

# PySI API

## Introduction

The PySI API facilitates its users to build new SI-Plugins from scratch or extend and modify existing ones. An SI-Plugin is an implementation of an *effect* of an *interactive region* (early concept: [1]). Such an *effect* is triggered once end-users overlap two *interactive regions*, defining a *collision*. Additionally, *interactive regions* can be *linked*. Such *linking relationships* are defined according to *attributes*. Application developers specify in the implementation of an *effect* which *attributes* of that *effect* can be linked to which *attributes* of other *effects* and which *linking action* shall occur. Standard *attributes* are either provided by the PySI API, such as *position*, or may be created as ad-hoc identifiers which are chosen by application developers.

The PySI API aims to streamline the implementation of an SI-Plugin in partially descriptive, partially imperative and self-documenting fashion. This document's goal is to describe the PySI API beyond its code documentation. Therefore, the next sections describe PySI, its API and use case examples.

If you are just starting out with developing SI-Plugins it may be useful to play around with the PySI API first to get a feel for it. For that, skip ahead to the sections Use Cases Examples and Example: Implementation of a Tag-Effect. The more detailed aspects of PySI and its API are present in the other chapters and may be consulted as required. This document applies the best practices proposed by RFC2119 (keywords **must**, **should**, **may**) [2]

## Description

### API Integration in SIGRun

PySI exposes bindings to C++ datastructures and functions. These bindings are built with the use of Boost.Python [3] Each plugin file is stored in an *object* within SIGRun. This *object* functionally represents the pythonic *self* in SIGRun. Therefore, the python code implemented in the functions of SI-Plugins is executed by SIGRun once end-users trigger their events and actions. In this way, API users build application logic purely in python (with the use of python libraries available to them), while SIGRun provides the *glue* to present those applications to end-users.

### The Startup File

#### Goal

API users configure the startup sequence of SIGRun

- API users **must** provide a startup file in the *plugins* folder
- API users **may** spawn *interactive regions*
- API users **may** configure logging behaviour

## Naming Convention

Startup python file **must** be named *StartSIGRun.py*

## Structure

```
from libPySI import PySI
# further imports as you see fit such as plugin files for using their static members or aux

# other variables or functions

def on_start():
    # configuration logic **must** be written here
    pass

# other variables or functions
```

Summary:

- *StartSIGRun.py* **must** import libPySI
- *StartSIGRun.py* **must** include a module-level function called *on\_start()*
- *on\_start* **must** contain all configuration logic
- *on\_start()* **must not** have any arguments
- API Users **should** create additional functions and variables on module level

## The Plugin File

### Goal

- API Users implement *effects* for *interactive regions*
- API Users **must** specify collision capabilities (str-value) and callback functions for *collision event* emissions and receptions
  - *on\_enter* - triggered exactly once when first time collision of *interactive regions* is detected
  - *on\_continuous* - triggered in each subsequent SIGRun update when collision remains ongoing
  - *on\_leave* - triggered exactly once when a previously ongoing collision is detected to have stopped
- API Users **must** specify linking relationships according to attributes (str-values) and linking actions (functions)

## Naming Convention

- plugin files **must** contain exactly one class
- names of plugin files **must** be the exact same name as the classes they implement
- example: Tag.py contains a class called Tag

## Structure

MyEffect.py

```
from libPySI import PySI
from plugins.standard_environment_library.SIEffect import SIEffect

# same name as python file
# it is highly recommended to inherit from SIEffect for ease of use
class MyEffect(SIEffect):
    regiontype = PySI.EffectType.SI_CUSTOM
    regionname = PySI.EffectName.SI_STD_NAME_CUSTOM

    def __init__(self, shape=PySI.PointVector(), uuid="", kwargs={}):
        super(MyEffect, self).__init__(shape, uuid, "res/my_effect.png", Button.regiontype,

        self.qml_path = "path/to/MyEffect.qml"

        # further member variables

        # further member functions
```

Summary:

- plugin files **must** import libPySI
- plugin files **must** use static member variables *regiontype* and *regionname*
- *regiontype* and *regionname* **must** be initialized according to PySI effect types and effect names respectively (if unsure use above example initialization)
- *init*'s signature **must** look exactly as the example above
- *self.qml\_path* **must** be left empty if no QML file is required to accompany the plugin
- plugin files **should** import SIEffect for ease of use
- *init*'s default parameters **may** be changed

## Enabling and Disabling of Effects and Linking Relationships

```
from libPySI import PySI
from plugins.standard_environment_library.SIEffect import SIEffect

class MyEffect(SIEffect):
    regiontype = PySI.EffectType.SI_CUSTOM
    regionname = PySI.EffectName.SI_STD_NAME_CUSTOM

    def __init__(self, shape=PySI.PointVector(), uuid="", kwargs={}):
        super(MyEffect, self).__init__(shape, uuid, "res/my_effect.png", Button.regiontype,

        self.qml_path = "path/to/MyEffect.qml"
```

```

# enable an effect, according to a capability (str-value), to be emitted according
# collision_event_on_enter_emit must be a function (e.g. self.on_collision_event_enter_emit)
# collision_event_on_continuous_emit must be a function (e.g. self.on_collision_event_continuous_emit)
# collision_event_on_leave_emit must be a function (e.g. self.on_collision_event_leave_emit)
self.enable_effect("your_capability", SIEffect.EMISSION, <collision_event_on_enter_emit>)

# enable an effect, according to a capability (str-value), to be received according
# "your_other_capability" can be any str-value
# collision_event_on_enter_recv must be a function (e.g. self.on_collision_event_enter_recv)
# collision_event_on_continuous_recv must be a function (e.g. self.on_collision_event_continuous_recv)
# collision_event_on_leave_recv must be a function (e.g. self.on_collision_event_leave_recv)
self.enable_effect("your_other_capability", SIEffect.RECEPTION, <collision_event_on_enter_recv>)

# disable an effect, according to a collision capability (str-value), so the effect stops emitting
# "your_capability" can be any str-value
self.disable_effect("your_capability", SIEffect.EMISSION)

# disable an effect, according to a collision capability (str-value), so the effect stops receiving
# "your_other_capability" can be any str-value
self.disable_effect("your_other_capability", SIEffect.RECEPTION)

# enable a link, according to an attribute (str-value), so that the effect emits the link
# "your_attribute" may be any str-value
# <linking_action_for_emission> must be a function (e.g. self.linking_action_for_emission) as parameter
self.enable_link_emission("your_attribute", <linking_action_for_emission>)

# enable a link, according to a source attribute (str-value) and target attribute, so that the effect emits the link
# "your_attribute" can be any str-value
# <linking_action_for_emission> must be a function (e.g. self.linking_action_for_emission) as parameter
self.enable_link_reception("source_attribute", "your_attribute", <linking_action_for_emission>)

# disable a link, according to an attribute (str-value), so that the effect stops emitting the link
self.disable_link_emission("your_attribute")

# disable a link, according to a source_attribute and target attribute, so that the effect stops receiving the link
# specify "your_attribute" to selectively disable that exact linking relationship
# leave "your_attribute" empty ("") to disable all linking relationships which are registered
self.disable_link_reception("source_attribute", "your_attribute")

```

Summary:

- API users **must** specify own names for capabilities and attributes if those are not part of PySI API
- API users **should** use the functions `enable_effect`, `disable_effect`, `enable_link_emission`, `disable_link_emission`, `enable_link_reception`, `disable_link_reception` to register required *collision events* and *linking actions*
- API users **should** use PySI built-in capabilities and attributes if available

e.g. PySI.CollisionCapability.DELETION (for collision with deletion region)  
or PySI.LinkingCapability.POSITION (for linking to position attribute)

## The QML File

### Goal

- API Users **must** use qml-files to define which styling an *effect* applies to its associated *interactive regions* beyond flat coloring
- QML-files and plugin-files **should** come in pairs
- Plugin-files `*+may*` have no associated qml-file
- QML[4] facilitates styling and modifying the style of regions at runtime for API Users

### Naming Convention

- QML-files **must** have exactly the same name as their associated plugin-files except for their file endings
- example: MyEffect.py and MyEffect.qml

### Structure

Item

```
{  
    function updateData(data)  
    {  
        // apply data  
        // e.g.  
        // image.width = data.width;  
    }  
  
    id: container  
    visible: true  
  
    // further QML components such as Item, Image, etc.  
    // .  
    // .  
    // .  
}
```

Summary:

- API Users **must** conform to QML-Syntax
- API Users **must** provide a function `updateDate(data)` which takes exactly one argument
- API Users **must** use the `updateData(data)`-function to update qml-file and styling at runtime

- API Users **should** define one *Item* in the qml-file which contains all other qml-components
- API Users **may** change the parameter *data* to another identifier of their choosing

## Data Application

Data originates from python calls in a SI-Plugin

```
self.add_QML_data("width", 200, PySI.DataType.INT)
self.add_QML_data("height", 300, PySI.DataType.INT)
self.add_QML_data("visible", True, PySI.DataType.BOOL)
```

and then data is applied in QML for registering the styling changes

```
Item
{
    function updateData(data)
    {
        // update the width, height and visibility of the container with data received from
        // the key/value pairs of data-parameter are equal to the ones specified in add_QML_
        container.width = data.width; // container.width now has the value 200
        container.height = data.height; // container.height now has the value 300
        container.visible = data.visible; // container.visible now has the value true
    }

    id: container
    visible: false

    // further QML components such as Item, Image, etc.
    // .
    // .
    // .
}
```

Summary:

- *data*-parameter is of type JavaScript-Object
- API Users **must** use the *updateData(data)*-function to update qml-file and styling at runtime
- API Users **must** use common JavaScript for the *updateData(data)*-function
- API Users **must** call the function *self.add\_QML\_data(key, value, datatype)* from within the python plugin to change data in the qml-file
  - *key*-parameter directly translates to the key identifier in the JavaScript object *data* (example: *self.add\_QML\_data("width", 100, PySI.DataType.INT)* will result in *data.width = 100*)
  - *value*-parameter depicts the value assigned to the perviously specified key of the *data*-object

- *datatype*-parameter is required by SIGRun to transfer the data changes to QML
- API Users **must** assign values of the *data*-object to qml-components pre-defined in the qml-file
- API Users **may** use this procedure to update any qml-component which is supported by QML such as Image, Item, etc.

## Use Of Non-Plugin Python Files

### Goal

API Users **may** perform *code decomposition* by defining python files and classes which SIGRun **must not** treat as SI-Plugins

### Naming Convention

- Classes in Non-Plugin Python Files **must** be named with two leading underscores (e.g. class `__MyClass`)
- Non-Plugin Python Files **may** contain classes
- API Users **may** name Non-Plugin Python Files as they chose

### Structure

Non-Plugin Python File

```
# module level functions
# SIGRun scans for classes and ignores module level functions per default
def do_job():
    pass

# static class for helper functions (e.g. for better definition of namespaces)
# __ as name prefix prevents SIGRun to read it as a SI-Plugin
class __MyHelperFunctions:
    @staticmethod
    def do_work():
        pass

#function
# .
# .
# .

# class defining behaviour or serving as a complex datastructure for use in plugins
# __ as name prefix prevents SIGRun to read it as a SI-Plugin
class __MyHelperDatastructure:
    def __init__(self, args, kwargs):
        # assignment
```

```

pass

# functions
# .
# .
# .

```

Summary:

- Non-Plugin Python Files **must not** contain classes which names do not start with two leading underscores
- API Users **should** use Non-Plugin Python files for decomposition purposes
- Non-Plugin Python Files **may** have module level functions
- Non-Plugin Python Files **may** have classes

## PySI Internal Bindings to C++-Datastructures

The following sections describes the special datastructures of PySI which **must** be used in SI-Plugins. These datastructures originate from SIGRun and are exposed via C++-bindings which are generated with Boost.Python. The PySI API streamlines their usage.

### Point2

```
p = PySI.Point2(5, 5)
```

Summary:

- custom datastructure to represent 2D points
- API users **must** provide two float values (x, y) to the constructor
- Point2 is a custom *initialization only* exposure of the datatype *glm::vec2* [5]
- API Users **may not** require to use this datastructure directly

### Point3

```
p = PySI.Point3(5, 5, 1)
```

Summary:

- custom datastructure to represent 3D points
- API users **must** provide three float values (x, y, z) to the constructor
- API users **should** initialize the z-value to 1
- Point3 is a custom *initialization only* exposure of the datatype *glm::vec3* [5]
- API Users **may not** require to use this datastructure directly

### Color

```
color = PySI.Color(255, 0, 255, 255)
```



Summary:

- custom datastructure to represent RGBA colors
- API users **must** provide four float values (r, g, b, a) to the constructor
- Color is a custom *initialization only* exposure of the datatype *glm::vec4* [5]

### LinkRelation

```
lr = PySI.LinkRelation(source_uuid, source_attrib, target_uuid, target_attrib)
```

Summary:

- custom datastructure to represent linking relationships
- API Users **must** provide four str values (source\_effect\_uuid, source\_effect\_attribute, target\_effect\_uuid, target\_effect\_attribute) to the constructor
- API Users **may not** require to use this datastructure directly

### PointVector

```
# construction
pts = PySI.PointVector() # empty
pts = PySI.PointVector([[1, 1, 1], [2, 2, 1], ..., [m, n, 1]]) # list of points

# appending
pts.append(PySI.Point3(x, y, z)) # appending Point3 directly
pts.append([x, y, z]) # appending Point3 indirectly with python list of exactly three
```

Summary:

- custom datastructure to represent a list of Point3 datastructures
- PointVector is usable in a pythonic way equally to python lists
- API Users **must** provide a list of lists of three floats to the constructor or leave it empty
- API Users **must** provide three floats inside the inner list when using python lists for initialization or appending
- API Users **may not** require to use this datastructure directly

### LinkRelationVector

```
# construction
lrv = PySI.LinkRelationVector() # empty
lrv = PySI.LinkRelationVector([source_uuid, source_attribute, target_uuid, target_attribute], ...)

# appending
pts.append(PySI.LinkRelation(source_uuid, source_attrib, target_uuid, target_attrib)) #
pts.append([source_uuid, source_attrib, target_uuid, target_attrib]) # appending LinkR
```

Summary:

- custom datastructure to represent a list of link relationships
- LinkRelationVector is usable in a pythonic way equally to python lists
- API Users **must** provide a list of lists of four str values to the constructor or leave it empty
- API Users **must** provide four str-values inside the inner list when using python lists for initialization or appending
- API Users **may not** require to use this datastructure directly

## StringVector

```
# construction
sv = PySI.StringVector() # empty
sv = PySI.StringVector(["str", "str2", ..., "strn"]) # list of link relationships

# appending
sv.append("str") # appending str value directly
```

Summary:

- custom datastructure to represent a list of strings
- StringVector is usable in a pythonic way equally to python lists
- StringVector is a custom exposure of **std::vector**
- API Users **must** provide a list of str values to the constructor or leave it empty
- API Users **must** provide a str-value when appending to a StringVector
- API Users **may not** require to use this datastructure directly

## PartialContour

```
# construction is not performed manually

# appending of new cursor id
self.__partial_regions__[cursor_uuid] = PySI.PointVector() # adding PointVector value

# appending of point according to cursor id
self.__partial_regions__[cursor_uuid].append(PySI.Point3(x, y, 1)) # adding PointVector
self.__partial_regions__[cursor_uuid].append([x, y, 1]) # adding PointVector value directly
```

- Custom datastructure to represent a contour/shape which is currently drawn by an end-user
- PartialContour is usable in a pythonic way equally to python dicts
- PartialContour is custom exposure of a **std::map<std::string, std::vector>** datastructure
- PartialContour uses uuids of cursor plugins as key (str value) for values of type PointVector
- API users **must** provide string/PointVector key/value pair to add data
- API users **must** provide a valid uuid of a valid cursor as key

- API Users **must not** call the constructor directly as this is handled by PySI
- API Users **should not** require to use this datastructure directly as it is only required when building a *canvas* plugin
- API Users **should not** build a new *canvas* plugin

## String2FunctionMap

*# e.g. linking relationships eligible for emission are stored in a String2FunctionMap p*

```
def link_emission():
    return "emit"

# construction
link_emit = PySI.String2FunctionMap()
link_emit = PySI.String2FunctionMap({"my_link_attribute_to_emit", link_emission})

# assignment
link_emit["my_link_attribute_to_emit"] = link_emission
```

- Custom datastructure to represent key/value pairs of str-values and python functions
- String2FunctionMap is usable in a pythonic way equally to python dicts
- API Users **must** provide a python dict with str-value keys and functions as values to the constructor or leave it empty
- API Users **may not** require to use this datastructure directly

## String2String2FunctionMapMap

In linking action source effect:

```
# emission functions used in example below
def emit_data(self):
    return x, y, z, self._uuid

self.enable_link_emission("my_source_attribute", self.emit_data)
```

In linking action target effect

```
# e.g. linking relationships eligible for reception are stored in a String2String2FunctionMap
self.enable_link_reception("my_source_attribute", "my_target ", self.receive_data)

# number of args is dependant on the number of returned values (tuple) of the emission
def receive_data(self, x, y, z, source_uuid):
    # use data of linking action
    pass

# construction
```

```

link_rcv = PySI.String2String2FunctionMapMap()
link_rcv = PySI.String2String2FunctionMapMap({"my_source_link_attribute": {"my_target_

# assignment
link_rcv["my_source_link_attribute"]["my_target_link_attribute"] = self.receive_data

```

- Custom datastructure to represent key/value pairs of str-values and String2FunctionMap
- String2String2FunctionMapMap is usable in a pythonic way equally to python dicts containing values of dicts
- API Users **must** provide a python dict with str-value keys and dicts as values, which represent String2FunctionMap, to the constructor or leave it empty
- API Users **may not** require to use this datastructure directly

## Use Cases Examples

### Collision

### Linking

### Example: Implementation of a Tag-Effect

## Pitfalls and Caveats

## References

- [1] Wimmer, R., & Hahn, J. (2018). A Concept for Sketchable Workspaces and Workflows. <https://epub.uni-regensburg.de/36818/1/A%20Concept%20for%20Sketchable%20Workspaces%20and%20Workflows.pdf> {#ref1}
- [2] RFC2119. <https://tools.ietf.org/html/rfc2119> {#ref2}
- [3] Boost.Python. [https://www.boost.org/doc/libs/1\\_73\\_0/libs/python/doc/html/index.html](https://www.boost.org/doc/libs/1_73_0/libs/python/doc/html/index.html) {#ref3}
- [4] QML. <https://doc.qt.io/qt-5/qtqml-index.html> {#ref4}
- [5] GLM. <https://glm.g-truc.net/0.9.9/index.html> {#ref5}