# Demo and API Documentation - for Rhino

## Table of Contents

# INSTALLATION AND SETUP FOR RHINO

## Run ‘setup’ widget

Run the ‘setup’ widget inside the folder ‘sortal-setup’. This will install packages such as ‘future’, ‘enum’, ‘mpmath’ and the ‘sortal’ library in the right locations, which are necessary for compatibility between IronPython and the Sortal API.

The computer may prompt for whether to allow the widget to make changes; click ‘Yes’ or ‘Allow’. Wait for the packages to finish installing.

Alternatively, if you are unable to run the ‘setup’ widget, you may manually copy-paste the files inside ‘sortal-setup’ in the following locations:

- sortal-packages ≫ sortal
    - Copy-paste to **C:\Program Files (x86)\Rhinoceros 5\Plug-ins\IronPython\Lib** or equivalent on your computer
- site-packages ≫ all files inside
    - Copy-paste to **C:\Program Files (x86)\Rhinoceros 5\Plug-ins\IronPython\Lib\site-packages** or equivalent on your computer

## Set up link to sortal library

➔ Open Rhino
➔ Type in the command box: EditPythonScript
➔ In the Rhino Python Editor window, go to Tools
➔ Select ‘Options’
➔ Select ‘Files’ (default tab)

Add in the following directory into Module Search Paths (change accordingly if your Rhinoceros 5’s IronPython is installed in a different folder):

- C:\ Program Files (x86)\Rhinoceros 5\Plug-ins\IronPython\Lib\site-packages

In the script engine tab, check the Frames Enabled option and click ‘OK’.

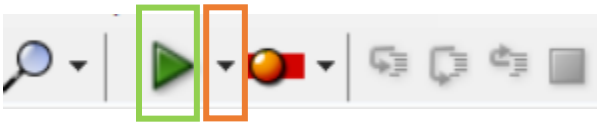
Then close Rhino (close it totally, not just closing the Python editor) and relaunch it again for the changes to take effect.

If you are having trouble with this step, please refer to Annex E: Installation (Windows) for an illustrated explanation.

## Run ‘set_up_all.py’ script

Before running any Sortal API functions, it is necessary to set up the sortal library first. There are two ways to do this:

1. **Run ‘set_up_all.py’ first using the ‘Reset engine and debug’ option** in the Rhino Python Editor, found where the **orange** box is indicating in the picture below. See if you get the response ‘Setup complete’ in your Python Editor window, in which case the sortal library is now ready for use.



   Any scripts run after ‘set_up_all.py’ that use Sortal API functions (all found in ‘sortalgi’, which can be directly imported as it contains all the API functions) are run using the green Play button or the ‘Run Script (no debug)’ option, indicated by the **green** box. This script is further explained in the section ‘set_up_all.py’.

2. **Create your own setup script** by importing the ‘sortalgi’ API into your code:

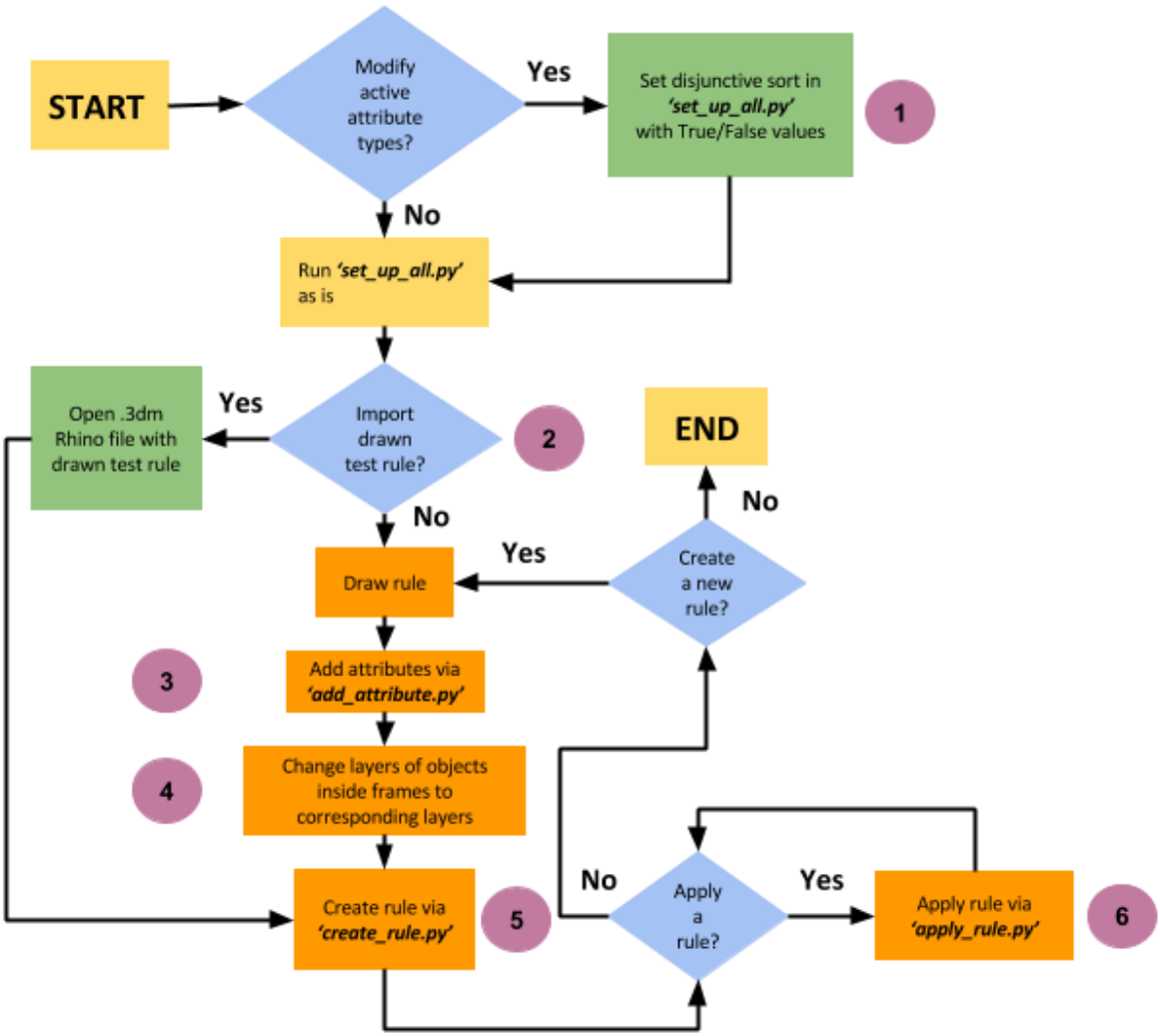
```
import sortalgi as _sgi
_sgi.sortal_setup(
    (   (True, True, True, True, True, True, True),
        (True, True, True, True, True, True, True)))
```

   To ensure that this script works, save it in the same file hierarchy level as sortalgi. To run the script, use the **‘Reset engine and debug’ option** in the Rhino Python Editor, found where the **orange** box indicates in the picture above. How to set this function is further explained in the section ‘sortal_setup’.

# RUNNING DEMO SCRIPTS

The flowchart on the next page describes the general process of the demo scripts in defining a grammar and applying a rule onto a shape. Alongside the demo scripts, a .3dm Rhino file is provided with a sample initial shape and rule definition inside ('drawn_grammar_test_pyramid.3dm').

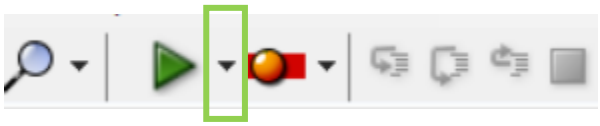
## Which script to run




1. **NEW GRAMMAR:** The disjunctive sort in 'set_up_all.py', that will define all sortal forms in the 'create_rule.py' and 'apply_rule.py' demo scripts, is already set up with all attributes enabled, but this can be modified by changing the True/False inputs in the script. The developer may modify which attributes and geometric sorts are enabled in their own scripts based on the input to the function 'sortal_setup'.

More information about how to set up the disjunctive sort according to shape and attribute requirements can be found in the section 'sortal_setup' and in the script 'set_up_all.py' itself. The basic geometric and attribute sort types used in the class that does the parsing together of the disjunctive sort, which the method 'sortal_setup' uses and is called 'parse_disjunctive_sort', are hardcoded in the script 'sortTypes.py' (found in sortalgi -> sortal_lib_api -> setup).

Once the disjunctive sort's True/False inputs have been set, run 'set_up_all.py' by selecting the downwards pointing arrow next to the green 'Play' button. **Select 'Reset engine and debug'** (**green** box) to empty the sortal library of any previous data (such as rules) and to reset the Python engine. **This is a necessary step to set up the entire sortal library in the Rhino Python engine when first opening Rhino and using the Sortal API.**



Note that to only set up the disjunctive sorts and not draw the empty grammar frames, comment out the 'g.set_up_grammar()' line.

2. **DRAWING IN RHINO or IMPORT DRAWN TEST RULE:** Before running the next demo script 'create_rule.py', either draw the initial shape and rule sides inside the frames, or open a pre-existing draw test rule (.3dm Rhino file).

3. **ADDING ATTRIBUTES:** Run 'add_attribute.py' (by pressing the green 'Play' button, indicated by the **green** box below) to add labels or weights to the drawing. There are also functions that attach pre-existing text dots to lines as the labels of these lines, remove labels from the drawing, add color, descriptions or enumerative values to Rhino objects.

4. Change the figures to their respective layers. Rules should be on a separate layer from initial shapes.

5. **CREATING A RULE:** Run 'create_rule.py' by pressing the green 'Play' button in the Python Editor. The script will prompt for selection of the LHS and RHS frames, rule name and rule description.

    i Enter rule name.
    ii If the rule name already exists in the rule register, there will be a prompt as to whether to overwrite that rule or rename the rule currently being created.
    iii Select the frames corresponding to the LHS shape and RHS shape by clicking on them.
    iv Enter rule description.

6. **APPLYING RULE:** Run 'apply_rule.py'. This script houses the workflow for applying a rule onto a shape: first selecting a subshape to generate matches for, then the shape that houses the subshape, then followed by selecting which rule to apply.

    i Select the subshape by selecting Rhino objects, then press enter. Select the frame where the subshape is housed in by clicking on the frame.
    ii Select the frame that contains the main shape.
    iii Select which rule to apply onto the shape.
    iv The code will then generate potential rule applications. Select which application to apply.
    v The rule is applied onto the shape. The previous 'initial' shape is deleted in Rhino, and the new one is drawn inside the frame of the main shape.

Each demo script is discussed in further detail in the following sections.

## set_up_all.py

'set_up_all' initializes the disjunctive sort and other basic geometric sorts to be used in rule creation and application in the sort register and/or creates new empty grammar frames. **This script should be run first before running any other scripts that involve the Sortal API.** It has the following steps:

1. Setting up empty grammar frames:

    ```
    g.set_up_grammar()
    ```

    This line is used to set up the initial shape and rule frames. It can be commented out before running the script if the grammar is already drawn in Rhino.

2. Setting up the disjunctive sort and other basic geometric and attribute sorts necessary for sortal library operations. The Sortal API 'sortalgi' is imported here as 'sgi'.

    ```
    sgi.sortal_setup(((True, True, True, True, True, True), (True, True, True, True, True, True),(False, False, False, False, False, False), (False, False, False, False, False, False)))
    ```

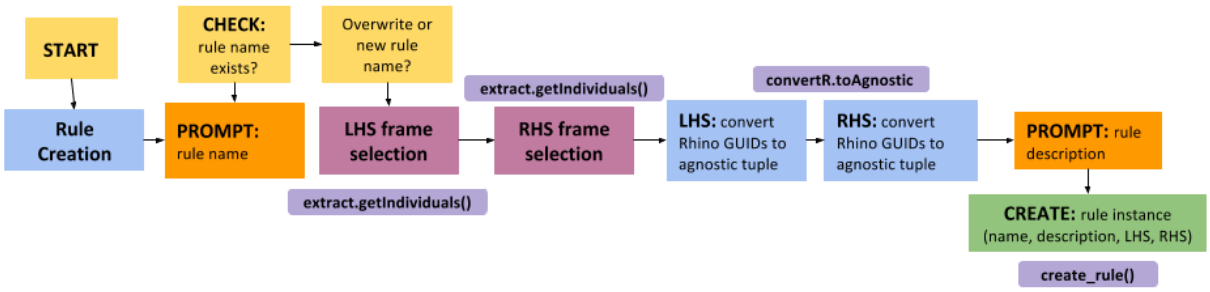
    This step is crucial for any use of the sortal library and its API. It initializes the sort types used to create the data structures (forms) that will be passes through the sortal library, and is needed to run any API functions such as those used in the next few scripts described.

    The crucial function in this step is 'sgi.sortal_setup()', which takes as input a tuple with two sub-tuples as elements. The tuple acts as an indicator of which geometric types (line segments,

points) and attributes to enable in the succeeding use of the Sortal API. More information on how to set the tuple can be found in the section on 'sortal_setup'.

## create_rule.py

The script 'create_rule.py' contains methods for creating a rule instance. It prompts for the frames containing the shapes of the LHS and RHS, the rule name and the rule description. The major steps of the code are described by the flowchart below.




The specific function that creates the rule is 'create_rule', found in sortalgi -> sortal_lib_api -> create_rule. The function 'create_rule()' takes as input the arguments <rule name>, <rule description>, <LHS> and <RHS> (in agnostic tuple form), and returns the rule instance and the rule name.

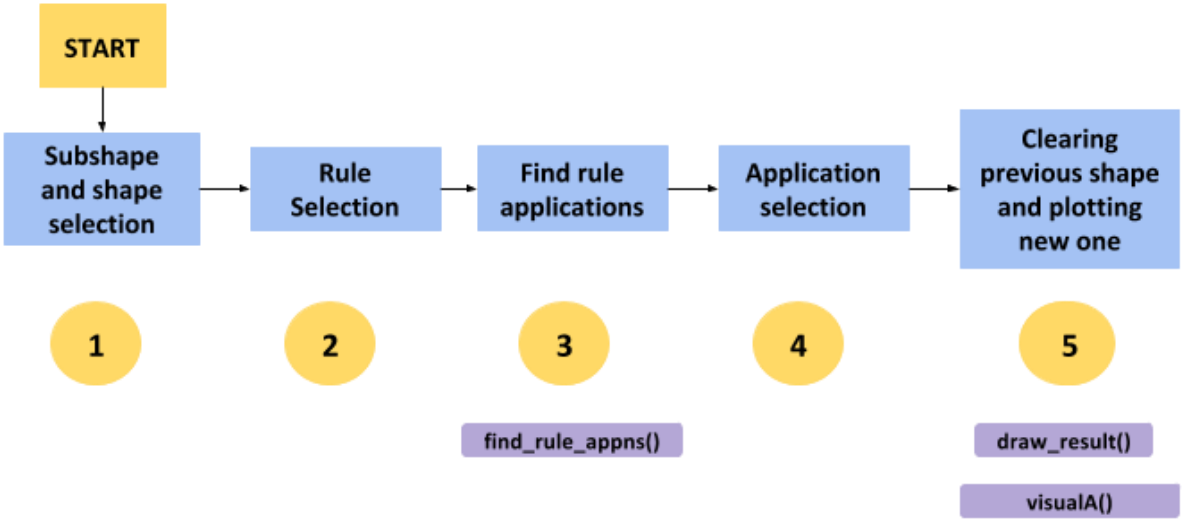
```
ruleNew, ruleName = sgi.create_rule(ruleName, ruleDesc, lhsAgnostic,
rhsAgnostic)
```

More information about the agnostic tuple form can be found in Agnostic Form Structure. 'create_rule' is further explained in the 'API' section of this document.

## apply_rule.py

'apply_rule.py' contains methods for selecting a rule and applying it onto a shape. The script prompts for selection of a subshape to narrow down match and result generation, then for the main shape. After selecting these inputs, it prompts for the rule to apply onto the main shape, generates potential rule applications, prompts for which application to use, then clears the main shape and plots the new shape after rule application in the same frame.

The major steps of the code are described by the flowchart on the next page.




The next flowchart describes the workflow in more detail, with the **blue boxes on the left and the numbered circles** indicating the major sections of the code. The section on the right describes the breakdown of each step.

1. Subshape and Shape Selection

   The subshape is selected by using 'rs.GetObjects()', which prompts for selection of a set of Rhino objects. This is followed by selection of the frame block instance that the subshape is in, which then generates the block instance insertion point. The list of Rhinos GUIDs, frame instance GUID and its reference point are fed into 'convertR.toAgnostic', which converts the subshape into an agnostic tuple and returns the relevant Rhino GUIDs (based on the geometric base sorts that were included in the disjunctive sort initialized in 'set_up_all.py').

```
subshape = rs.GetObjects('Select subshape as basis for matches')
frame_instance = rs.GetObject('Select frame of subshape')
refPointSub = rs.BlockInstanceInsertPoint(frame_instance)
subshapeAgnostic, subshapeIds = convertR.toAgnostic(subshape, refPointSub)
```

   A similar process is used to select the main shape, with the method 'extract.getIndividuals()' taking the place of 'rs.GetObjects()'. More information about 'extract.getIndividuals()' can be found in Annex B: Conversion Classes.

```
shape, refPointMain, layerNameMain = extract.getIndividuals('Select the
frame containing the main form')
shapeAgnostic, shapeIds = convertR.toAgnostic(shape, refPointMain)
```

2. Rule selection

   Enter which rule to use in rule application. The list of rules is generated from those already stored in the rule register inside the sortal library.

```
# Select rule to apply onto forms
choice = raw_input('Select rule by entering number of rule')

# Check if choice entered is a digit
if choice.isdigit():
    choice = int(choice)
    while True:
        if choice > 0 and choice <= len(rule.register):
            chosenRule = ruleRegisterList[choice - 1]
            chosenRule = sgi.getRule(chosenRule)
            break
            …
```

3. Finding rule applications

   The function 'find_rule_appns' takes as arguments the shape and subshape (each in agnostic form), and the rule instance or rule name of the chosen rule. It checks if the subshape is part of the shape first. If no, then the function is exited. If yes, then it generates a list of transformation matrix-LHS-RHS-rule application combinations, using the subshape to narrow down the possibilities. The rule instance of the chosen rule can be created using the 'create_rule' (see further explanation on create_rule) function or retrieved using the function 'getRule'.

```
chosenRule = sgi.getRule['rule_name']
ruleAppns = sgi.find_rule_appns(chosenRule, shapeAgnostic,
subshapeAgnostic)
```

   It returns a list of LHS-RHS-rule application tuples in agnostic form. The function 'find_rule_appns' is further explained in the API section of this document.

4. Application selection

   The list 'ruleAppns' is used to show the applications generated by 'find_rule_appns'. In 'apply_rule.py', these are printed.

```
print ('\nAvailable rule applications', len(ruleAppns))
for i in ruleAppns:
    print (str(ruleAppns.index(i)+1) + ' . ', i)

choice = raw_input('Select rule application by entering number')

while True:
    if choice.isdigit() and len(ruleAppns) != 0:
        choice = int(choice)
           if choice > 0 and choice <= len(ruleAppns):
               …
```

5. Clearing previous shape and plotting shape after rule application
   After selecting which application to apply onto the shape, 'draw_result()' clears the previous shape and plots the shape after rule application. This function is found in demo scripts -> rhino_specific -> draw_result. For more information on 'draw_result', refer to Annex A: rhino_specific.

```
shapeIdsNew = draw_result(shapeIds, choice, ruleAppns, layerNameMain,
refPointMain)
```

# AGNOSTIC FORM STRUCTURE

## General Structure

Inputs and outputs of the API functions in sortalgi are all in agnostic form and do not rely on the sortal library. An agnostic object is a dictionary composed of 2 (or potentially more) elements, each corresponding to a geometry type. The index key for each element is the name of its geometry type in plural form. Currently, points, line segments and polylines are supported.

Any agnostic object has the following format:

**{ 'points': [<p1>, <p2>, …] , 'line segments': [<ls1>, <ls2>, …] }**

where the content of each element is a list of individuals or an empty list. Geometries are stored according to their type under the corresponding index key. In cases where there are only line segments and no points, or vice versa, the dictionary will have only one element, e.g.:

**{ 'line segments': [<ls1>, <ls2>, …] }**

Each geometric individual is a tuple composed of the coordinate tuple and the attribute dictionary. Coordinates like the start and end points of line segments and the coordinates of a point are stored inside tuples since they are meant to be immutable.

For the format of an individual geometric object:

1. Points: **(<x, y, z>, {dictionary of attributes, if any})**
2. Line segments **((<x1, y1, z1>, <x2, y2, z2>), {dictionary of attributes, if any})**

Where <x, y, z> is a coordinate tuple and {attributes if any} is a dictionary of attributes. The index keys of the attribute dictionary correspond to the attribute type, i.e.

**att_dict = { 'labels': ['a'], 'descriptions': […, …], 'weight': 0.3, 'color': [203, 201, 100] }**

All text attributes are stored in lists (labels, descriptions, enumeratives). This allows for several of these text attributes to be linked to a geometric individual.

## Coordinate Structure

The structure of the geometry types' coordinates vary per geometry type, but all are tuples in (x, y, z) format. Here are examples for points and line segments:

1. Points – (x, y, z) - **(24.0, 27.0, 15.0)**
2. Line segments – ( (x1, y1, z1), (x2, y2, z2) ) - **( ( (11.839816578, 7.31118696075, 0.0), (11.839816578, 7.31118696075, 24.0) )**

## Attribute Structure

If the individual has any attributes, then it is stored in a dictionary, which is always the second element of the individual geometric object's tuple. Each element of the dictionary has a corresponding index key, which relates to the attribute type, as follows:

```
{ 'labels': ['a'], 'descriptions': […, …], 'weight': 0.3, 'color': [203, 201, 100] }
```

Sample attribute dictionary:

```
att_dict = {'label': ['a'], 'weight255': [100,100,100], 'color': [255,255,0])
```

All text attributes are stored as lists, since there can be more than one label/description/enumerative per geometric individual. Weight is stored as a float number, while color and weight255 are stored as lists of rgb values

If an individual geometry has no attributes, it has an empty attribute dictionary as its second element.

## Individual Structure

An example of an individual tuple object inside the agnostic tuple would then be:

```
(((2.3160365998, 20.1679989656, 0.0), (11.839816578, 7.31118696075, 24.0)), {'color': [100.0,100.0,100.0], 'labels' ['label text'], 'weight': 1.5})
```

This describes a line segment with color, label and weight attributes. If an individual object has no attributes, the attribute tuple is left blank:

```
(((17.096841323439989, 19.006387598569567, 0.0), (18.262391789932455, 22.8328078721171, 0.0)), {})
```

## Sample Agnostic Object

Putting all these together, an example of a complete agnostic object (composed of points and line segments) can look like this:

```
{'line segments': [(((17.409052693004977, 18.36419106456523, 0.0), (17.409052693004977, 18.36419106456523, 11.999999999999986)), {'weight': 0.01}), (((17.409052693004963, 22.364191064565222, 0.0), (17.409052693004963, 22.364191064565222, 5.999999999999993)), {'weight': 0.01}), (((21.40905269300498, 18.364191064565226, 0.0), (21.40905269300498, 18.364191064565226, 5.999999999999993)), {'weight': 0.01}), (((17.409052693004977, 18.36419106456523, 5.999999999999993), (17.409052693004963, 22.364191064565222, 0.0)), {'weight': 0.01}), (((17.409052693004977, 18.36419106456523, 11.999999999999986), (17.409052693004963, 26.3641910645652, 0.0)), {'weight': 0.01}), (((21.40905269300498, 18.364191064565226, 5.999999999999993),
```

(21.409052693004959, 22.36419106456523, 0.0)), {'weight': 0.01}),
(((17.409052693004977, 18.36419106456523, 0.0), (17.409052693004963,
26.3641910645652, 0.0)), {'weight': 0.01}), (((17.409052693004977,
18.36419106456523, 5.999999999999993), (17.409052693004963,
22.364191064565222, 5.999999999999993)), {'weight': 0.01}),
(((21.40905269300498, 18.364191064565226, 0.0), (21.409052693004959,
22.36419106456523, 0.0)), {'weight': 0.01}), (((17.409052693004977,
18.36419106456523, 5.999999999999993), (21.40905269300498,
18.364191064565226, 0.0)), {'weight': 0.01}), (((17.409052693004963,
22.364191064565222, 5.999999999999993), (21.409052693004959,
22.36419106456523, 0.0)), {'weight': 0.01}), (((17.409052693004977,
18.36419106456523, 11.999999999999986), (25.409052693004973,
18.36419106456524, 0.0)), {'weight': 0.01}), (((17.409052693004963,
22.364191064565222, 0.0), (21.40905269300498, 18.364191064565226, 0.0)),
{'weight': 0.01}), (((17.409052693004963, 22.364191064565222,
5.999999999999993), (21.40905269300498, 18.364191064565226,
5.999999999999993)), {'weight': 0.01}), (((17.409052693004963,
26.3641910645652, 0.0), (25.409052693004973, 18.36419106456524, 0.0)),
{'weight': 0.01}), (((17.409052693004977, 18.36419106456523, 0.0),
(25.409052693004973, 18.36419106456524, 0.0)), {'weight': 0.01}),
(((17.409052693004977, 18.36419106456523, 5.999999999999993),
(21.40905269300498, 18.364191064565226, 5.999999999999993)), {'weight':
0.01}), (((17.409052693004963, 22.364191064565222, 0.0), (21.409052693004959,
22.36419106456523, 0.0)), {'weight': 0.01})], 'points':
[((18.409052693004973, 19.364191064565219, 0.99999999999999878), {'labels':
['a']}), ((18.409052693004973, 19.364191064565219, 6.999999999999991),
{'labels': ['a']}), ((18.40905269300497, 23.364191064565205,
0.99999999999999878), {'labels': ['a']}), ((19.40905269300498,
20.364191064565219, 1.9999999999999976), {'labels': ['a']}),
((22.409052693004973, 19.364191064565226, 0.99999999999999878), {'labels':
['a']})]}

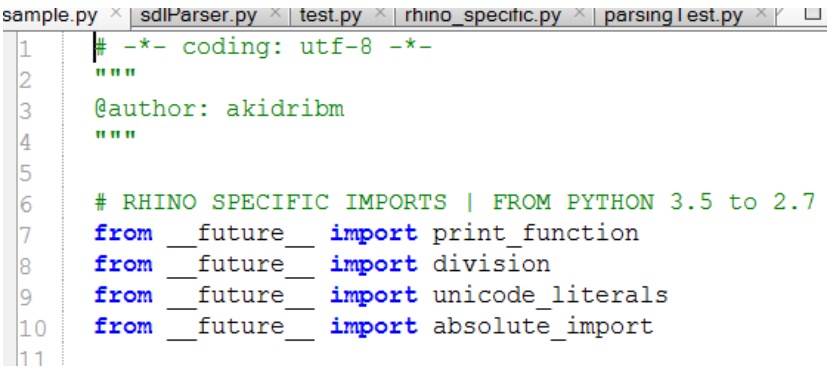
# API

## Importing future for Python 3.5 to Python 2.7 compatibility

When writing scripts using the Sortal API and Library in Rhino or Python 2.7, it is necessary to put the following lines of code at the start of every script to import the 'future' package. They enable compatibility between Python 3.5 (which is used to write the Sortal API and Library) and RhinoPython, which uses IronPython (equivalent to Python 2.7).

```
from __future__ import print_function
from __future__ import division
from __future__ import unicode_literals
from __future__ import absolute_import
```

For example:




## API functions

This section will focus on the functions from the 'sortalgi' package:

- sortal_setup – sets up the Sortal Library; it is necessary to run this function first before doing anything else with the Sortal API
- create_rule – creates rules from rule name, rule description, agnostic tuple form of LHS and RHS
- find_rule_appns – generates rule applications from inputs rule object or rule name, subshape (agnostic) and main shape (agnostic)
- maximalize – maximalizes an agnostic object
- partOf – determines if a subshape input is part of the main shape input
- getRuleName – returns rule name
- getRuleDesc – returns rule description
- getLHS – returns LHS agnostic object
- getRHS – returns RHS agnostic object
- getRule – returns corresponding rule object (if it exists) based on rule name input

- setRuleName – changes name of rule instance to inputted rule name
- setRuleDesc – changes description of rule instance to inputted description
- overwriteRule – overwrites rule instance's LHS and RHS, while maintaining the original rule name and description of the rule object

The use of 'sgi' in this section refers to the package 'sortalgi', which contains all the methods listed above and is imported in the code snippet examples as follows:

```
import sortalgi as sgi
```

## *sortal_setup*

When writing scripts with the Sortal API, it is necessary to use the function 'sortal_setup' once at the start, before using any other functions from the Sortal API. 'set_up_all.py' does this for the demo scripts, and the documentation inside it also shows how to set the values of the function depending on which geometry/attributes are to be enabled. As this is a function within 'sortalgi', it can also be used in other custom scripts.

### *Syntax*

```
sgi.sortal_setup((point_and_attributes_tup, line_segment_and_attributes_tup))
```

### *Parameters*

point_and_attributes_tup: Required. Tuple of True/False values that specifies if points and which attributes are enabled

line_segment_and_attributes_tup: Required. Tuple of True/False values that specifies if line segments and which attributes are enabled

These two tuples are combined into one tuple input, in the same order as in 'Syntax'. How to set the values of each tuple is further explained in 'Example'.

### *Returns*

True. If successful

False. If unsuccessful

### *Example*

```
sgi.sortal_setup((True, True, True, True, True, True, True), (True, True,
True, True, True, True, True)))
```

This line enables points and line segments, both with label, description, weight, weight255, enumerative, and color.

The input is a tuple with two sub-tuples. Each tuple of True/False values corresponds to a geometric individual:

1. 1st Tuple – Points
2. 2nd Tuple – Line Segments

To switch on specific attributes for a geometric base (e.g. point, line segment, plane, volume), the order of value setting activates certain attributes, with the numbers indicating the corresponding index of the attribute in the tuple:

0. Existence of geometry

1. Label

2. Description

3. Weight – this weight attribute has a range of decimal values from 0.0 to 1.0

4. Weight255 – this weight attribute has a range of integer values from 0 to 255

5. Enumerative - values and sorting order need to be specified by developer; these are hard coded in the file 'sortTypes.py'.

6. Color – RGB value

Therefore, for a point with the attributes label, description, weight255 and enumerative enabled, the tuple is written as:

```
(True, True, False, True, True, False, False)
```

This format applies to line segments as well. Alternatively, if attributes are set as consecutive False values they can be left out entirely of the tuple, for example:

```
sgi.sortal_setup(((True,), (True, True)))
```

where the function enables points with no attributes and line segments with labels, but no other attributes. **Do note that for any tuple with only a single value, the single value must be followed by a comma and enclosed by parentheses to be recognized as a tuple input by the function.** However, if the False values are not consecutive, i.e. if they are followed by a True value, then they must all be listed until the last True value.

For example:

```
sgi.sortal_setup((True, True), (True, False, True, False, True))
```

where the function enables points with labels, and line segments with descriptions and weight255. Lastly, to set the sortal library to not enable a specific geometry at all (i.e. line segments or points only):

```
sgi.sortal_setup(((False,), (True, True)))
```

where the function sets up the library to only allow for line segments with labels, but no points. As said previously, when setting a single value tuple, the single value must be followed by a comma and enclosed by parentheses to be recognized as tuple inputs.

## *create_rule*

This function is used to create a rule using agnostic shapes.

*Syntax*

```
new_rule_object, new_rule_name = sgi.create_rule(rule_name, rule_description,
lhsAgnostic, rhsAgnostic)
```

*Parameters*

rule_name: Required. Rule name (string).

rule_description: Required. Rule name description (string).

lhsAgnostic: Required. Agnostic object representing rule left hand side.

rhsAgnostic: Required. Agnostic object representing rule right hand side.

*Returns*

new_rule_object: Newly created sortal rule object.

new_rule_name: Name of newly created sortal rule object.

None. If unsuccessful.

*Example*

```
ruleNew, ruleName = sgi.create_rule(ruleName, ruleDesc, lhsAgnostic,
rhsAgnostic)
```

The function also stores the newly created rule instance in the sortal library. A rule object can be retrieved later by the function 'getRule' from the 'sortalgi' package.

How this function works is further explained in Annex F: More Information on SortalGI Functions.

## *find_rule_appns*

This method generates the transformation matrices, matches, results and rule applications for a given subshape (optional), main shape (that the subshape should be part of) and rule.

*Syntax*

```
ruleAppns = sgi.find_rule_appns(rule, shapeAgnostic, subshapeAgnostic)
```

*Parameters*

rule: Required. Rule name (string input) or rule instance. If the rule name does not exist in the rule register or the input is not a rule object/name, then a Value Error will be raised.

shapeAgnostic: Required. The shape under rule application.

subshapeAgnostic: Optional. The subshape used to determine rule matches, that is, transformations under which the left shape is a part of this subshape. This subshape is optional; if provided, it must be a part of the given shape; if not provided, the shape will instead be used to determine rule matches.

*Returns*

ruleAppns: [rule_appn, …] Returns a list of potential rule applications (tuples), where:

rule_appn: (t, t_of_a, t_of_b, c_prime). Where:
t: matrix. The matrix of the transformation under which the left shape is a part of the given subshape (or shape, if no subshape is provided).
t_of_a: shape_agnostic. The match found in shapeAgnostic
t_of_b: shape_agnostic. The transformed right shape based on t_of_a
c_prime. shape_agnostic. The resulting shape, i.e., the result of applying the rule under the transformation t to the given shape.

None. If unsuccessful

*Example*

```
ruleAppns1 = sgi.find_rule_appns(ruleName, shapeAgnostic, subshapeAgnostic)

ruleAppns2 = sgi.find_rule_appns(ruleObject, shapeAgnostic)
```

where the first input is either the rule name (string, ruleName) or rule object (ruleObject), shapeAgnostic is the agnostic tuple form of the initial main shape, and subshapeAgnostic is the agnostic tuple of the subshape. How this function works is further explained in Annex F: More Information on SortalGI Functions.

## *maximalize*

This function takes as input an agnostic object and returns its maximalized form.

*Syntax*

```
sgi.maximalize(shapeAgnostic)
```

*Parameters*

shapeAgnostic: Required. The shape to be maximalized, in agnostic form.

*Returns*

shapeAgnosticMax. The maximalized shape in agnostic form, If successful.

None. If unsuccessful.

*Example*

```
shapeAgnosticMax = sgi.maximalize(shapeAgnostic)
```

How this function works is further explained in Annex F: More Information on SortalGI Functions.

## *partOf*

This function checks if a subshape (agnostic object) is part of another shape (agnostic object). It returns the Boolean value True (subshape is part of shape) / False (subshape is not part of shape).

*Syntax*

```
sgi.partOf(subshapeAgnostic, shapeAgnostic)
```

*Parameters*

subshapeAgnostic: Required. The subshape in agnostic form, that is to be checked against another shape.

shapeAgnostic: Required. The shape in agnostic form.

*Returns*

True. If successful

False. If unsuccessful

*Example*

```
result = sgi.partOf(subshapeAgnostic, shapeAgnostic)
```

where result is a Boolean value. How this function works is further explained in Annex F: More Information on SortalGI Functions.

## *getRuleName, getRuleDesc, getLHS, getRHS*

These functions take as input a rule object/rule name and return the rule object/description/LHS (agnostic form)/RHS (agnostic form) of the rule object. Only the rule object is accepted as input for 'getRuleName'.

*Syntax*

```
sgi.getRuleName(ruleObj)
sgi.getRuleDesc(ruleName or ruleObj)
sgi.getLHS(ruleName or ruleObj)
sgi.getRHS(ruleName or ruleObj)
```

*Parameters*

Either input is accepted for these functions.

ruleName: Full name of rule to be retrieved.

ruleObj: Rule object where name/description/LHS/RHS will be retrieved from. 'getRuleName' only accepts ruleObj.

*Returns*

sgi.getRuleName: Returns rule name (string).

sgi.getRuleDesc: Returns rule description (string).

sgi.getLHS: Returns rule LHS (agnostic object).

sgi.getRHS: Returns rule RHS (agnostic object).

None. If unsuccessful.

*Example*

```
ruleName = 'rule_1'
ruleObj = sgi.getRule(ruleName)
print ('Name:', sgi.getRuleName(ruleObj))
print ('Description:', sgi.getRuleDesc(ruleObj))
```

```
print ('LHS:', sgi.getLHS(ruleObj))
print ('RHS:', sgi.getRHS(ruleObj))
```

where ruleObj is the rule instance.

The information about the rule is printed out in each line; the functions return the outputs as either strings (for 'getRuleName', 'getRuleDesc') or as agnostic dictionaries ('getLHS', 'getRHS').

## *getRule*

This function retrieves the rule name's corresponding rule object from the sortal library, if a rule object with that name exists.

*Syntax*

```
sgi.getRule(ruleName)
```

*Parameters*

ruleName: Required. The full name of the rule to be retrieved.

*Returns*

Rule object. If successful, it returns the rule object with the same rule name.

None. If unsuccessful.

*Example*

```
rule_object = sgi.getRule('rule_1')
```

## *setRuleName*

This function changes the name of a rule object. It returns the altered rule object with its name changed.

*Syntax*

```
sgi.setRuleName(ruleObj, newRuleName)
```

*Parameters*

ruleObj: Required. Rule object to be altered.

newRuleName: Required. String input of new rule name.

*Returns*

Rule object. If successful, it returns the rule object after its name has been changed.

None. If unsuccessful

*Example*

```
sgi.setRuleName(ruleObj, newRuleName)
```

## *setRuleDesc*

*Description and Use*

This function changes the description of a rule instance. It returns the altered rule object with its description changed.

*Syntax*

```
sgi.setRuleDesc(ruleObj, newRuleDesc)
```

*Parameters*

ruleObj: Required. Rule object whose description will be changed.

newRuleDesc: Required. String input of new description.

*Returns*

Rule object. If successful, the rule object containing the new rule description is returned.

None. If unsuccessful

*Example*

```
newRuleDesc = 'new rule desc'
sgi.setRuleDesc(ruleObj, newRuleDesc)
```

## *overwriteRule*

*Description and Use*

This function changes the sides of a rule object.

*Syntax*

```
sgi.overwriteRule(ruleObj, lhsAgnosticNew, rhsAgnostic)
```

*Parameters*

ruleObj: Required. Rule object whose sides are to be changed.

lhsAgnosticNew: Required. Agnostic object of new LHS, or can be old LHS.

rhsAgnosticNew: Required. Agnostic object of new RHS, or can be old RHS.

Returns

Rule object. If successful, the new rule object with changed sides but the same name and description is returned.

None. If unsuccessful.

*Example*

```
newRule = sgi.overwriteRule(ruleObj, lhsAgnosticNew, rhsAgnostic)
```

where ruleObj is the original rule instance, lhsAgnosticNew is an agnostic object which is different from that of the current LHS of ruleObj, and rhsAgnostic is an agnostic object that could potentially also be different from that of the current RHS of ruleObj.

To illustrate further, as in 'use_sortal_api_functions.py':

```
# Obtains new LHS agnostic object
# This will replace the original rule's LHS
lhs, refPointL, layerNameL = extract.getIndividuals('Select the frame
containing the new LHS')
lhsAgnostic, lhsObjects = convertR.toAgnostic(lhs, refPointL)

# Shows original rule sides
print ('Name:', sgi.getRuleName(ruleObj))
print ('Description:', sgi.getRuleDesc(ruleObj))
print ('Original LHS:', sgi.getLHS(ruleObj))
print ('Original RHS:', sgi.getRHS(ruleObj))

rhsAgnostic = sgi.getRHS(ruleObj)

# Changes LHS of ruleObj to new LHS agnostic object
newRule = sgi.overwriteRule(ruleObj, lhsAgnostic, rhsAgnostic)

# Shows altered rule sides
print ('\nAltered rule')
print ('Name:', sgi.getRuleName(newRule))
print ('Description:', sgi.getRuleDesc(newRule))
print ('New LHS:', sgi.getLHS(newRule))
print ('RHS:', sgi.getRHS(newRule))
```

In this example, a frame containing the edited LHS is selected and turned into an agnostic object. The original RHS of the rule instance is obtained. They are used together as inputs for the function ‘overwriteRule’, which returns a rule object with the same name and description as before, but with an altered LHS.

# ANNEXES

## Annex A: rhino_specific

'rhino_specific' houses functions that handle plotting geometry and attributes in the Rhino viewport. The folder houses the following files:

- draw_result
- visualA
- rhino_specific_attributes

More information about each file can be found in Annex A: rhino specific.

## Annex B: Conversion Classes

The conversion codes are stored in the 'convert' folder.

- convertR
- convertS
- convertA
- extract

These classes handle extraction and conversion between Rhino GUID, agnostic and sortal formats. More information about each convert class and the extract class can be found Annex B: Conversion Classes.

## Annex C: Relevant Terminology

Annex C: Relevant Terminology lists relevant terminology in the Sortal Grammar Interpreter library.

## Annex D: Description of Imports

Annex D Description of Imports describes the functions of sortal imports and Rhino-specific imports. These are mostly seen in the files 'sortTypes.py', 'parse_disjunctive_sort.py', 'convertA', and 'convertS'.

## Annex E: Setting up IronPython (Windows)

To install the sortal library in Rhino (Windows), follow the steps in Annex E: Installation (Windows).

## Annex F: More Information on SortalGI

Annex F: More Information on SortalGI explains, step by step, how some of the functions in SortalGI work.