formated text in Notepad.
The second function does the opposite: it converts plain text to richtext, using a default formatting, to be compatible with the Rich Text attribute.

GenerateBlockIconFile(filename, iconData)

GenerateFlowIconFile(filename, iconData)

Dynamic icons are available only with **arKItect** 2.3 and above

This function generates an icon file (*.ico*) as specified by first parameter from the icon data obtained with the icon property of a type, as in this example:

```
import pyark
from arki.utils import arkiutils
data = pyark.GetArkMatrixType("foobar").GetProperty(pyark.ARK_ATYPE_ICON)
arkiutils.GenerateBlockIconFile( r"c:\foobar.ico", data )
```

It returns True on success, False otherwise. It works only with dynamic Skin-based icons, any other kind of icon will make the function fail. The file name will be treated as-is, even if it hasn't the *.ico* extension.
Use the first function if the type is considered a non-flow type, the second otherwise. This cannot be automatically detected since a type may be both flow and non-flow depending on its location in the Metamodel.

get_all_flows(object, flow_types)

This function return tuple of 3 lists:

- list of produced flow
- list of consumed flow
- list of internal flow (ie = flow produced and consumed by objects under object given in parameter)

produced flows / consumed flows don't need to be produced/consumed by object itself : can be a child

```
obj = ...
print au.get_all_flows(obj, [flow_type, flow_type2 ... ])

#([<ArkObjRef object at 0x0E85C500>], [<ArkObjRef object at 0x0E85C420>, <ArkObjRef object at 0x0E85C4A0>], [<ArkObjRef object at 0x0E85C4E0>])
```

Example :

```
from arki.utils import arkiutils as au
 obj = au.FindObject("1.1.3. Draw External Scenarios", "enabling system")
    produced, consumed, internal = au.get_all_flows(obj, "Sequence information")
    for p in produced:
        print p

    print "============"
    for c in consumed:
        print c

    print "============"
    for i in internal:
        print i


Result :
extern (Sequence information)
============
toto (Sequence information)
titi (Sequence information)
============
intern (Sequence information)
```

Linkage class

Class Linkage is an utility to track flows between objects. Current API of arKitect built around ArkObjRef instances does not provide a straightforward way to do this.

Linkage class

```
class Linkage:
    def __init__( self, link_types, object_walker ):
  ...
    def linked_from( self, source_obj, return_link=False ):
        """Returns list of the objects, linked with the given one in specified
direction"""

    def linked_to( self, dest_obj, return_link = False ):
        """Returns list of the source objects for the given destination one"""

    def linked( self, source_obj ):
        """Returns list of all objects,m linked to the given one"""
    def get_flow_producers( self, flow ):
        ...
    def get_flow_consumers( self, flow ):
        ...
```

Usage pattern is:

1. Create **Linkage** instance, specifying flow types you are interested in, and providing sequence of objects to look at (usually using **walk_objects** function). This is expensive procedure, and must not be done many times. Try to reuse **Linkage** instance if possible. Example:

   ```
   linkage = Linkage( ['flow'], walk_objects( root_arkobjref ) )
   ```

2. Now use the Linkage instance to get answers to questions:
   a. What objects are linked by ingoing flows to this object **obj**:

      ```
      for producer_obj in linkage.linked_from( obj ): ...
      ```

   b. What objects are linked by out flows to this object **obj**:

      ```
      for consumer_obj in linkage.linked_to( obj ): ...
      ```

   c. Above methods can return flow object too:

      ```
      for producer_obj, flow in linkage.linked_from( obj, return_link=True ): ...
      for consumer_obj, flow in linkage.linked_to( obj, return_link=True ): ...
      ```

   d. Alternatively, you can get list of producer or consumer objects, if flow object is known:

      ```
      for producer_obj in linkage.get_flow_producers( flow_obj ): ...
      for consumer_obj in linkage.get_flow_consumers( flow_obj ): ...
      ```

IterHistory(dateFrom, dateTo, user=None, hpidFrom=None, hpidTo=None)

Generator version of **pyark.GetHistory**, returning whole history in the specified range. Contrary to **pyark.GetHistory**, returned value is generator, not list, and iterates over the whole diapason without limitation of 1000 records.

xmltodict

xmltodict.py is a Python extension module, containing xml and dictionnary transformation utility Python functions.

It's purpose is to convert an XML file into a python dictionary and vice versa.

It's an open source library, you can follow this link to discover the entire changelog, issues and future updates : xmltodict

In order to use it in scripts use "***from arki.utils.xmltodict import xmltodict***" to access it's functions.

In this module you should be using only 2 functions.

Functions in the xmltodict module

*parse(xml_input, encoding=None, expat=expat, process_namespaces=False, namespace_separator=':', **kwargs)*

Returns an ordered python dictionary (OrderedDict) for a given XML input. This XML input can be a string (which format is an XML format) or a file-like object (which is the return value of open function for example).

For example, the following code will return a python dictionary which data comes from a file :

```
import os
from arki.utils.xmltodict import xmltodict

sampleFile = open(os.path.expanduser(r'~\\Desktop\\') + 'sampleFile.xml')
sampleData = sampleFile.read()
sampleFile.close()
sampleDict = dict(xmltodict.parse(sampleData)) # the return is an OrderedDict, if you
wish, you can cast it to dict as in this example
```

You can choose your needed encoding, by default 'utf-8' is selected.

There is another example with a given string with XML format :

```
import os
from arki.utils.xmltodict import xmltodict

sampleData = ('<a prop="x">'
     '<b>1</b>'
     '<b>2</b>'
     '</a>')
sampleDict = dict(xmltodict.parse(sampleData)) # the return is an OrderedDict, if you
wish, you can cast it to dict as in this example
print(sampleDict)
>>> {
 "a": {
  "@prop"="x"
  "b": [
   "1",
   "2"
  ]
 }
}
```

A complete description of arguments is not available, you should investigate source code.

unparse(input_dict, output=None, encoding='utf-8', full_document=True, **kwargs)

You should provide a dictionary to this function. The input dictionary can contain dictionaries, lists and nodes of data.

This function does not necessarily return a value. If parameter output is not provided, it returns a string which contains XML data. Elsewise if you give it a path as an output it does not return and write the data into the file.

For example, the following code will return an XML string which data comes from a python dictionary then write the XML string into a file :

```
import os
from arki.utils.xmltodict import xmltodict
import xml.dom.minidom

sampleData = {
    "a": {
                "@prop": "x",
                "b": [
                 "1",
                 "2"
                 ]
                }
             }
xmlData = xmltodict.unparse(sampleData)
xmlDoc = xml.dom.minidom.parseString(xmlData)
xmlDataPretty = xmlDoc.toprettyxml()
xmlFile = open(os.path.expanduser(r'~\\Desktop\\') + 'sampleFile.xml', 'w+')
xmlFile.write(xmlDataPretty)
xmlFile.close()
```

You can choose your needed encoding, by default 'utf-8' is selected.

There is another example with a given string with XML format (care, this way the xml file will be written as flat content without newlines like this : "<?xml version="1.0" encoding="utf-8"?><a>12<prop>x</prop>") :

```
import os
from arki.utils.xmltodict import xmltodict
import xml.dom.minidom

sampleData = {
    "a": {
                "@prop": "x",
                "b": [
                 "1",
                 "2"
                 ]
                }
             }
sampleFile = open(os.path.expanduser(r'~\\Desktop\\') + 'sampleFile.xml', 'w+')
xmltodict.unparse(sampleData, sampleFile)
sampleFile.close()
```

A complete description of arguments is not available, you should investigate source code.

You can notice in the examples that if you need a property "prop" into your node you should declare it "@prop"

utils.revisions

Library that contains miscellaneous utility functions to work with object revisions.

Detecting modifications

modifiedObjects(root, ark_types=None, ignoreObjectsWithoutRevisions=True)

Generator that yields sequence of all objects that have modifications since the last revision. Arguments:

- **root** - ArkObjRef object, where to start search. Usually project root, returned by **pyark.GetRoot()**
- **ark_types** - can be None, name of the rule (string) or list of rules names. Allows to limit search only to objects of specified type.
- **ignoreObjectsWithoutRevisions** - if False, then objects without saved reivisions are also considered modified.

hasModifiedObjects(root, ark_types=None, ignoreObjectsWithoutRevisions=True)

Returns True, if there is at leas one modified object. Arguments are the same as in the **modifiedObjects**.

checkModifiedObjects(views, ark_types=None)

Checks multiple view to find modified objects with saved revisions. If found, revisions dialog is called.

Return value: True if dialog was called.

arkinternal

Module, containing utilities, complementing basic Python API. Many of these functions are essential for development and can not be implemented only with public APIs.

- Data Reader Cache
    - class DataReaderCacheContext
- Network packaging (bulk mode)
    - class BulkModeContext
        - Methods
    - class ArkiUniqueConstraintException(Exception)
        - Methods
- Local Script Execution
    - class LocalScriptContext
- Generate Documentation (Export data) and GUI elements updates
    - class ExportContext

Data Reader Cache

Performance helper when only reading actions are done on multiple objects and they are linked with objects hierarchy or attributes. It can also be used when non enum attributes are changed.

class DataReaderCacheContext

Context manger, that enables data reader cache on *entering* and automatically disables it on *exiting*. Nesting contexts is allowed.

```
from arki.utils.arkinternal import DataReaderCacheContext
...
with DataReaderCacheContext():
    #get multiple objects hierarchy and then check object's attributes, identify object's flows, get object's children
```

To query, whether Data Reader cache is active now, API function **pyark._IsDataReaderCacheEnabled()** is used. It returns boolean value.

Network packaging (bulk mode)

arKItect operates on project data by sending requests to the data server. When modifying large number of objects (including creation, linking, attribute/property/choice modification), one request per modification is sent, which causes significant overhead and low performance. To improve performance, **bulk mode** is provided in arKitect.

When **bulk mode** is enabled, arKItect accumulates requests in memory instead of immediately sending them to server. Requests are sent later in a single or several shots, depending on whether the **bulk mode** is finished or 200 requests are accumulated.

The important consequence is, when **bulk mode** is enabled, some errors that can only be detected on server, are not detected immediately. Currently, unique constraint violation on attributes fall in this category. Programmer must check for such errors separately after exiting context.

class BulkModeContext

Context manager, that enables **bulk mode** on *entering* and automatically disables it on *exiting*. Nesting contexts is allowed.

```
from arki.utils.arkinternal import BulkModeContext
...
with BulkModeContext():
    #modify multiple objects
    #messages are accumulated in memory and not sent to server.
```

Methods

- **__init__**(verbose=False, onConflictRaiseError=True)
  if **verbose** flag is **True**, then context manager prints text messages, when enabled or disabled.
  if onConflictRaiseError is **True**, then on leaving context, the manager will raise an exception of type arki.utils.arkinternal.ArkiUniqueConstraintException, if there were any conflicts, reported by server. Currently, only violations of unique constraint on attributes are reported, but this can be changed in future.

class ArkiUniqueConstraintException(Exception)

Exception type, raised by **BulkModeContext**, if errors were reported from server after committing requests.
Methods

- **raiseNested**()
  If code block, executed in **BulkModeContext**, raised some exception, and there were conflicts, reported after committing changes to server, then **ArkiUniqueConstraintException** is raised, "shadowing" the original exception. Information about the original exception is stored in this object. Programmer may call **raiseNested** to ensure that shadowed exception is not lost.
  Code sample:

```
from arki.utils.arkinternal import BulkModeContext, ArkiUniqueConstraintException

...
try:
 with BulkModeContext():
  ...make some actions, causing conflicts...
     raise ValueError("user error")
except ArkiUniqueConstraintException as err:
 #ArkiUniqueConstraintException shadows ValueError("user error")
 ...process conflicts...
 err.raiseNested() #re-raises ValueError("user error")
```

Local Script Execution

Sometimes it is needed that some actions are done locally on the data model (in the memory) and not sent to the database, e.g. Functional chains. It means that these changes are important for visualization but not for storing them. It happens with scripts (API) when on some event we need to color objects or show / hide some objects. So, information about object colors or objects presence in some chains is stored in some attributes and later when user navigates into specific diagrams, special events are triggered so that needed information is propagated to objects of the diagram and objects are shown/hidden/colored. Clearly there is no need to save applied modifications.

Another useful application of local script execution is to debug some import operations until you got the satisfying result. As soon as import looks ok you may

- replace LocalScriptContext() with BulkModeContext() and run the script again OR
- in case you fill the empty project - export project to arkz and create a new project from that arkz

Starting from arKItect 4.0.3 it has become possible to apply some modifications only locally. This should drastically decrease network load for many actions which are from user point of view are not operations at all.

class LocalScriptContext

Context manager, that enables **Local Script Execution** on *entering* and automatically disables it on *exiting*. Nesting contexts is allowed.

```
from arki.utils.arkinternal import LocalScriptContext
...
with LocalScriptContext():
    #call whatever API is needed which is meant for local execution only
    ...
```

Generate Documentation (Export data) and GUI elements updates

One of advantages of the arKItect solution is that it allows generating reports from the data model. Very often this requires iterative navigation of diagrams so that images are generated and exported. Sometimes thousands of objects are navigated one by one with diagrams being calculated, shown and saved externally. From arKItect point of view this is a specific operation and arKItect should be informed about it somehow. Why?

When user navigates objects manually, arKItect takes care of updating multiple GUI elements: properties, options, location, palette, project tools, displaying treeviews etc. so that they correspond to actual object selection.

When a script navigates through objects it is definitely not the GUI elements which are of interest but usually the diagrams. So it looks logically to disable GUI updates when a script works in this way (navigation).

class ExportContext

Context manager, that enables Generate Documentation mode on *entering* and automatically disables on *exiting*. Nesting contexts is allowed. Available in arKItect 4.0.3

```
from arki.utils.arkinternal import ExportContext
...
with ExportContext():
    #call whatever API is needed for iterating through diagrams
    ...
```

arki_deepcopy

Functions in the arki_deepcopy module

DeepCopy(source, dest, renaming_rule = None)

Create a new object , child of dest (must be an ArkObjRef instance) object This new object will be a clone of source object (must be an ArkObjRe f instance)

Thus, it will have same properties, options and attributes values

Default name is source object name + datetime (date of the day). Parameter "renaming_rule" can be used to customize new object name.

Example 1 : Default name

```
from arki.utils.arkideepcopy import DeepCopy
souce = ...
dest  =  ...
clone = DeepCopy(source, dest)
```

Example 2 : Customized name

```
from arki.utils.arkideepcopy import DeepCopy
souce = ...
dest  =  ...
clone = DeepCopy(source, dest, lambda n: "Clone of " + source.GetName())
```

arkiexport

Python library for exporting parts of architectures to XML and importing XML files.

Export

Export is done with function:

xmlexport(exportRoot, outFileName, export_filter=None, descend_filter=None, export_graph=True):

The arguments are:

- **exportRoot** - root object to export to XML.
- **outFileName** - name of the file to write the exported data
- **export_filter** - filtering predicate, determining, which objects will be exported (see "Filtering" section below). If **None**, all objects are exported.

- **descend_filter** - predicate, determining,should the children of the object be exported or not (see "Filtering" section below). If **None**, children are always exported.
- **export_graph** - boolean flag,

Filtering

The **xmlexport** function can receive two filtering predicates. Both predicates must be Python functions (callable objects), receiving two arguments, for example: def sample_predicate(root, obj):
...
First argument (**root** in the example above) is the root exported object (of type *ArkObjRef*), second argument (**obj** in the example) is the tested arKItect object. The predicate function must return **True** or **False**. If the predicate returned **True**, tested object is exported, if not, then object is ignored.

Predefined predicates

The library arkiexport.py defines several predicates, that are ready for use.

- **filter_type(type_name1, type_name2, ...)**
  This function returns predicate, that is True, when tested object has specified abstract type. For example, following command will export all object of type "flow" or "link":xmlexport(..., export_filter=filter_type("flow", "link")

- **filter_name(name1, name2, ...)**
  This function returns predicate, that is true for objects with given name.

- **filter_flow (flow_code)**
  This function returns predicate, that tests, whether the object is flow or not. The only argument, **flow_code** must be string, containing characters "i", "o" or "n", representing input flows, output flows and non-flows correspondingly. For example:
  - filter_flow ("i") - **True** for input flows
  - filter_flow ("io") - **True** for any flow
  - filter_flow ("n") - **True** for non-flows
  - filter_flow ("ion") - Always True.

Predicate operations

In addition to predefined simple predicates, the library defines several functions, allowing combining the predicates. All these functions receive one or several predicates as arguments and return one new predicate.

- **p_and (pred1, pred2, ...)**
  This function returns new predicate, that is **True**, if all given predicates are **True**. For example, this call will create predicate pred, that is **True**, only if tested object has name "system" and abstract type "System":

  ```
  pred = p_and( filter_name( "system" ), filter_type( "System" ) )
  ```

- **p_or (pred1, pred2, ...)**
  Same as **p_end**, but new predicate is **True**, if at least one of argument predicates is **True**.

- **p_not (pred)**
  Returns predicate, that is **True**, if given predicate is **False** and vice versa.

- **p_parent (pred)**
  Applies predicate to the parent of the object. New predicate is **True**, if given predicate **pred** is **True** for the object's parent. For example, following predicate is **True** for all objects, that are children of object of type "System":

  ```
  pred = p_parent( filter_types( "System" ) )
  ```

- **p_hasparent (pred)**
  Returns predicate, that is **True**, if at least one of object's parents in hierarchy matches given predicate. For example, this predicate is **Tru e** for all objects of type "System" and all objects below them in hierarchy:

```
pred = p_hasparent ( filter_type ( "System" ) )
```

- **p_haschild (pred)**
  Returns predicate that is **True**, if given predicate **pred** is **True** for any child of object. For example, the following predicate is True for object with name "flow2" and all its parents:

```
pred = p_haschild( filter_names( "flow2" ) )
```

User-defined predicates

User can define own filtering functions (predicates), if predefined ones are not enough. For example, following function uses regular expressions to check object name:

```
import re
def user_predicate(root, obj):
    obj_name = obj.GetName()
    return re.match( r"^flow_\d+$" obj_name ) #get only objects, that has name of form
flow_<index>.
```

User-defined predicates can be combined, using the above-mentioned predicate combination functions.

Usage example

The following code exports object "A_2" and its children to the file "out.xml", applying complex filtering. This code should be executed in arKItect.

```
import arki.features.impexp.arkiexport as xx
from arki.utils import arkiutils

def run(self):
    #Import filtering functions and functions for predicate manipulation
    from arkiexport import p_and, p_or, p_not, p_parent, p_hasparent, p_haschild,
filter_type, filter_name, filter_flow

    #Get object "A_2" for export, using arkiutils helper module
    obj = arkiutils.FindObject("all", "A_2")

    #Export object to the file "out.xml", exporting all object, that has at least one
child, that is flow or has name, not equal to "F_2":

    xx.xmlexport(obj, "out.xml",  \
        export_filter=p_haschild( \
            p_or( \
                p_not(filter_name("F_2")), \
                filter_flow("io") \
            ) \
        ) \
    )
```

Import

For importing architecture back from XML, function **xmlimport(importRoot, inFileName)** is used. Its arguments are:

- **importRoot** - object, where imported architecture will be created
- **inFileName** - path to the XML file, containing exported architecture

arkilayout

Python library for organizing graphs' layouts. It requires arkilayout_graph.py, arkilayout_tools.py (which, now, needs arkixmltree.py).

organize

Arkilayout allows to improve a group of views' layout. This is done with the function:

organize(self=None, target=None, treeView=None, Tmax=None, showSummary=True)

The arguments are:

- **self** - *ArkObjRef* focused.
- **target** - can be:
  - "children" : you are interested in the view that contains the children of self
  - "view" : you are interested in the view that contains self
  - "all" : you are interested in all treeViews (if it's not specified or is None, it will be only the treeViews specified by the next argument)
- **treeView** - indicates the name of the treeView concerned.
- **Tmax** - allows to limit the time of computation. It is a time in seconds and the calculation of one single view shouldn't last too much than this time.
- **showSummary** - indicates if you want a computation's output or not.

Usage example

- The following code fully organizes the view that contains the children of the object.

```
from arki.graph import arkilayout
def main(self):
    arkilayout.organize( self=self, target="children" )
```

- The following code fully organizes the view that contains the object.

```
from arki.graph import arkilayout
def main(self):
    arkilayout.organize( self=self, target="view" )
```

- The following code fully organizes all the views in the treeView named "T".

```
from arki.graph import arkilayout
def main(self):
    arkilayout.organize( treeView="T" )
```

  You can add **Tmax** and **showSummary** arguments to specify a time limit or if you want an output or not.

Outer port dividing

outerportmanager allows to divide merged outer ports if possible. That is necessary phase before merging. This is done with the function:

divide_ports(main_obj, skip_objects = [])

The arguments are:

- **main_obj** - *ArkObjRef* contaning graph to be processed.
- **skip_objects** - List of *ArkObjRef* objects. Outer ports assotiated with those arkobjrefs will not be divided

Usage example

- The following code divides all outerports for **self** object except ports assotiated with first child.

```
from arki.graph import outerportmanager
def main(self):
    outerportmanager.divide_ports(self, [self.GetChildList()[0]])
```

Outer port merging

outerportmanager allows to merge outer ports assotiated with a object (per port type). This is done with the function:

merge_ports(main_obj, skip_objects = [])

The arguments are:

- **main_obj** - *ArkObjRef* contaning graph to be processed.
- **skip_objects** - List of *ArkObjRef* objects. Outer ports assotiated with those arkobjrefs will not be merged

Usage example

- The following code merges all outerports for **self** object except ports assotiated with first child.

```
from arki.graph import outerportmanager
def main(self):
    outerportmanager.merge_ports(self, [self.GetChildList()[0]])
```

set size

setsize allows to make same size all objects of a given type in all treeviews. This is done with the function:

msetTypeSize(type, height, width)

The arguments are:

- **type** - the type of objects to be processed (string).
- **height, width** - the desired size

Usage example

- The following code makes the size 30x30 for all objects of type 'obj'.

```
from arki.graph import setsize

setsize.setTypeSize('obj',30,30)
```

arkimatrixexport

Script for exporting and importing matrix (meta-model) to the XML and from the XML Public functions (the only functions that user of the library should call):

Export

ExportMatrixXml(filename, export_trees=True, export_unchecked_rules=False):

Exports matrix (meta-model) to the XML file with specified name. If **export_trees** flag is **True**, tree information also exported. Otherwise, trees are not exported. The **export_unchecked_rules** flags defines, whether rules, that are not checked in the tree, are exported, when tree is exported. Setting it to **True** can produce significantly bigger xml. Currently, there is no need to use this flag, and it should be considered as deprecated.

Import

ImportMatrixXml(fname, rename_rules=True, delete_rules=True, delete_attribs=True, import_trees=True, preserve_rules=None, preserve_attrs=None, really_import=True)
Works only with arKItect Designer

Imports matrix and optionally trees from the XML file generated by the **ExportMatrixXml**. When matrix is empty, script simply creates new rules and attributes, according to the information in the XML file.

When matrix already contains some rules, this script tries to update it by renaming/adding/removing new rules and by adding/deleting attributes. To determine renamed rules, heuristic algorithm is used. This algorithm takes in account differences between names, child and parent types and attributes and searches for the best guess.

Rules from matrix, that were not found in the XML, and were not detected as the renamed, are removed.

Arguments:

- **fname** - path to the XML file to import
- **rename_rules** - if **True**, algorithm tries to use heuristic algorithm to guess renamed rules. Used for updating matrix.
- **delete_rules** - if **True**, algorithm would delete rules, that were not found in the XML.
- **delete_attribs** - if **True**, algorithm would delete attributes, that were not found in the XML.
- **preserve_rules** - Can be **None** or list of strings. All the rules in this list will not be deleted, even if **delete_rules** is **True**.
- **preserve_attribs** - Can be **None** or list of strings. All attributes in this list will not be deleted, even is **delete_attribs** is **True**.
- **really_import** - if **False**, script performs "dry run", without really modifying the matrix. The library 'arkisubstitute.py' must be available to use this option. If this library is not available, exception would be raised.

Examples

Export

```
from arki.features.impexp import arkimatrixexport
def run( self ):
   arkimatrixexport.ExportMatrixXml(r"c:\data\matrix.xml")
```

Import

```
from arki.features.impexp import arkimatrixexport
def run( self ):
   arkimatrixexport.ImportMatrixXml(r"c:\data\matrix.xml", preserve_rules=["SomeRule"]
) #Import matrix, preserving rule "SomeRule" from deletion, even if it is not present
in the XML
```

arkivariantsexport

Python library for exporting variants' and choices' structures to XML and importing XML files. It requires arkixmltree.py.

Export

Export is done with function:

ExportVariantXml(fileName):

The argument is:

- **fileName** - path and name of the XML to export.

Usage example

The following code exports the choices' and variants' structures of the current project. This code should be executed in arKItect.

```
from arki.features.merger import arkivariantsexport

def main(root):
    arkivariantsexport.ExportVariantXml('C:\\variant.xml')
```

Import

Import is done with function:

ImportVariantXml(fileName):

The argument is:

- **fileName** - path and name of the XML to import.

Usage example

The following code imports the choices' and variants' structures to the current project. This code should be executed in arKItect.

```
from arki.features.merger import arkivariantsexport

def main(root):
    arkivariantsexport.ImportVariantXml('C:\\variant.xml')
```

matrixupdater

Python library for updating matrix from a matrix export. It requires arkimatrixexport.py, arkiexcel.py, arkixmltree.py. Updating the matrix of a project in a new one require six operations:

- Export the matrix of the latest project.
- 1 script to run in the project (to update) in order to show rules', treeviews' and attributes' problems which the user need to solve in order to update the old matrix. I will create a excel file.
- 1 excel file to fill in order to resolve problems.
- 1 script to run in the project (to update) that correct rules', treeviews' and attributes' names and import the new matrix.

ShowProblems

After having exported the latest matrix, you need to run the ShowProblems's script in the project you want to update in order to show problems encountered. This is done with the function:

ShowProblems(updateName, exportFile)

The arguments are:

- **updateName** - name of the update. It can be whatever you want, it's only used to differentiate different updates you might do at the same time. This argument must be same for all functions called for one update.

- **exportFile** - path of the latest matrix export. It must be an xml.

This function will create an excel file named with *updateName* into the cache of arKItect application data's folder and it's opened in a excel window.

Usage example

The following code prepare the export of the current project. This code should be executed in arKItect.

```
from arki.features.metamodeltools import matrixupdater

def run(root):
    updateName = "NewUpdate"
    exportFile = "C:\\Matrix.xml"

    matrixupdater.ShowProblems(updateName,exportFile)
```

The name of the update operation is *NewUpdate* and the path of the matrix export is *C:\Matrix.xml*. The file *NewUpdate.xls* is created into the cache of arKItect application data's folder.

Excel file

It shows all problems to resolve in order to updates. It has 3 sheets: one for the rules, one for the treeviews and one for the attributes. Each sheet has its two first columns filled. The first column is filled with elements of the latest matrix (rules, treeviews or attributes) that have not be found in the second project (the project you want to update) and the second column is filled with elements of the old matrix that have not be found in the good matrix. The user needs to move the cells in these two columns (you can change the row of a cell but a cell in the same column). Two elements in the same row will be supposed to be the same and, in the project you want to update, this element will be renamed (before importing). You can let cells empty. In a row, if you leave one cell empty, no rename will be done. If you leave two rows empty, the method consider that this is the end of the sheet.

CorrectProblemsAndImport

After having modified correctly the excel file, you need to run the CorrectProblemsAndImport's script in the project you want to update in order to resolve problems with this file and import the latest matrix. This is done with the function:

CorrectProblemsAndImport(updateName,exportFile, ImportMatrixXmlKeys)

The arguments are:

- **updateName** - name of the update. It can be whatever you want, it's only used to differentiate different updates you might do at the same time. This argument must be same for all functions called for one update.
- **exportFile** - path of the latest matrix export. It must be an xml.
- **ImportMatrixXmlKeys** - optional arguments of **ImportMatrixXml**.

This function will use the excel file named with **updateName** present into the cache of arKItect application data's folder to rename rules, treeviews and attributes if necessary. Next, it will import the matrix export.

Usage example

The following code prepare the export of the current project. This code should be executed in arKItect.

```
from arki.features.metamodeltools import matrixupdater

def run(root):
    updateName = "NewUpdate"
    exportFile = "C:\\Matrix.xml"

    matrixupdater.CorrectProblemsAndImport(updateName,exportFile)
```

The name of the update operation is *NewUpdate* and the path of the matrix export is *C:\Matrix.xml*. The file *NewUpdate.xls* is read from the cache of arKItect application data's folder. You can add the argument *delete_tree=True* to **CorrectAndImport** to delete useless TreeViews.

projectsmerger

Python library for merging two projects in a new one. It requires arkiexcel.py, arkixmltree.py, arkivariantsexport.py, arkiexport.py (which, now, needs layoutsmerger.py, arkilayout.py, arkilayout_graph.py, arkilayout_tools.py).

Merging two projects in a new one requires six operations:

- 2 scripts to run in each original project.
- 1 excel file to fill in order to resolve names conflicts.
- 1 script to run in the new project.

PrepareExport

First, you need to run the PrepareExport's script in the two original projects. This is done with the function:

PrepareExport(mergeName, defaultTypes, unexportedTreeviews)

The arguments are:

- **mergeName** - name of the merge. It can be whatever you want, it's only used to differentiate different merge you might do at the same time. this argument must be same for all functions called for one merge.
- **defaultTypes** - list of types which names' conflicts will be corrected automatically. All objects of this types are supposed to be different and if there are two objects (of one those types) with the same identifier (type and name), one of them will be renamed automatically before merging the two projects.
- **unexportedTreeviews** - list of treeviews' names which won't be exported and, thus, merge in the new project.

This function will create a new folder named with **mergeName** into the cache of arKItect application data's folder and a sub-folder for each projects to merge. In its sub-directory, the project will be exported. When the second sub-folder is created, *conflict.xls* is created in the merge's folder and opened in a excel window.

Usage example

The following code prepares the export of the current project. This code should be executed in arKItect.

```
from arki.features.merger import projectsmerger

def run(root):
    mergeName = "NewMerge"
    defaultTypes = ["follows"]
    unexportedTreeviews = ["tree0"]


projectsmerger.PrepareExport(mergeName,defaultTypes=defaultTypes,unexportedTreeviews=u
nexportedTreeviews)
```

The name of the merge operation is **NewMerge**. The conflicts between the objects with the type *follows* will be corrected automatically. The treeview named *tree0* won't be exported and, thus, merged in the new project.

conflict.xls

It shows all names' conflicts to resolve before merging. The user needs to fill the third and fourth column with **suffix** to add to objects in conflict in each project. If one of these cells is empty, no suffix will be added to the respective object's name in the respective project. If two objects have the same name (for example: the same suffix, empty or not), they will be merged and they will become one object in the final project. Once filled, save the file and close the excel window. and execute *FinalizeExport*

FinalizeExport

Next, you need to run the FinalizeExport's script in the two original projects. This is done with the function:

FinalizeExport(mergeName, defaultTypes, unexportedTreeviews)

The arguments are:

- **mergeName** - name of the merge. It can be whatever you want, it's only used to differentiate different merge you might do at the same time. this argument must be same for all functions called for one merge.
- **defaultTypes** - list of types which names' conflicts will be corrected automatically. All objects of this types are supposed to be different and if there are two objects (of one those types) with the same identifier (type and name), one of them will be renamed automatically before merging the two projects.
- **unexportedTreeviews** - list of treeviews' names which won't be exported and, thus, merge in the new project.

This function will rename all objects which identifiers are in conflict. The objects with a type in **defaultTypes** will be renamed automatically if necessary and the other one will be renamed with the suffix found in the *conflicts.xls* file. When renaming is finished, the script will export the project, once again, in the good sub-directory.

Usage example

The following code finalizes the export of the current project. This code should be executed in arKItect.

```
from arki.features.merger import projectsmerger

def run(root):
    mergeName = "NewMerge"
    defaultTypes = ["follows"]
    unexportedTreeviews = ["tree0"]


projectsmerger.FinalizeExport(mergeName,defaultTypes=defaultTypes,unexportedTreeviews=
unexportedTreeviews)
```

The name of the merge operation is *NewMerge*. The conflicts between the objects with the type *follows* will be corrected automatically. The treeview named *tree0* won't be exported and, thus, merged in the new project.

MergeProjects

Next, you need to run the MergeProjects's script in the new project. This new project must have the matrix and all the treeviews exported. This is done with the function:

MergeProjects (mergeName, unexportedTreeviews)

The arguments are:

- **mergeName** - name of the merge. It can be whatever you want, it's only used to differentiate different merge you might do at the same time. this argument must be same for all functions called for one merge.
- **unexportedTreeviews** - list of treeviews' names which won't be exported and, thus, merge in the new project.

This function will import the two projects (with names corrected) in the current project (the new one) and clean the merge folder (it will destroy it).

Usage example

The following code finalizes the export of the current project. This code should be executed in arKItect.

```
from arki.features.merger import projectsmerger

def run(root):
    mergeName = "NewMerge"
    unexportedTreeviews = ["tree0"]

    projectsmerger.MergeProjects(mergeName,unexportedTreeviews=unexportedTreeviews)
```

The name of the merge operation is *NewMerge*. The treeview named *tree0* won't be exported and, thus, merged in the new project.

Clean

If the merge fails, you should clean the cache directory. This is done with the function:

Clean(mergeName)

The arguments are:

- **mergeName** - name of the merge. It can be whatever you want, it's only used to differentiate different merge you might do at the same time. this argument must be same for all functions called for one merge.

This function will clean the merge folder (it will destroy it).

Usage example

This code should be executed in arKItect.

```
from arki.features.merger import projectsmerger

def run(root):
    mergeName = "NewMerge"

    projectsmerger.Clean(mergeName)
```

The name of the merge operation is *NewMerge*.

ui

ui.py contains modules graphical user interfaces.

objectBrowser

GetObject

This function allows user to get an object by specifying "Views" and "Types".

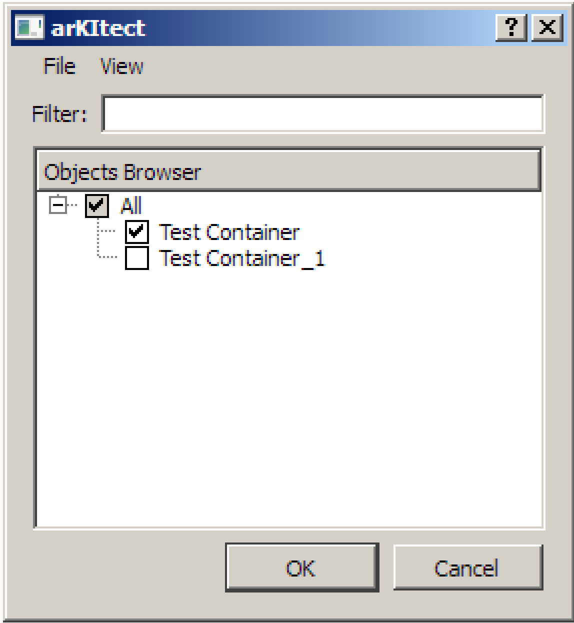
- Following, the GetObject usage:
  - "views" contains the list of views where objects will be researched.
  - "types" contains the list of types to match.
  - "multiSelection" is a boolean ("False" by default) indicating if multi-selection is available.

note: if multiSelection is true it returns a list containing checked objects, else it returns the selected object.

```
import arki.ui.objectBrowser as oB
def run(self):
 listView = ["All"]
    listType = ["Test Container"]
    arkobjref = oB.GetObject(views = None, types = None, selectableType=None, parent =
None, multiSelection = False)
```

Here is an example of a using of this script:

```
from arki.utils.modelgateway.gateway import Gateway
import arki.ui.objectBrowser as oB


def run(self):

 template = r'C:\Users\XXXXXX\Desktop\template\some_document_template.xlsx'

    views = ["All"]
    types = ["Test Container"]

    obj = oB.GetObject(views, types)

    userMacros = { "%Root%": obj.GetName() }
    Gateway().Export(template= template, macroDict=userMacros)
```



The window below is opened, and allows you to select the wanted object and return it to the used function. Filter field allows user to find some objects by name quickly. Filter is applied automatically after user finished typing.

addUnder

This function displays a window containing object that wich can be parent of current object.

Then after selecting the wanted parent, by pressing the "Ok" button, the current object is added to the selected parent.

addUnder takes up to 3 parameters: "addUnder(objRef, views = None, parentTypes = None)"

- Here is the addUnder usage:

```
import arki.ui.objectBrowser as oB

def run(self):
    newparent_obj= oB.addUnder(self)
```

Filedialog

Filedialog is a package containing function to to allow user to select files or folders.

GetOpenFileName(title, filter, initialPath, pathId)

GetOpenFileName is used to open an existing file

```
from arki.ui.filedialog import GetOpenFileName
path = GetOpenFileName("Select xml document", "xml(*.xml)", pathId= "openXML" )
if path:
   ....
```

GetSaveFileName(title, filter, initialPath, pathId)

GetSaveFileName is used to create a new file or replace an existing file

```
from arki.ui.filedialog import GetSaveFileName
path = GetSaveFileName("Select xml document", "xml(*.xml)", pathId= "CreateXML" )
if path:
   ....
```

word_template

Introduction

The use of python as scripting language in arKItect allows to use many public (or not) libraries. To read or edit or generate a Microsoft Office document, one should use *pywin32*.
At Knowledge Inside, we thought it is too much "developer oriented" for people who can use a bit of Python but don't want or don't have time to read and understand its documentation. We propose here an easy method to generate a word document and to fill it with arKItect data.

Tutorial

The selected method has 3 steps:

1. prepare a template document with:
    a. the desired organization and style of the final document
    b. reserved space for additional data. A space is reserved using bookmarks

2. prepare the date in python (it is a list of dictionaries of lists of dictionaries etc.)
3. run **word_template.Generate**

Microsoft Word Bookmarks

Bookmarks are part of any Microsoft Word document. They are used to tag some part of the document.

By default, bookmarks are not visible. To show them, go to the Word options, advanced and check the "Show Bookmarks" check-box. Bookmarks will be shown. This might depend of the version of Microsoft Word you use. (If you use Microsoft Word in French, "Bookmarks" are called "Signet")

To insert a bookmark on a piece of text, select it and choose "Bookmark" in the "Insert" menu. In the pop-up dialog, enter a name for the bookmark and click "Add". Note that bookmark name can only contain Latin letters, numbers and underscores.

If you want to learn more on bookmarks, you should refer to this page http://office.microsoft.com/en-us/word-help/CH006104941.aspx

Hello Word

In the simplest case, document template is a Word document with non-overlapping bookmarks. Each bookmark is a placeholder for data.

1. In Microsoft Word create template document:
   The [and] represent bookmarks limits.

   ```
   Name: [NAME]
   Surname: [SUR]
   ```

2. In Python:

   ```
   import arki.utils.word_template  as wt
   data = [
       {"NAME": "John",
        "SUR": "Smith"
       }
   ]
   wt.Generate("template.doc", data)
   ```

3. Result in Microsoft Word:

   ```
   Name: John
   Surname: Smith
   ```

Repeated sections of template

In addition to simple templates, module word_template supports also templates with repeated sections. Such sections will be automatically repeated by report generator needed number of times.

1. In Microsoft Word create template document:
   The signifies that the [and] refer to the bookmark USERS.

   ```
   Report name: [REP_NAME]
   Generation date: [GEN_DATE]

   Users list follows:
   [USERS
   Name: [NAME]
   Surname: [SURNAME]
   ]USERS
   ```

2. In python:
   We present 2 equivalent possibilities to create the data. The first is more readable but the latter is more "programmatic" and will by use when one programs a document generation with dynamic data.

```
import arki.utils.word_template as wt
data = [{
       "REP_NAME": ' "The report name" ',
       "GEN_DATE": "28.02.2012",
       "USERS": [
          {
              "NAME": "John",
              "SURNAME": "Smith"
          },
          {
              "NAME": "Jean",
              "SURNAME": "Jacques"
          },
          {
              "NAME": "Mayuratan",
              "SURNAME": "Rathakrishnan"
          }
       ] #end of "USERS"
    }]
wt.Generate("template.doc", data)
```

```
import arki.utils.word_template as wt
users = []
users.append({
 "NAME": ' "John" ',
 "SURNAME": ' "Smith" '
})
users.append({
 "NAME": ' "Jean" ',
 "SURNAME": ' "Jacques" '
})
users.append({
 "NAME": ' "Mayuratan" ',
 "SURNAME": ' "Rathakrishnan" '
})
data = []
data.append({
 "REP_NAME": ' "The report name" ',
 "GEN_DATE": "28.02.2012" ,
 "USERS": users
})
wt.Generate("template.doc", data)
```

3. Result in Microsoft Word:

```
Report name: "The report name"
Generation date: 28.02.2012

Users list follows:

Name: John
Surname: Smith


Name: Jean
Surname: Jacques


Name: Mayuratan
Surname: Rathakrishnan
```

Insert a picture

To insert a picture, you must set a function into the dictionary

1. In Microsoft Word create template document:

```
Image: [NAME]
[IMAGE]
```

2. In Python:

```
import pyark
import arki.utils.word_template  as wt

def run(self):
    data = [
        {"NAME": self.GetName(),
         "IMAGE": wt.ViewShotInserter( self  ) }
        # "IMAGE": wt.ViewShotInserter( self, pyark.ARK_GRAPH_PEER_VIEW ) } to retrieve a relation block diagram instead of internal block diagram
]

wt.Generate("template.doc", data)
```

3. Result in Microsoft Word:

Image: System_1

System: System_1

System_7
Check Unicity: Edit...
ID: 4
Init: Edit...

System_8
Check Unicity: Edit...
ID: 5
Init: Edit...

Insert Rich Text

Rich Text attributes are available only with **arKItect** 2.2 and above

Rich text can be retrived from an object's RichText attribute, or by reading a plain .rtf file.

It is inserted by using the **RtfInserter** function, wich takes RTF data as input.

1. In Microsoft Word create template document:

```
Rich-text: [NAME]
[DATA]
```

2. In Python:

```
import pyark
import arki.utils.word_template  as wt

def run(self):
    data = [
        {"NAME": self.GetName(),
         "DATA": wt.RtfInserter( self.GetAttribute( "richtext" ).GetValue()  ) }
]

wt.Generate("template.doc", data)
```

3. Result in Microsoft Word:

```
Rich-text: System_1
Hello Rich World!
```

See an example: the full export of a projection

We've developed an example to demonstrate the capacity of this method.

Add a file Export.py containing the code bellow to the folder *Scripts* near arKItect.exe (you might need to create the folder).

```
import pyark
from arki.features.gendoc import WordReportGeneration as w

object,graphType = pyark.GetActiveView()
w.Generate( object )
pyark.SetActiveView( object, graphType )
```

It added a global menu entry *Tools* Category *Scripting* Panel *Common Tools* *Export*. You can try it!

arki.graph

Collection of packages and libraries to operate on layout of graphical objects in views (graph XML manipulation). Python APIs for this scripts are ArkObjRef.GetGraphXML() and ArkObjRef.SetGraphXML()

arki.graph.transfer_positions

Facilitates copying and pasting graph layouts between arKItect instances.

Entry points

- **copy_positions**() - copies positions of the graph in the active view to the clipboard. Format is XML.
- **copy_recursively**() - copies to the clipboard positions of objects in the current graph and all its sub-graphs, in XML format.
- **paste_positions**() - applies contents of the clipboard to the current view, or to the current view and all its sub-views.

All functions operate Windows clipboard, using PyWin32 module and report errors in popup windows, using PyQT.

User scripts

This library is intended to be used via user scripts, placed into the "Scripts" folder near to the arKitect executable.

Minimal user script must contain 2 lines: import statement and call of the entry point function. Since there are 3 entry points, 3 user scripts are available. They are located in the repository: tools/UserScripts/CopyPasteGraph and attached to this page User scripts

arki.graph.graph_parser

The arki.graph.graph_parser library facilitates manipulation of grphical views, supporting the following features:

- Getting and modifying object object graphical position
- Getting and modifying object expand status
- Getting and modifying flows, their waypoints and label positions
- Getting and modifying oouterbox, goto/from box information

The library provides thefollowing classes:

- **Graph**
- **GraphObject**
- **GraphLink**
- **GraphGotoFrom**
- **GraphOuterBox**

Common usage patterns

Modifying object positions

1. Create **Graph** instance from the **ArkObjRef** (see constructor for parameters)
2. Modifying objects:
    a. If you need to modify existing object, you ask **Graph** to get corresponding **GraphObject** (use **Graph.GetChild**), then modify this object's position using **SetPos**
    b. If you want to add new object, not yet existing in graph (if **GetChild** returned **None**), you can use **Graph.AddChild**
3. After you finished editing, you call Graph.Update() to write modifications back to arKitect.

Alternatively to 2b), you can ask arKitect to create default graph in the Graph constructor, using parameter **regenerate_xml=True**, and use 2a).

Reading object positions

1. Create **Graph** instance from the **ArkObjRef** (see constructor for parameters)
2. Having **ArkObjRef**, get the corresponding **GraphObject** using **Graph.GetChild**, then read this object's position using **GetPos**

```
#parent : ArkObjRef instance
#child: ArkObjRef instance, child of parent

#Get IBD graph
graph = Graph(parent, pyark.ARK_GRAPH_INNER_VIEW)
gchild = graph.GetChild(child)

x,y,width,height = gchild.GetPos()
print "Object position is:", x, y, width, height
```

advanced

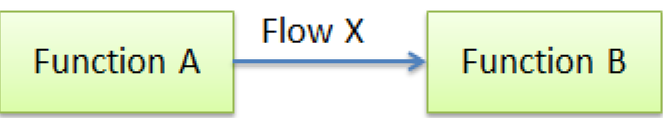
Content

- references

references

Use case

Example of a functional architecture:



Some requirements description will be written in such way:
"The function A should create Flow X with precision"
Problem: if Function or Flow are renamed (or attribute value changed), the requirement will become inconsistent with the functional architecture.
The concept is to allow to make reference to some **object name** inside **MEMO attributes** and **object names**.
Such references should have textual pointer to the object which will allow automatic renaming of values.

Proposed Logic

Inside MEMO attributes:
The function A should create Flow X with precision
The [function A] should create [flow X] with precision
Inside title of an other object :
Child of function A
Child of [function A]

The idea is that names inside '[...]' will be identified as a reference to some other object.
the '[...]' are just an example, in the script configuration you can use any other prefix and suffix as object name delimiters.

Running the script

The script should be run at objects renaming. he will parse all the arKItect project, search for references to the object name, and changes the name inside **objects's names** and **MEMO attributes**.

As you see in the usage below, there are two ways to run the script

- Run it on an event, or integrate it into some other script: you can provide the old name and the new name, script doesn't need the information about the ArkObjRef that is being renamed, he will parse the whole project and make the changes
- Run it using the dedicated renaming GUI: You shouldn't provide the old name nor the new name in this case. You just provide the ArkObjRef that needs to be renamed. The script will propose a small dialog that will ask the user for a new name. It will then make the renaming, and parse the whole project to propagate the renaming into the places it is referenced.

usage:

```
from arki.features.advanced import references
references.RenameRefs(arkobjref = None, old_name = None, new_name = None, prefix =
"[", suffix = "]", projections_list = None)
```

Parameters:

arkobjref: the referenced arKItect object that is being renamed (optional if old_name and new_name provided)

old_name: the previous name of the referenced object (optional if arkobjref provided)

new_name: the new name of the referenced object (optional if arkobjref provided)

prefix/suffix: delimiters of the text referencing. default values are '[]' (these are the values used in the example above)

projections_list: an optimization parameter. Provide arKItect projections list to be used to search for references. This can make the script go faster if you ask it to only search in the required projections. If None, script will search all projections.

An example is to put the following script in object's program attribute. This will create a right-click menu for renaming using dedicated GUI and propagating references according to default parameters:

```
from arki.features.advanced import references
def Run(self):
 references.RenameRefs(self)
```

If you like to put this script on the "onRename" python event in arKItect, you should use a usage similar to the one below. This will automatically launch the references propagation each time the object is renamed.

```
import pyark
from arki.features.advanced import references
def runEvent(self, arg_dict):
    if arg_dict['eventType'] == pyark.EVENT_ONRENAME:
  references.RenameRefs(old_name = arg_dict['oldName'], new_name = self.GetName())
```

Event "onRename" will not be launched if object is renamed using some import script for example.

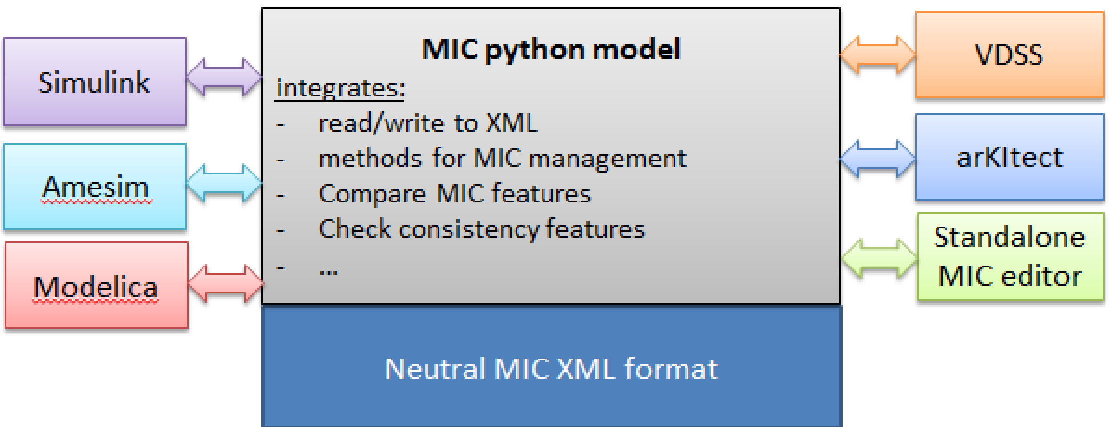
This is a limitation of the current implementation. Such behavior should be managed otherwise.

MIC Model

Content

- MIC generic model
- MIC arKItect interfaces
- MIC Simulink interfaces
- MIC Amesim interfaces
- MIC standalor editor interfaces

MIC python model is library to manage MIC related objects. It consists of a set of methods that helps developers build interfaces and tools around the MIC concept.



Technically, each interface will consist of a transformation between the software format and the MIC python model format.

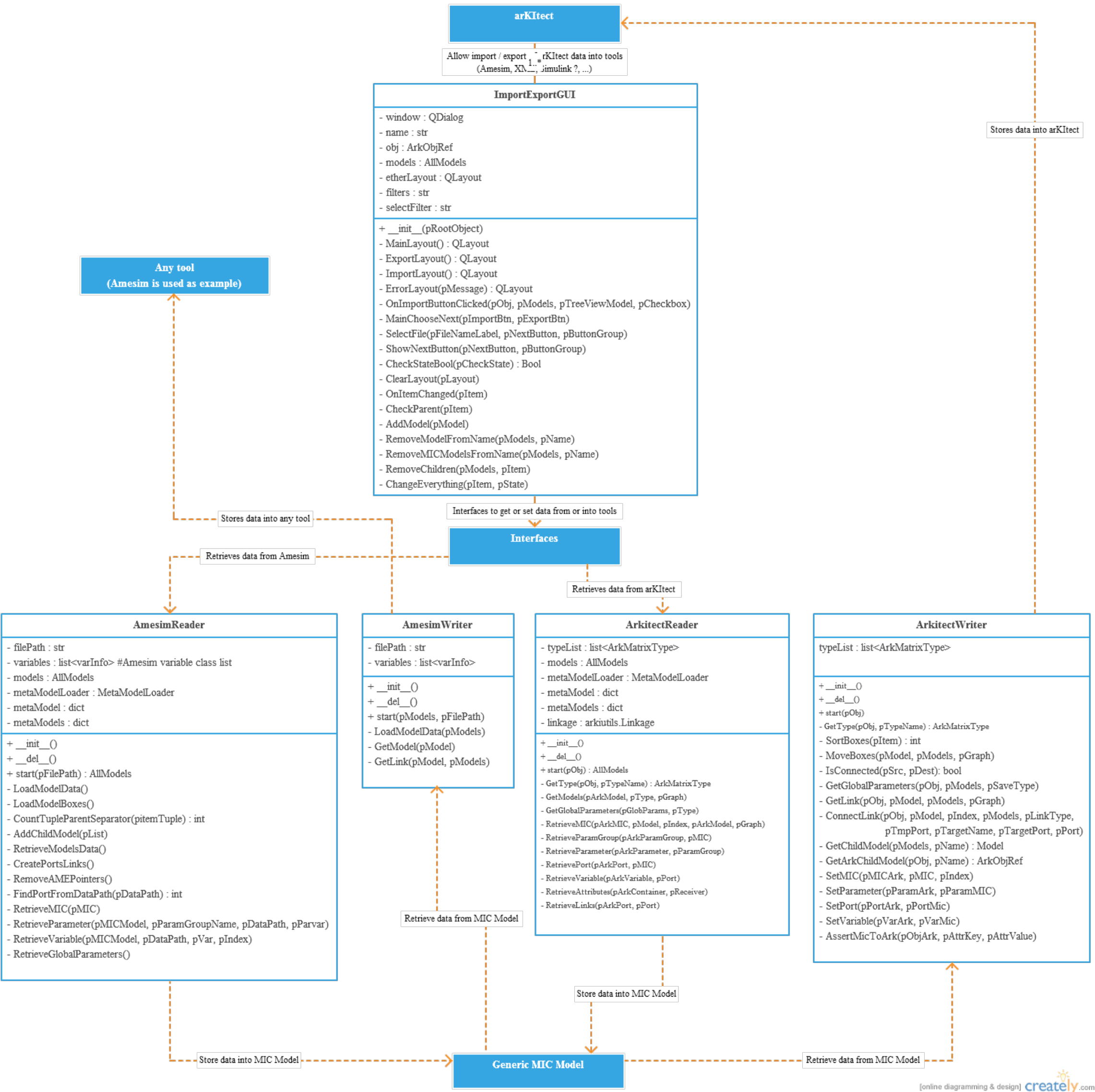
Interfaces behavior

> *Usually, interfaces should be written in 2 modules, one dedicated to "write data" in the tool of your choice and on the other hand another module to "read data" from a tool.*

If you wish you can link your interfaces to the ImportExportGUI object, add the option to use your interfaces in the "MainLayout"

then add the needed layouts for your interfaces and manage the transition between your layouts.

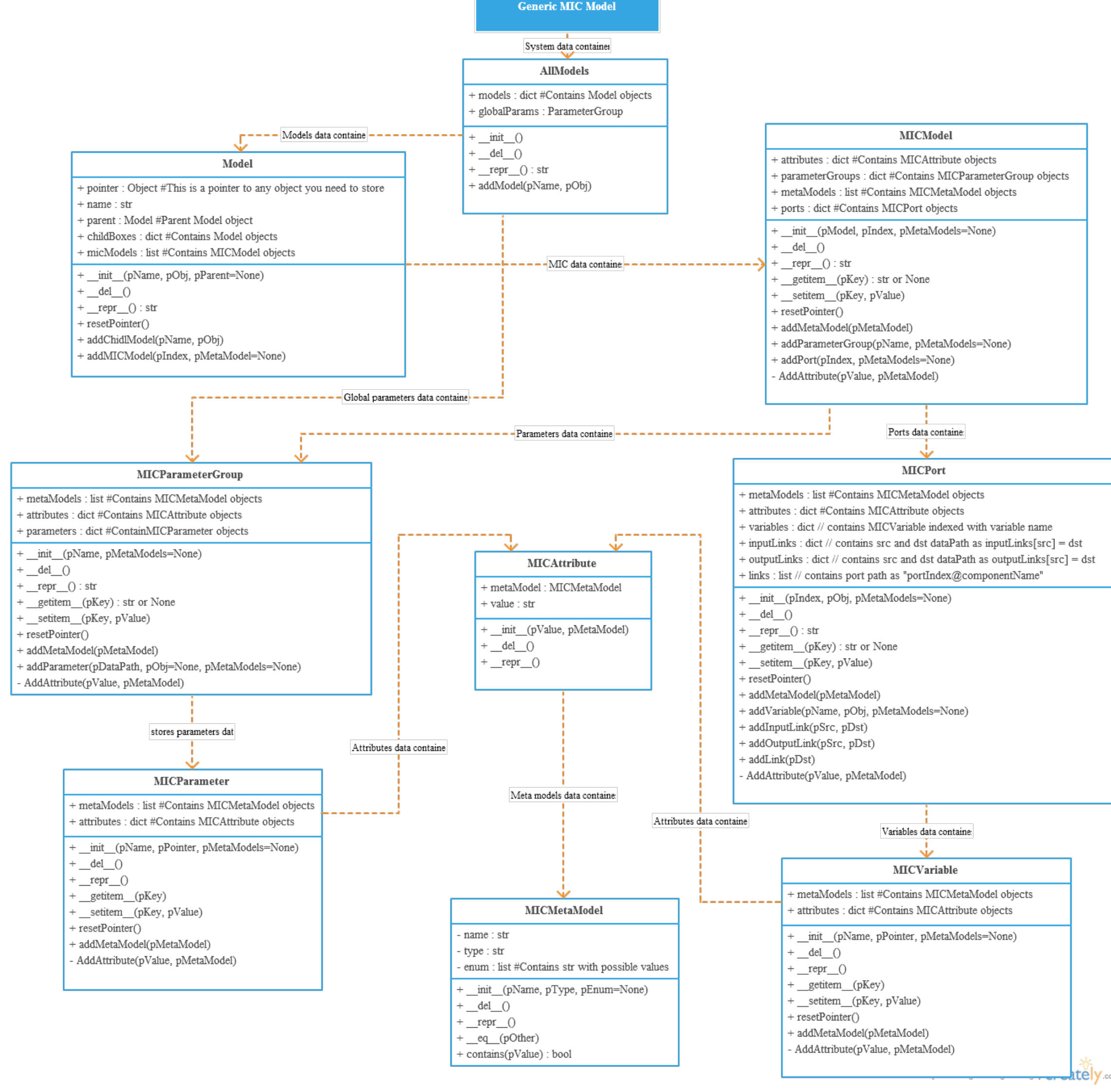
The following picture illustrates the behavior of ImportExportGUI and existing interfaces :



This UML shows the content of each interface and their use of genericModel

MIC generic model

This page describes the python MIC generic model



This UML shows the content of each object in genericModel and their dependancies

MIC arKItect interfaces

MIC Simulink interfaces

MIC Amesim interfaces

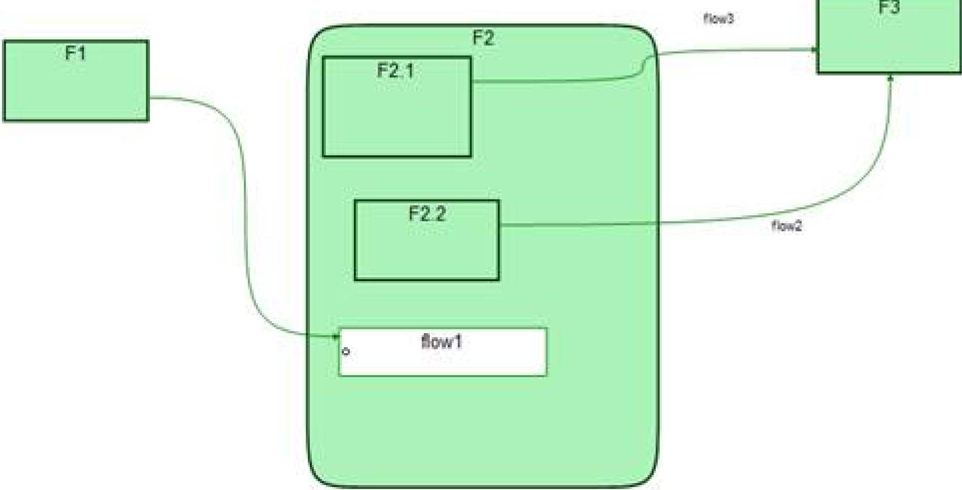
MIC standalor editor interfaces

Generic Chains (GChains) API

Generic Chains is a powerful tool

To manage them with python API, a specific way is need.

Let take a use case:



The goal is to create a Generic Chain where we will show F2 (and its children), F3 and all flows between those objects.

There will have 2 steps :

1. Create objects and allocates objects (as usual)
2. Show objects in the Chain

```
from arkiext.ki.chains.generic_chains import Chain
from arki.utils.arkiutils import GetAllChildren

"""
step 1 : Objects creation
"""
root = pyark.GetRoot("Chain View")

# create a new GChain
arkChain = root.AddChildObject("GChain", "New Chain")

# Functions allocation
arkChain.AddChildReference(f2)
arkChain.AddChildReference(f3)


"""
step 2: Show objects in the Chain
"""
# By default, all objects are hidden, we must show them
# To show objects, we must use arkiext.ki.chains.generic_chains.Chain class
chain = Chain(arkChain)

def addChildren(parent):
    # Show object in Chain
    for child in GetAllChildren(parent):
       chain.add(child) # add method is the way to show an object in Chain

addChildren(f2)
addChildren(f3)

# As we show all objects under F2 and F1 is not under the chain, flow1 will be shown
as "?"

# let's hide all F2 inputs flows
for flow in f2.GetInputFlowList():
    chain.hide(flow) # method hide is the way ti hide an object


# update information
chain.update() # method update is need to validate
```