Python Bridge

in RTMaps

PythonBridge v3.0.0 (January 2020)

Table of content

1 Quick introduction to python 4
2 Why Python in RTMaps? 4
3 Python Bridge in RTMaps 4
3.1 A few technical details 4
3.2 Logs 4
3.3 Upgrading from python_v2 5
4 Python Setup 5
4.1 Prerequisites 5
4.2 Windows 5
4.2.1 Installing PYTHON 5
4.2.2 Choose your python installation 6
4.2.3 How do I install python libraries you do not provide? 7
4.3 Linux 7
4.4 Troubleshooting 8
4.4.1 Numpy import problem 8
4.4.2 Library import problem 8
5 A quick look at the python component 8
5.1 Basic configuration 11
5.2 Advanced configuration: 11
5.3 User Properties 12
6 Inputs, outputs and properties 12
6.1 Definition 12
6.2 Inputs 13
6.2.1 Available types as inputs 13
6.2.2 Accessing inputs 14
6.2.3 Reading policies 15
6.2.4 I/O type mapping 15
6.2.5 Additional variables 17

6.3 Outputs 17
6.3.1 Available types for output 17
6.3.2 How to write on outputs 18
6.3.3 I/O type mapping 18
6.4 Properties 19
7 IOElt 20
8 Using RTMaps Functions 21
9 Debugging 22
10 Advanced usage 23
11 Libraries 23
11.1 Matplotlib 23
11.2 PIL 24
11.3 RPy2 24
11.4 Tensorflow 24
11.5 Pyqt 24
11.6 Lupa 25

1 QUICK INTRODUCTION TO PYTHON

Python is a multi-paradigm programming language, most renowned for its simplicity, readability and accessibility compared to other languages like C++ or Java. Python is a powerful interpreted language, has a lot of scientific libraries available and has many features, most notably the possibility to write and edit code during its execution. No doubt Python is one of the most popular languages nowadays.

Using Python, writing an RTMaps component has never been so easy! More about Python here: https://www.python.org/about/

2 WHY PYTHON IN RTMAPS?

Compiled languages like C++ are faster and more optimized for real-time jobs than Python. However, there are several reasons to use Python as well:

- **Python is easy to use:** Python can be easily used with little programming knowledge. Its syntax is very similar to Matlab® so many users find its syntax intuitive.
- **Python is very powerful!** You can use python to implement complex and object-oriented codes in a very few lines.
- **Python has many libraries:** most of them in the scientific domains, but not only. *numpy* for manipulating arrays and matrix, *matplotlib* to easily plot your data, *tensorflow* for deep learning, etc.
- **Change your code on the flow:** Change the code during the execution and see immediately the results! **No compilation required,** because Python is an interpreted language so there is no extra step to perform, just write your code and test it!

3 PYTHON BRIDGE IN RTMAPS

3.1 A FEW TECHNICAL DETAILS

Every *PythonBridge* component on the diagram will spawn a dedicated process, hidden from the user. In this process Python will be embedded and will communicate with RTMaps through shared memory. This separation between Python process and RTMaps process is done to ensure Python runs in a natural environment as it is meant to be.

For example, when you play with some libraries like matplotlib, those libraries expect to run their code in the main thread which is not possible by running in the same process as RTMaps.

You can set specific environment variables as well, this is very useful if you want to use LD_PRELOAD under Linux or override PYTHONPATH in some specific shell like PreScan.

3.2 LOGS

Python process relies on g3log (https://github.com/KjellKod/g3log) for logging. So should you encounter any crash or problem, please read this log first. If you have problems understanding it, please send it to Intempora (support@intempora.com) and explain your problem. This will help us understanding your issue.

3.3 UPGRADING FROM PYTHON_V2

Upgrading from old version *python_v2* has to be done manually, the existing code has to be slightly updated to work on the *PythonBridge*. Indeed, you have to :

- Create a *Dynamic()* function. This function was not used in python_v2 as every inputs, outputs and properties allocation were made in the *__init__(self)* function. This is not the case anymore.

- Move all your inputs, outputs and properties creations in this *Dynamic()* function. Typically, you will have to move your *self.add_input*, *self.add_ouput* and *self.add_property* calls if existing.

4 PYTHON SETUP

4.1 PREREQUISITES

First of all, *PythonBridge* does not provide any Python3 installation. Indeed, you have to set up a working python installation by yourself. Depending on the Operating System you use, the steps are different. Paragraph 4.2 deals with Windows while 4.3 covers the Linux OS. Here are a few common points on those two OS:

- You must install python that is **compatible with RTMaps version**, which means the **same bitness** (64 bit python for a 64bit RTMaps). RTMaps supports 3.6 and 3.7 under Windows and only the python already installed on the distribution under linux.

- **Numpy** must be installed on the python version you want to use. Numpy version must be superior or equal to 1.13.

- If you have multiple installations of python on your machine (and you may have some that you don't know about) we recommend to **set the python install path specifically**.

- Last but not least, please read the rest of this section carefully, it contains addition information (qt.conf, etc…)

4.2 WINDOWS

On Windows, *PythonBridge* is compatible with Python 3.6 and superior. You have to provide your own Python installation that *PythonBridge* will use. If you have no Python installation on your machine we recommend to install latest Python3 64bit (see next paragraph).

4.2.1 INSTALLING PYTHON

We advise you to install the latest **Python3 64bit** version: https://www.python.org/downloads/. Please choose the *Windows x86-64 executable installer*. Also please remember not to install Python in a folder that contains spaces in its name.

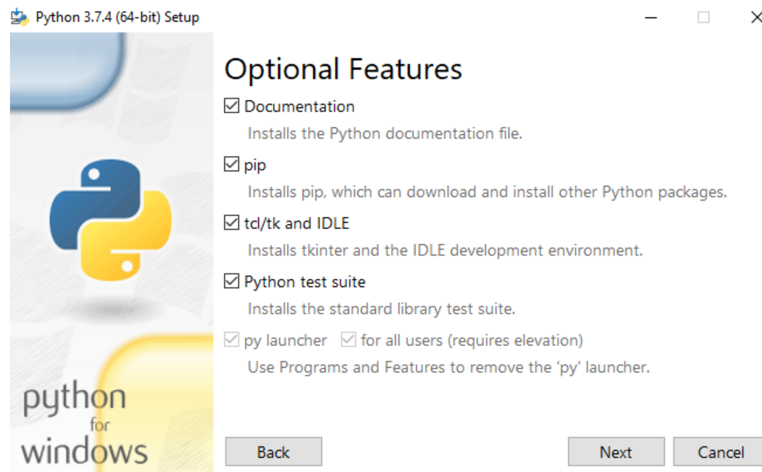
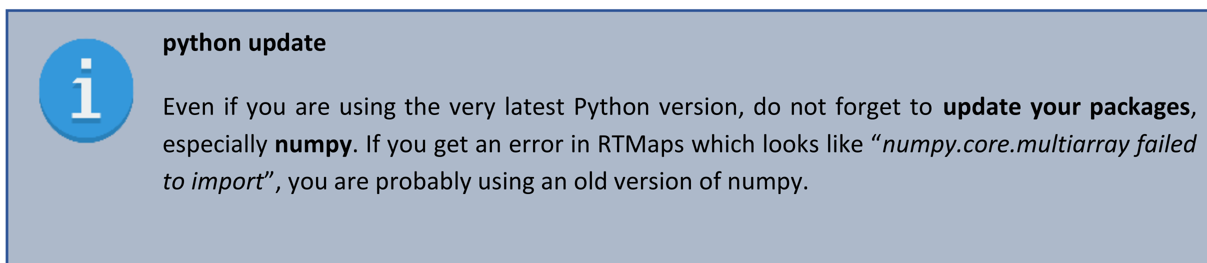


Figure 1 : Advanced Installation Options of Python Setup



python update

Even if you are using the very latest Python version, do not forget to **update your packages**, especially **numpy**. If you get an error in RTMaps which looks like "*numpy.core.multiarray failed to import*", you are probably using an old version of numpy.

Another possibility is to install **Anaconda** distribution. This is very convenient in some cases.

4.2.2 CHOOSE YOUR PYTHON INSTALLATION

There are 2 ways of setting your python installation into the *PythonBridge* component:

- **First method (RECOMMENDED)**

Create **rtmaps_python_bridge.conf** file and specify inside the path containing your installation folder. This file should be placed in the PythonBridge installation folder
(C:\Program Files\Intempora\RTMaps 4\packages\rtmaps_python_bridge\rtmaps_python_bridge.conf)
or in the user directory (C:\Users\[USER]\RTMaps-4.0\rtmaps_python_bridge.conf)
Example: C:\Python37\

- **Second Method**

Specify the Python Installation path directly in the component.

If no python installation folder is set, PythonBridge will try to guess using the Windows Registry.

The search order is:

1. The PythonBridge component **property** PythonInstallationPath
2. *rtmaps_python_bridge.conf* in the **user** folder
3. *rtmaps_python_bridge.conf* in the **installation** folder

4.2.3 HOW DO I INSTALL PYTHON LIBRARIES YOU DO NOT PROVIDE?

Now that you have successfully installed Python3, let's see how to install additional libraries. The recommended method is to use the **pip** package manager. First, you have to start a Windows Shell in the Python3 installation folder.

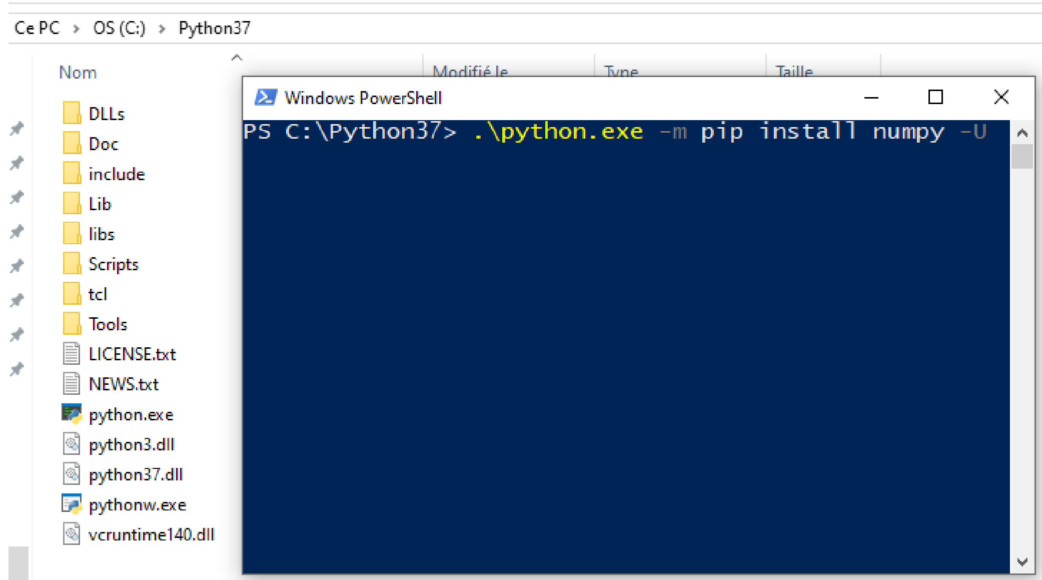


Figure 2 : Open Windows shell to invoke pip

Then, using pip :

- python.exe -m pip install pip –upgrade
- python.exe -m pip install numpy--upgrade
- python.exe -m pip install ***yourpackagename***

For some libraries using Qt like matplotlib or rpy2, you will have to copy the ***qt.conf*** file and paste it in your python folder in RTMaps. The *qt.conf* is located in your Anaconda installation (*C:\...\Anaconda3*) and need to be pasted next to the python_process executable (*C:\...\Intempora\RTMaps 4\packages\rtmaps_python_bridge\python_process\python36*\ for example).

If you are using PySide2, then *qt.conf* is irrelevant.

4.3 LINUX

On Linux RTMaps uses the existing python installation on your distribution. Ubuntu 16.04 and superior are supported. Ubuntu 16.04 has Python 3.5 installed and Ubuntu 18.04 has Python 3.6 installed so RTMaps uses those versions. Only Python3 is available now.

To work properly, **you will have to install** the development package of python and a recent version of numpy as follow (for python3):

- apt-get install python3-dev python3-pip
- pip3 install pip –U
- pip3 install numpy -U

If you have a custom python that defines PYTHONHOME and PYTHONPATH, this is fine but please remember that the python package was compiled against default python coming on the system.

To install python libraries, the procedure is very similar to Windows, using pip3 directly.

- pip3 install **yourpackagename**

4.4 TROUBLESHOOTING

4.4.1 NUMPY IMPORT PROBLEM

If you have the following message : *numpy.core.multiarray failed to import*

Then please update your numpy library following the previous instructions. If you have already done that, then maybe you are not using the Python installation you think you are using.

4.4.2 LIBRARY IMPORT PROBLEM

In some rare cases, you could have some problems using a particular library. We have heard problems with *opencv* and *tensorflow* for now. The problems happens when we try to reload the library, it gives us an error in the code. You should be warn about this in the console. In that case, please uncheck "Auto Reload Submodules" property, it should solve your problem.

5 A QUICK LOOK AT THE PYTHON COMPONENT

Let's put a Python component on the RTMaps diagram. As you can see (Figure 3: Python component), it is a component that has no inputs, no outputs and no properties. Adding inputs, outputs and properties has to be done directly in the python script in the constructor (*Dynamic function only*, see later section).

Figure 3: Python component

To write python code, you can use your favorite editor (Notepad++, Sublime, PyCharm, Spyder, etc.). Please also note that a tiny code editor comes with the python component. You can open it via an action (right click on the component). This editor is generally used to make small code modifications and is convenient to begin with Python.

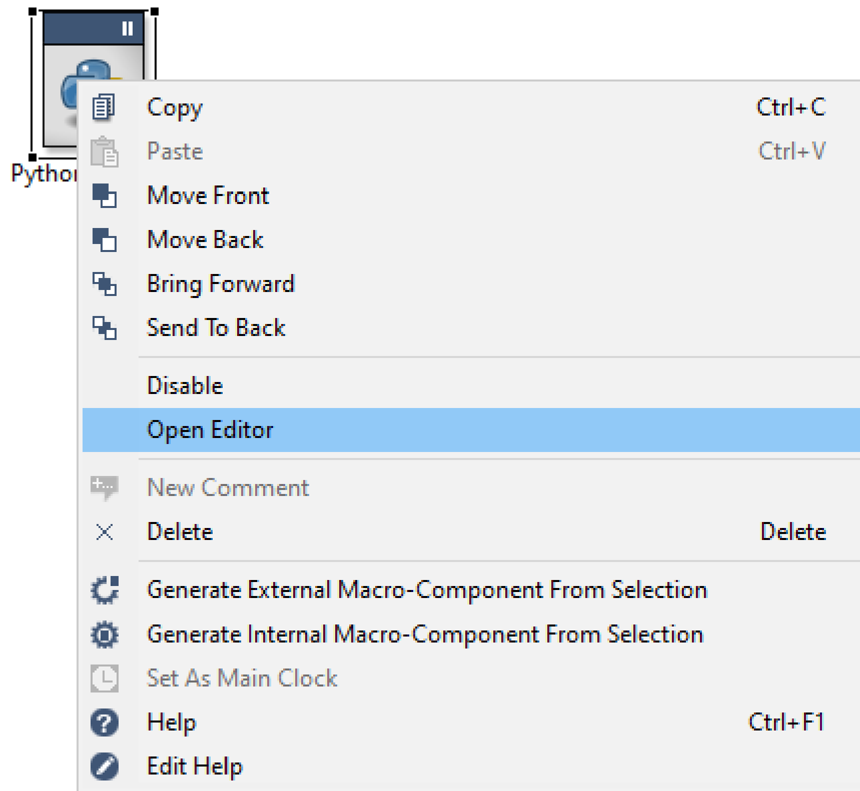


Figure 4: Python action. When right-clicking on the python component, it is possible to open the Python editor. This action will create a new window where it will be possible to create/edit a Python script file.

If you trigger the "Open Editor" action, the code editor will appear and displays a template code that simply copy the input to the output. The Figure 5 shows how this template code looks like. It is composed of one Python class and 4 functions:

- The class constructor: **__init__**(self). It is the Python constructor… Inside this function you can initialize your variables for example. If you want to force the reading policy of your python component, that would be the place to call *self.force_reading_policy* as well.
- ***Dynamic***(self). This function mirrors the C++ SDK, so **if you need to add inputs, outputs or properties, you have to do it inside this function.**
- ***Birth***(self). This function is called once when the diagram starts and every time you save your program dynamically.
- ***Core***(self). Core is called when inputs have arrived on your Python component. Depending on your reading policy, it may be called every new input (Reactive) or only when inputs have a matching timestamp (Synchro). You can also set the reading policy to sampling policy, so that Core will be called periodically at a given sampling rate.
- ***Death***(self). Death is called once when the user shutdowns the diagram.

Code Editor Lite v1.0

No title | Console

```
#This is a template code. Please save it in a proper .py file.
import rtmaps.types
import numpy as np
import rtmaps.core as rt
import rtmaps.reading_policy
from rtmaps.base_component import BaseComponent # base class


# Python class that will be called from RTMaps.
class rtmaps_python(BaseComponent):
    def __init__(self):
        BaseComponent.__init__(self) # call base class constructor

    def Dynamic(self):
        self.add_input("in", rtmaps.types.ANY) # define input
        self.add_output("out", rtmaps.types.AUTO) # define output

# Birth() will be called once at diagram execution startup
    def Birth(self):
        print("Python Birth")

# Core() is called every time you have a new input
    def Core(self):
        out = self.inputs["in"].ioelt # create an ioelt from the input
        self.outputs["out"].write(out) # and write it to the output

# Death() will be called once at diagram execution shutdown
    def Death(self):
        pass

```

Figure 5: Dynamic()/Birth()/Core()/Death() functions; those works similarly to their counterparts in the C++ SDK.

In Python, remember that **indentation** of the code is mandatory. The indentation must be 4 spaces, not tabs.

Now let's take a look at the component properties. If you are using RTMaps 4.5.5 or superior, the properties will show themselves in subcategories : **Advanced** configuration, **Basic** configuration and **User properties**.

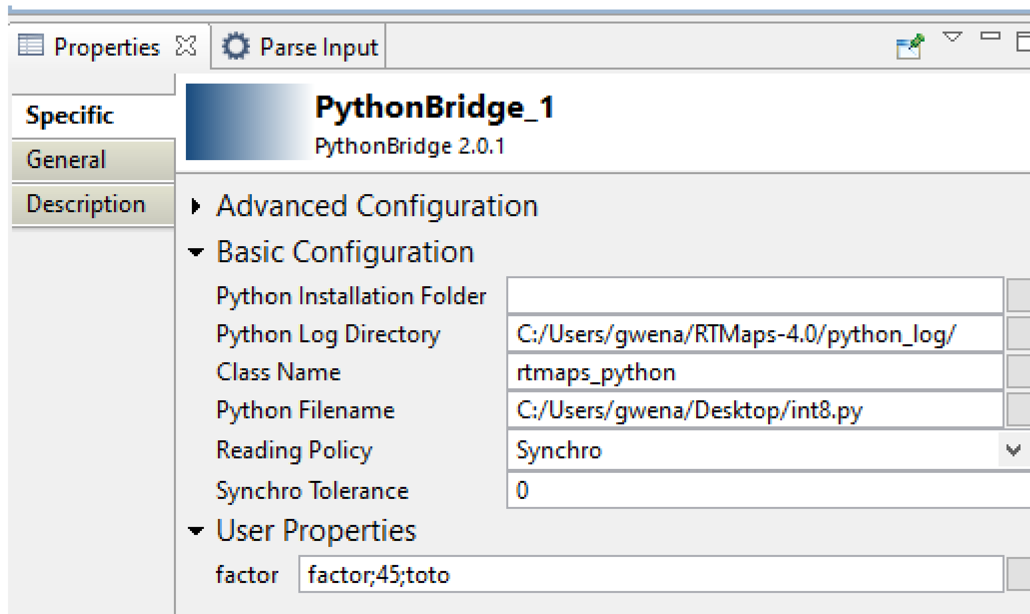


Figure 6: Properties of the Python Component.

5.1 BASIC CONFIGURATION

- ***Python installation folder:*** The python installation RTMapsBridge should use.
- ***Python Filename***: Path to the python script (.py file) to execute. You have to set it yourself.
- ***Python log Directory:*** This log directory will contain all python logs. For each python component on a diagram, a log file will created. The name of the log file will contain the PID of the process, the name of the *PythonBridge* component and the date of the creation of this file.
- ***Reading Policy***: Can be *Synchro*, *Reactive, Sampling or Triggered* (see more in 6.2.3) Reading Policies.

5.2 ADVANCED CONFIGURATION:

- ***Death Behavior:*** When executing *Death()* function in Python, three different strategies can be defined:
  - **Wait for python process indefinitely**: *Death()* is called and we expect that it ends correctly. So the *PythonBridge* component waits for the *Death()* function to be finished. This behavior can be dangerous if *Death()* does not terminate properly as you won't be able to start running again.
  - **Kill process after timeout**. This is the default setting, *Death()* is expected to end within 5 seconds. The python process will be killed if it exceeds this timeout and a new process will be created in that case.
  - **Start new process**. *PythonBridge* component waits 5 seconds, but if the timeout expires, it will not wait for *Death()* to finish its job. Another process will be created to run again. This mode is convenient if you want to keep many plots alive when using matplotlib for example.
- ***Show Traceback In Console:*** Show python tracebacks in RTMaps console when true. This is convenient to quickly read an error without looking at the python log file.

- ***Auto Reload SubModules****:* When checked, if you save your python script during execution, all modules used in your script will be reloaded as well (for example, numpy, scipy, etc…). Depending on how your external modules are imported, it will be reloaded or not. For example, if you use the "import foo" syntax, the module foo will be reloaded. But if you use the "from foo import *" nothing will be reloaded.

- ***Memory Size***: Set the size of the shared memory. The total shared memory must be able to contain inputs, outputs and properties. The default value is fine in most cases but you can adjust it by hand if necessary. If the space is not enough, the Python component will adjust this value automatically and reboot itself. In that case a sample will be lost.

- ***PID***: The PID of the python process. This is useful to attach to the python process with a debugger.

- ***Additional Environment Variables***: Empty field by default. You can set additional variables here that will be passed to the process during its creation. *LD_LIBRARY_PATH* or other environment variables might be useful here.

5.3 USER PROPERTIES

This subcategory will contain all the properties contained in the Python code. It is listed in a subcategory for display convenience.

6 INPUTS, OUTPUTS AND PROPERTIES

When naming an input/output, it is impossible to name them the same way as a property of the component itself (for example: it is impossible to name an input 'x' as this property name already exists).

6.1 DEFINITION

As you can see in Figure 5 the rtmaps_python class inherits from the *BaseComponent* class, declared in the base_component.py file. We recommend you to look at it, you can get it from the following path:

…\Intempora\RTMaps 4\packages\rtmaps_python_bridge\python_process\python36\rtmaps

There you can see some useful functions like *commit_suicide()* or *name()*. But the three most important are:

- ***add_input****(self, name, type, typename):* you use it by writing *self.add_input("name", rtmaps.types.ANY)*.
    - ***name:*** the name of your input. It must be unique.
    - ***type:*** If you want all possible inputs, you can specify *rtmaps.types.ANY.* Otherwise you can specify one of the following supported types: *rtmaps.types.INTEGER8, rtmaps.types.UINTEGER8, rtmaps.types.INTEGER16, rtmaps.types.UINTEGER16, rtmaps.types.INTEGER32, rtmaps.types.UINTEGER32, rtmaps.types.INTEGER64, rtmaps.types.UINTEGER64, rtmaps.types.FLOAT32, rtmaps.types.FLOAT64, rtmaps.types.STREAM8, rtmaps.types.TEXT_ASCII, rtmaps.types.CAN_FRAME, rtmaps.types.CANFD_FRAME, rtmaps.types.IPL_IMAGE, rtmaps.types.MAPS_IMAGE,*

rtmaps.types.REAL_OBJECT, rtmaps.types.DRAWING_OBJECT, rtmaps.types.MATRIX, rtmaps.types.CUSTOM_STRUCT

  - ***typename:*** only useful for *CUSTOM_STRUCT*, it expects the name of the custom structure to be that name.

- ***add_property****(self, name, value, subtype = 0, flag = 0):* you use it by writing *self.add_property("name", value)*
  - ***name:*** the name of your property. It must be unique
  - ***value:*** the value of the property
  - ***subtype:*** optional argument. It can be rtmaps.types.FILE, rtmaps.types.PATH or rtmaps.types.ENUM
  - ***flag:*** option argument. It can be rtmaps.types.MUST_EXIST. This flag only applies for FILE and PATH subtype.
- ***add_output****(self, name, type, buffer_size = 0):* you use it by writing *self.add_output("name", rtmaps.types.AUTO, size, typename)*.
  - ***name:*** the name of your output. It must be unique.
  - ***type***: The type of your output. If you want to force the output type, you can specify it here. For example (rtmaps.types.INTEGER64 for 64-bit integers). You can also use rtmaps.types.AUTO which means that the output will be typed automatically when the first write will be done on the output.
  - ***buffer_size:*** The maximum size of your output (like the C++ SDK). This size is used to make the allocation of the output on RTMaps side. If you put 0, the memory allocation will be delayed and will be automatically calculated using the ioelt later on.
  - ***Typename:*** In case of type if CUSTOM_STRUCT, you can specify here the name of the custom structure desired.

6.2 INPUTS

6.2.1 AVAILABLE TYPES AS INPUTS

All RTMaps types except dynamic custom structs can be read in PythonBridge. Here is the exhaustive list:

- Integer8, UInteger8 (alone or array)
- Integer16, UInteger16 (alone or array)
- Integer32, UInteger32 (alone or array)
- Integer64, UInteger64 (alone or array)
- Float32 (alone or array)
- Float64 (alone or array)

- Stream8
- Text ASCII and UTF-8
- CAN Frames and CANFD Frames. The vector will be represented as a python list []
- IPL Image and MAPSImage
- Matrix
- Real Objects. The vector will be represented as a python list []
- Drawing Objects. The vector will be represented as a python list []
- Custom structure

6.2.2 ACCESSING INPUTS

You can read inputs using the ***self.inputs*** dictionary. {key = **name** : value = **IOElt**}

This dictionary **keys** are the inputs name and the **values** are *Input* class instances. So by writing *self.inputs["foo"]*, you will get an *Input* instance of the input named "foo". As the inputs are stored into a dictionary, you can easily iterate on all the inputs as well. The *self.inputs* dictionary is updated every time a new input is available. So while the "foo" input has not received any data, the *self.inputs* dictionary will not contain the key "foo". Moreover, if the "foo" input already received some data but has not been updated for a while, it still contains the old data, without any update.

WARNING: SHALLOW COPY vs **DEEPCOPY**

When you write *out = self.inputs["input"].ioelt* to access the input Ioelt, you do NOT get a copy of the Ioelt. In python, the Plain Old Data (simple integers, floats, etc…) are fully copied but for complex types (dict, structs, …) only shallow copies are performed. So here *out* is just an alias for the input Ioelt, do not modify it unless you know you can do it. Please use copy.deepcopy if you want a full copy.

`https://docs.python.org/3.6/library/copy.html`

```
class Input:
    def __init__(self):
        self.type = 0
        self.name = "name_of_this_input"
        self.ioelt = None # IOElt
```

Figure 7: Input Class

The Input class has three member variables:

- **type**: the type of the data. All data types are stored into the types.py python file.
- **name**: The name of the data. This name is already present in the key, it is repeated here for convenience.
- **ioelt**: This contains the data itself in the form of IOElt. Please see section 7 to know more about it.

If you have multiple inputs in reactive mode, all inputs do not arrive simultaneously on your Python component. So at the beginning of the diagram execution, *self.inputs* will not contain all inputs for some time. You can test if a specific input "foo" is available by testing the *"foo in self.inputs"* expression. Or you can just use your code normally and expect to have a *KeyError* exception when you try to access to a data that does not exist in the dictionary for now.

6.2.3 READING POLICIES

The reading policies are:

- ***Synchro***: This is the same as SynchroStartReading in C++. All the inputs will be read if they are under the synchro tolerance threshold. It means if you choose 0 tolerance, you will receive only inputs that have the exact same timestamp.

- ***Reactive***: This is the same as multiple StartReading. The Core() function will be called every time a new input has arrived. When a new data arrive, the variable *self.input_that_answered* is set to the actual input that answered so that you can know which input has arrived precisely. Note also that Reactive has a *timeout* property set to zero by default, which means your Python component will wait for a data forever. But if you plan to wait for a data for a specific duration and not more, then you can specify a timeout value in microseconds. If the timeout occurs, *Core()* will be called with a *self.input_that_answered* set to **-1**.

- ***Sampling***: Here you choose to launch *Core()* every X microseconds. In this mode, you are independent of the inputs sampling rate.

- ***Triggered***: The Python component will be triggered by the **first input**. Every time a new data comes on the first input, *Core()* will be called.

Note that if the Python component has no input at all and you are not in sampling mode, *Core()* will be called in a loop, so you have to use a blocking function like *sleep* (from *time* package) for example.

You can force the reading policy from the python code by using the *force_reading_policy* function. This function should be called from the *__init__(self)* function only.

self.force_reading_policy(rtmaps.reading_policy.REACTIVE, optional_additional_argument)

The four reading policies are available:

- *rtmaps.reading_policy.REACTIVE*.

- *rtmaps.reading_policy.SAMPLING*

- *rtmaps.reading_policy.SYNCHRO*

- *rtmaps.reading_policy.TRIGGERED*

For the REACTIVE, SAMPLING and SYNCHRO, you can pass an additional parameter to fix respectively the *timeout*, *sampling_period* and *tolerance*. If the additional parameter is not set, then the user is free to choose a value for it.

6.2.4 I/O TYPE MAPPING

From RTMaps to Python:

RTMaps types	Python types
Integer8	Long
Integer8 array	*numpy.ndarray<NPY_INT8>*
UInteger8	Long
UInteger8 array	*numpy.ndarray<NPY_UINT8>*
Integer16	Long
Integer16 array	*numpy.ndarray<NPY_INT16>*
UInteger16	Long
UInteger16 array	*numpy.ndarray<NPY_UINT16>*
Integer32	Long
Integer32 array	*numpy.ndarray<NPY_INT32>*
UInteger32	Long
UInteger32 array	*numpy.ndarray<NPY_UINT32>*
Integer64	Long
Integer64 array	*numpy.ndarray<NPY_INT64>*
UInteger64	Long
UInteger64 array	*numpy.ndarray<NPY_UINT64>*
Float32	Float
Float32 array	*numpy.ndarray<NPY_FLOAT32>*
Float64	Float
Float64 array	or *numpy.ndarray<NPY_FLOAT64>*
Stream8	*numpy.ndarray<NPY_UINT8>*
TextAscii	Unicode object (*PyUnicode_FromString* is used)
CAN Frames	**List** of CANFrame. See examples
CANFD Frames	**List** of CANFDFrame. See examples
Drawing Objects	**List** of DrawingObject. See examples
Real Objects	**List** of Real Objects. See examples

Custom Structure	*numpy.ndarray<NPY_UINT8>*

Table 1: Type Mapping RTMaps to Python

CANFrames, CANFDFrames, RealObjects, DrawingObjects arrives to Python in a list[], so you can access to every sample in the list using the [] operator for example.

Custom Structure is mapped as Raw data, you can pack it yourself, see samples about custom structures in the samples folder (chapter2).

6.2.5 ADDITIONAL VARIABLES

self.rtmaps_diagram is defined so that you know the absolute path of the RTMaps running diagram from Python.

6.3 OUTPUTS

6.3.1 AVAILABLE TYPES FOR OUTPUT

All the types available for inputs are available for outputs. Except Integer64 and Float64, all types are not available as single number directly, because 64bit types are promoted for simplicity. So if you want to output a 32bit integer or 16bit integer, or 8bit integer, it is still possible but you will have to create a numpy array like this:

```
def Core(self):
    out = self.inputs["in"].ioelt # create an ioelt from the input
    out.data = np.array([out.data]) # 32bit
    self.outputs["out"].write(out) # and write it to the output
```

Use the numpy ***astype*** function to cast the array to another type.

List of available types for output:

- Integer8, UInteger8 (only for arrays)
- Integer16, UInteger16 (only for arrays)
- Integer32, UInteger32 (only for arrays)
- Integer64
- Float32 (only for arrays)
- Float64
- Stream8
- Text ASCII and UTF-8 (this is the same thing in RTMaps)
- CAN Frames and CANFD Frames. The vector will be represented as a python list []
- IPL Image and MAPSImage
- Matrix
- Custom structures

6.3.2 HOW TO WRITE ON OUTPUTS

You can access to outputs using the ***self.outputs*** dictionary. {key: value}

This dictionary **keys** are the outputs name and the **values** are *Output* class instances. So by writing self.outputs["foo"], you will get an Output instance of the output named "foo".

```
class Output:
    def __init__(self):
        self.type = 0
        self.name = "name_of_this_ouput"
        self.buffer_size = 0

    def write(self, data):
        rt.write(self.name, data)

    def alloc_output_buffer(self, size):
        self.buffer_size = size
        rt.alloc_output_buffer(self.name, size)
```

Figure 8: Output Class

The most important function would be the ***write*** function. It is used that way: *self.outputs["name"].write(out)*. The variable "out" can either be an IOElt (see more in section 7) or a data. We recommend to use the IOElt if you want to specify timestamps, vector_size, etc…

The *alloc_output_buffer* function allows to set the buffer_size of the output, the same way as you would do it in C++. It is only useful to call it once as an output will be allocated only once for a run.

To sum up, please note that there are **three ways** to allocate an output to a specific size:

- Use the ***add_output*** function in the __init__ constructor and specify a buffer_size.
- Use the ***alloc_output_buffer*** function just like above. Call this function once in the Birth() or Core() method.
- Specify a ***buffer_size*** in the IOElt. This buffer_size will be used if the output has not been initialized yet.

If you write a data as is without using any of the three methods above, then the output will be allocated using the actual size of the data. For example, if you write *self.outputs["out"].write("hello")*, then the size of the output will be adjusted to fit the "hello" string, not more. If you plan to write "Hello World !" during the same run, then a memory overflow will happen and you will get an error, because the second string does not fit into the allocated output.

As a matter of fact, there is another way to write on outputs, *self.write(0, "hello")* and *self.write("out", "hello")* will do the job as well. You can also specify the timestamp in this convenience function like this:

self.write(0, "hello", ts) **or** *self.write("out", "hello", ts)*

6.3.3 I/O TYPE MAPPING

From Python to RTMaps:

Python types	RTMaps types
Long	Integer64
Float	Float64
Bytes	Stream8
Unicode string	TextAscii
List of CANFrame	CAN Frames
List of CANFDFrame	CANFD Frames
List of DrawingObject	Drawing Objects
List of Real Objects	Real Objects
numpy.int8	Integer8
numpy.uint8	UInteger8
numpy.int16	Integer16
numpy.uint16	UInteger16
numpy.int32	Integer32
numpy.uint32	UInteger32
numpy.int64	Integer64
numpy.uint64	UInteger64
Custom type	Custom type

Table 2: Type Mapping Python to RTMaps

CANFrame, CANFDFrame, DrawingObject, RealObjects have to be contained in a list, even if there is only one sample.

In order to output custom type, you have to set the output type to CUSTOM_STRUCT explicitly. You will find examples of Custom structure in the examples folder of the PythonBridge installation. You can handle custom structs in 2 ways: using ctypes or pack/unpack from struct library.

- Here is ctype link: https://docs.python.org/3.7/library/ctypes.html
- Here is a link of supported types if you want to use pack/unpack: https://docs.python.org/3/library/struct.html#format-characters

6.4 PROPERTIES

You can access to properties using the ***self.properties*** dictionary. {key: value}

This dictionary **keys** are the properties name and the **values** are *Property* class instances. So by writing *self.properties["foo"]*, you will get a *Property* instance of the property named "foo".

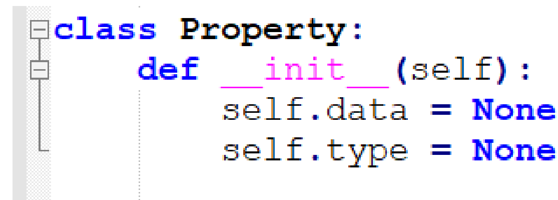


Figure 9: Property class

Properties can be of four types only: integer, double, bool or string. Like in the C++ SDK, the type is defined according to the python type. Examples:

- *self.add_property("foo", 42)* defines a **INTEGER64** property
- *self.add_property("foo ", 42.123)* defines a **FLOAT64** property
- *self.add_property("foo ", True)* defines a **BOOL** property
- *self.add_property("foo ", "Hello")* defines a **TEXT_ASCII** property
- *self.add_property("foo ", "C:/tmp", rtmaps.types.FILE)* defines a **FILE** property
- *self.add_property("foo", "Hello", rtmaps.types.PATH, rtmaps.types.MUST_EXIST)* defines a **PATH** property that must exist (like the C++ SDK).
- *self.add_property("foo", "3|0|a|b|c", rtmaps.types.ENUM)* defines a **ENUM** property (like the C++ SDK). The first number is the total number of choices present in the ENUM, the second number is the default index, then the enum separated by pipes.

Properties are exposed in the RTMaps component so that they can be modified by the end user without touching the Python script. Please note that the value modified by the user will remain even if the action "Update Inputs, Outputs and Properties" is called, except if the property type has changed.

For example, let's assume a property "factor" has been defined in the python script with a value of 2. A property will be created in the RTMaps component accordingly. If the user set it to 3, this value of 3 will never be changed unless the property "factor" gets a new type in the python script (double or string for example).

7 IOELT

In python inputs and outputs are **Ioelt**, as in the RTMaps SDK C++. An IOElt encapsulate the data itself with additional information (called "meta-information") such as the *timestamp* or the *buffer_size*. The Figure 10 shows the complete class declaration.

```
#Standard RTMaps type
class Ioelt:
    def __init__(self):
        self.data = None          # could be any type here : long, numpy array, CanFrame, IplImage, etc...
        self.buffer_size = 0      # buffer_size
        self.ts = None            # timestamp of the data
        self.toi = 0              # time of issue, READ-ONLY !

        # Next fields are optional
        self.vector_size = None # Set it to 0 to write an empty data. Otherwise it is AUTO set
        self.frequency = None
        self.quality = None
        self.misc1 = None
        self.misc2 = None
        self.misc3 = None
```

Figure 10: Ioelt Class

You can check the following contents in the three different files: types.py, drawing_object.py, real_objects.py. Normally the path to get those files is:

…\Intempora\RTMaps 4\ packages\rtmaps_python_bridge\python_process\python36\rtmaps

An IOElt has several variable:

- ***data**:* can be of any type (long, numpy array, IplImage, MapsImage, CanFrame, DrawingObject, RealObject…).
- ***buffer_size**:* same as the buffer_size of the MAPSIOElt (see C++ SDK).
- ***ts**:* timestamp of the data.
- ***toi**: Time of Issue. T*this variable is read-only.
- ***vector_size:*** If set to zero, you will write an empty data to your output. This is the only way to write a data that contains nothing. If set to 10 with a data containing 30 elements, then only the first 10 will be written on the output.
- ***frequency, quality, misc1, misc2, misc3:*** optional variables. These variables are the very same as the C++ SDK variables.

8 USING RTMAPS FUNCTIONS

These functions are found in *rtmaps.core* which needs to be imported, you can for example write *import rtmaps.core as rt*. Please refer to Python documentation about imports if needed.

Here is the list of functions that you can call in Python:

- ***is_dying***(): returns true if RTMaps was asked to shut down.
- **name**(): returns the name of the current Python component.
- ***report_info***(string): write some information on the RTMaps console (blue)
- ***report_warning***(string): write some information on the RTMaps console (yellow)
- ***report_error***(string): write some information on the RTMaps console (red)

- ***parse***(string): call the RTMaps Engine with a command
- ***async_parse***(string): call the RTMaps Engine in an asynchronous way. Very useful if you plan to call such action as « shutdown »
- ***get_property***(string): get the property value from an RTMaps component. For example, get_property("`Matrix_constant_generator_1.nbRows`") will retrieve the property 'nbRows' of the component *Matrix_constant_generator* you have on the diagram
- ***current_time***(): retrieve the MAPS::CurrentTime()
- ***time_speed***(): retrieve the MAPS::TimeSpeed()
- ***wait(int):*** wait for a specific appointment. This calls the Wait() from C++ SDK.
- ***rest(int):*** rest for a specific duration. This calls the Rest() from C++ SDK. The difference between rest and sleep is that rest follows the time speed of RTMaps.

9 DEBUGGING

Debugging is done as usual, you can attach any debugger (PyCharm for example) to the python process and add breakpoints, inspect variables… This is very convenient to look deep into your code.

Note that every python component spawns a separate process for running python interpreter. So if you have several python components in your diagram you will have to choose which process (and consequently which python interpreter) you want to attach to.

To do that, every Python component has a read-only property **pythonPID** that indicates the PID of the python process associated to this component. Thanks to this PID, you will be able to attach to a particular python interpreter and inspect your code as shown by the Figure 11 : Debugging with PyCharm.

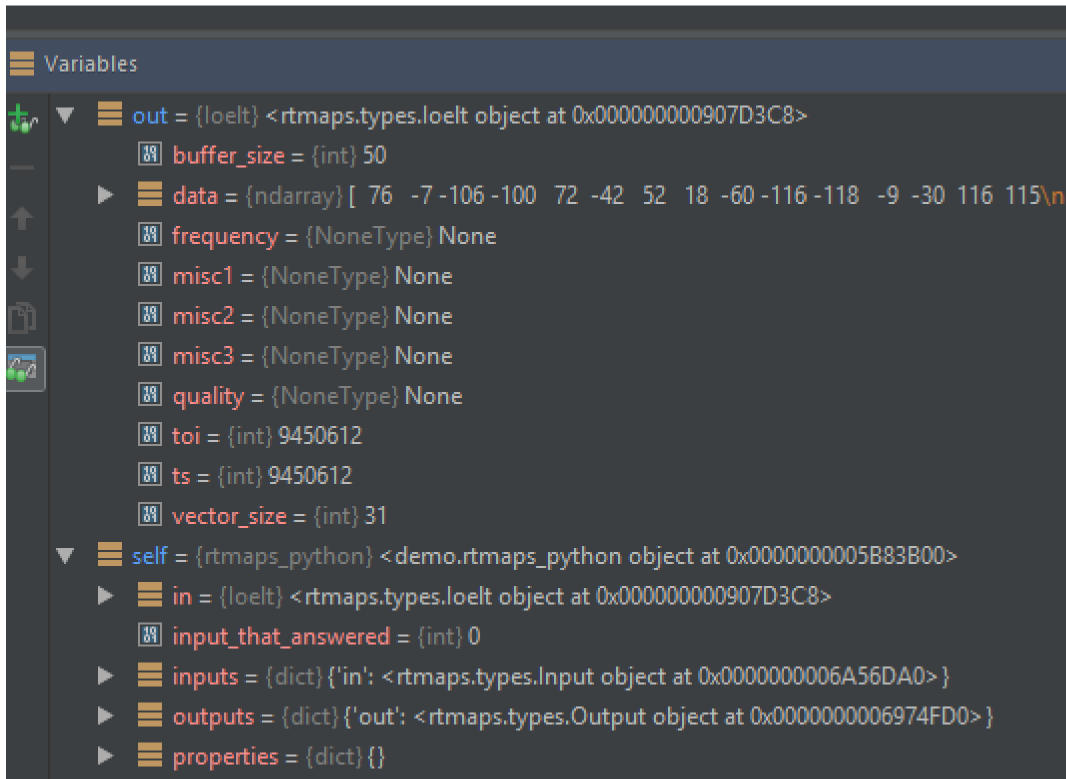


Figure 11 : Debugging with PyCharm

10 ADVANCED USAGE

Remember that the python interpreter runs in a separate process. Communication with the PythonBridge component is done through shared memory.

You can set additional environment variables for the python process thanks to the property `additionalEnvironmentVariables.` The format is the following:
`LD_LIBRARY_PATH=/opt/toto;OTHER=/home/dev;ANOTHER:/tmp`

11 LIBRARIES

In this section we will see some of the popular library in python. Don't forget to copy/paste the *qt.conf* file as seen in section 4.2.2 Windows, otherwise the Qt dependencies won't be found.

Please look at the samples in the run_python installation directory to know more about those libraries.

11.1 MATPLOTLIB

Matplotlib is a Python plotting library. You can easily generate plots, histograms, bar charts… by writing a few lines of code. Thanks to the property "Keep Python Alive After Death" of the Python component, you can also keep your graph open after you shut down your diagram and so analyze all the data collected. Please refer to the python sample to see a few possibilities of using Matplotlib.

Note: If you don't check the "Keep Python Alive After Death" property the Python component will stop but it will take some time, approximately four seconds.

More about Matplotlib here: https://matplotlib.org/

11.2 PIL

PIL (Python Imaging Library) is a library for image manipulation. It contains many modules that will allow you to process images of many format. In RTMaps, thanks to its fairly powerful image processing capabilities, it can help you with your image application.

More about PIL here: https://pillow.readthedocs.io/en/4.3.x/

11.3 RPY2

RPy2 is a python library to facilitate the use of R in Python scripts. R objects are exposed as instances of Python-implemented classes. R is a programming language, it is used for developing statistical software and data analysis. You can also create graphics with it.

Note: Even if you don't check the "Keep Python Alive After Death" property with rpy2, your graphs will stay alive after the shutdown.

More about rpy2 here: https://rpy2.readthedocs.io/en/version_2.8.x/#

11.4 TENSORFLOW

TensorFlow is an open source software library for numerical computation using data flow graphs. It can be used in a Python script. We invite you to go take a look at our Object detection with TensorFlow and Python demos. You can find it at this address: https://support.intempora.com/hc/en-us/articles/115002778193-Object-detection-with-TensorFlow-Python

Note: In order to download the library TensorFlow, we recommend you to use the standard package installer pip. Also you need to uncheck the "Auto Reload SubModules" when you use TensorFlow.

More about Tensorflow here: https://www.tensorflow.org/

11.5 PYQT

PyQt is a library that links the Python language with the Qt library. It enable to create graphic interface in python just by writing a few lines of code.

Note: When you shut down the diagram the Python component will be killed but it will take some time with PyQt, approximately seven seconds. Also remember to uncheck the "Keep Python Alive After Death" property while using PyQt in your script.

More about PyQt here: https://www.riverbankcomputing.com/software/pyqt/intro

11.6 LUPA

Lupa is a python library that integrates the runtimes of Lua into Python. Lua is a lightweight, multi-paradigm programming language. It is useful for embedded systems since the language runtime is very small.

More about Lupa here: https://pypi.python.org/pypi/lupa