GT-SUITE Python API

Getting started

API Documentation

Examples

Changelog

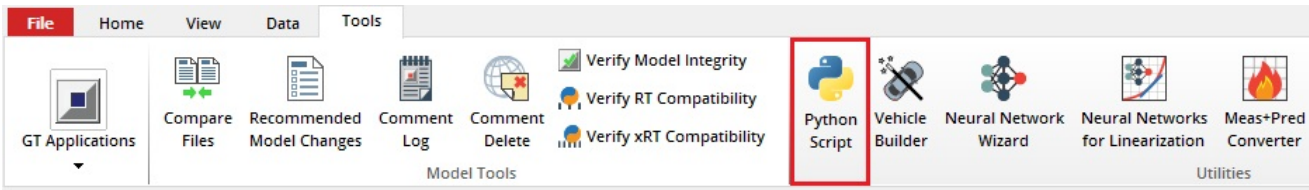
- v2021
- v2022

Python in GT-ISE

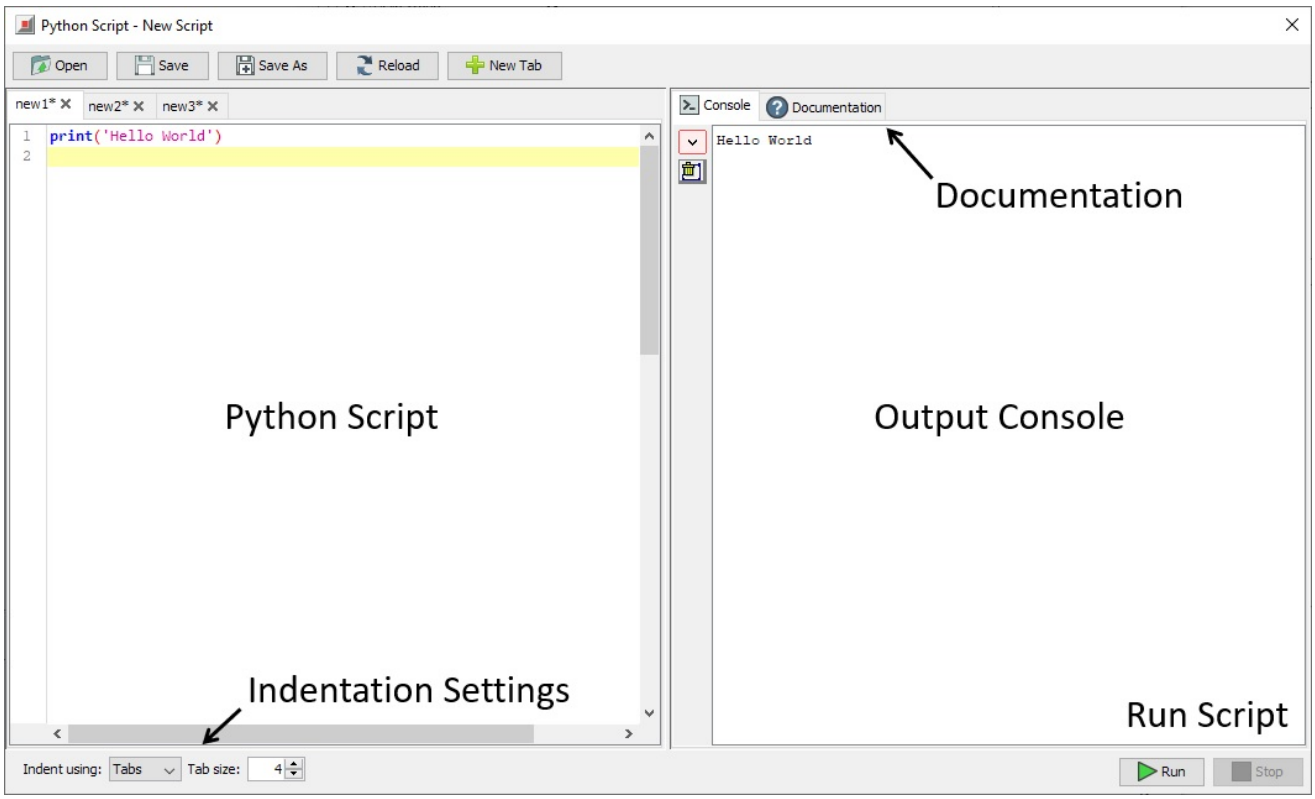
The GT-SUITE Python API can be used to programatically automate model-building tasks inside GT-ISE, such as creating new objects/parts, populating data, or configuring Case Setup. GT-ISE contains some additional features to aid developers in utilizing the API.

Built-In Python Editor

A simple, easy to use Python script editor/interpreter is built directly into GT-ISE. It can be accessed from the Tools tab > Utilities.



The built-in Python script editor includes features such as a multi-tab interface, syntax highlighting, an output panel, and the ability to run scripts directly from the editor.



Indentation Settings

The built-in Python editor has an auto-indent feature that automatically adds indentation after lines that require it (ex: *if* statements). This feature allows either tabs or spaces to be used for indentation.

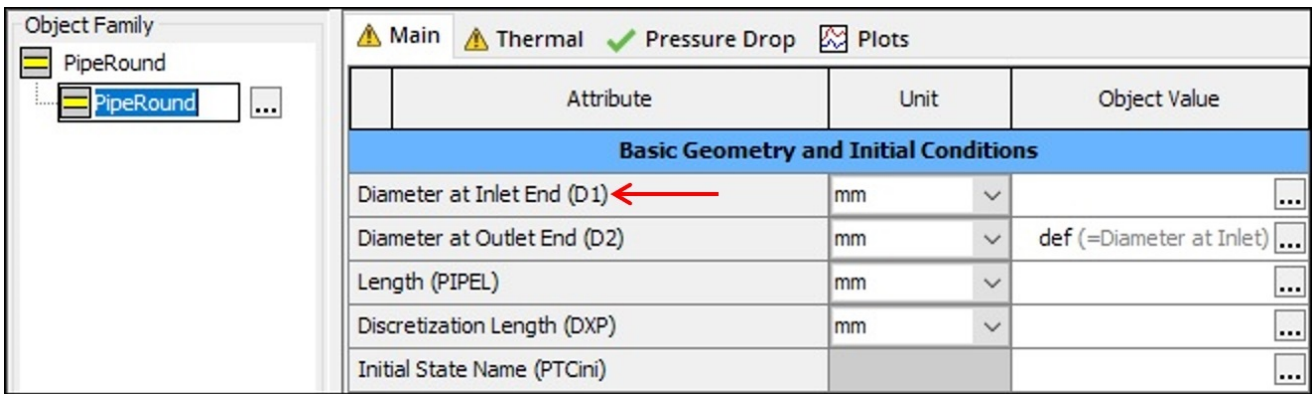
- When set to *Tabs*, the *Tab Size* setting changes the spacing inside the editor for display purposes only. This spacing does not get saved to the Python file, as the indentation is always saved as a tab character. The auto-indent feature will will insert one tab for each indentation level.
- When set to *Spaces*, the *Tab Size* setting determines how many spaces will be used by the auto-indent feature for each indentation level. This spacing will get saved to the Python file, as the indentation is saved as multiple discrete space characters.

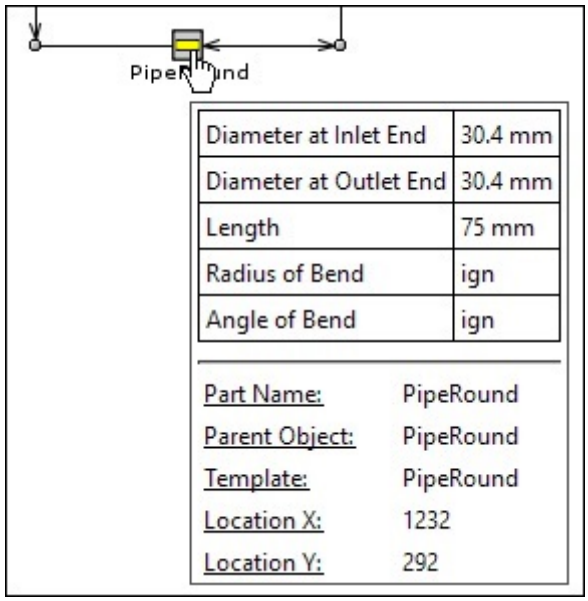
These settings should be chosen before starting a new Python script. If an existing Python script is opened, the editor will attempt to detect the indentation scheme that was previously used. If the settings are changed on an existing script (or inconsistencies are detected on opening), the editor will attempt to convert the tabs to spaces or vice versa. Note that Python 3 does not allow mixing the use of tabs and spaces for indentation.

Python Developer Info

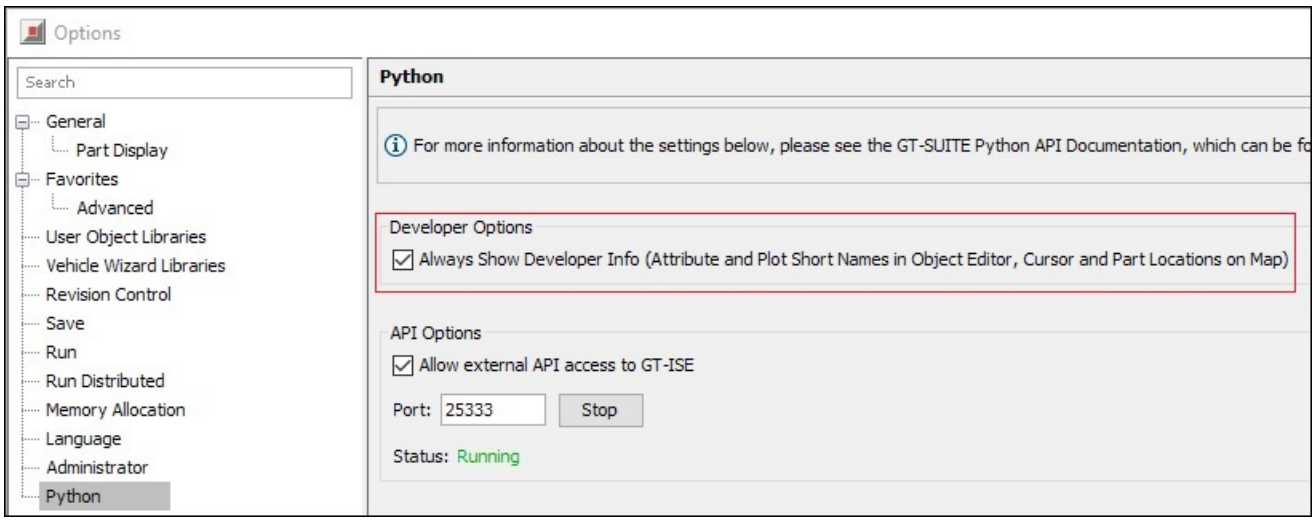
When the built-in Python editor window is open, GT-ISE displays additional information in the GUI that can be helpful when developing Python scripts. This includes:

- Attribute “short names” in the Object Editor (used when referring to attributes by name in the API)
- Plot “short names” in the Object Editor (used when referring to plots by name in the API)
- Cursor XY-location on the map
- Part locations on the map via tool-tip



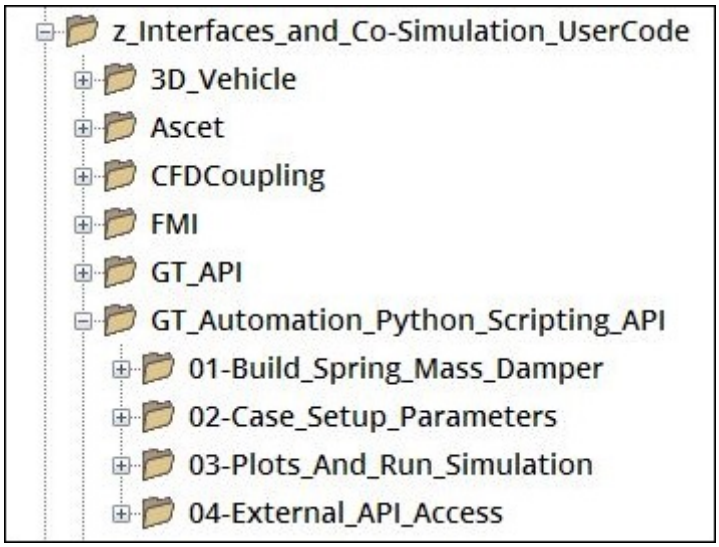


This feature can also be turned on at all times via the setting in File > Options > Python > Always Show Developer Info.



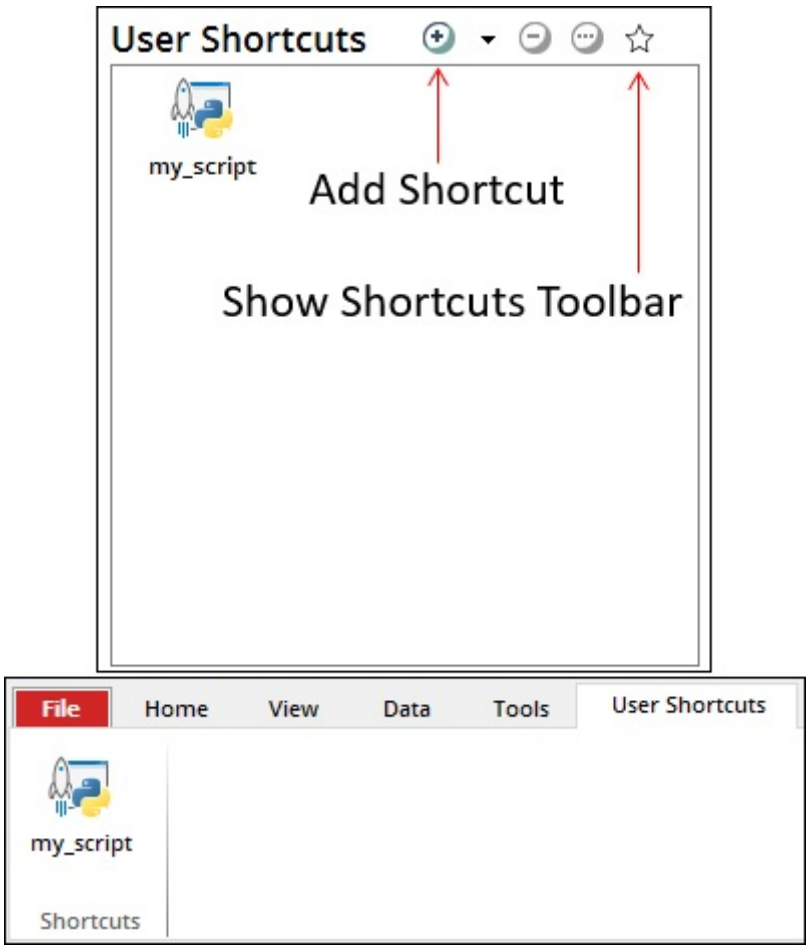
Examples

Example models and Python scripts can be found in GT-ISE examples browser by navigating to **File > Examples > z_Interfaces and Co-Simulation UserCode > GT Automation Python Scripting API**. Descriptions and explanations of these examples can be found in the Examples section of this documentation.



User Shortcuts

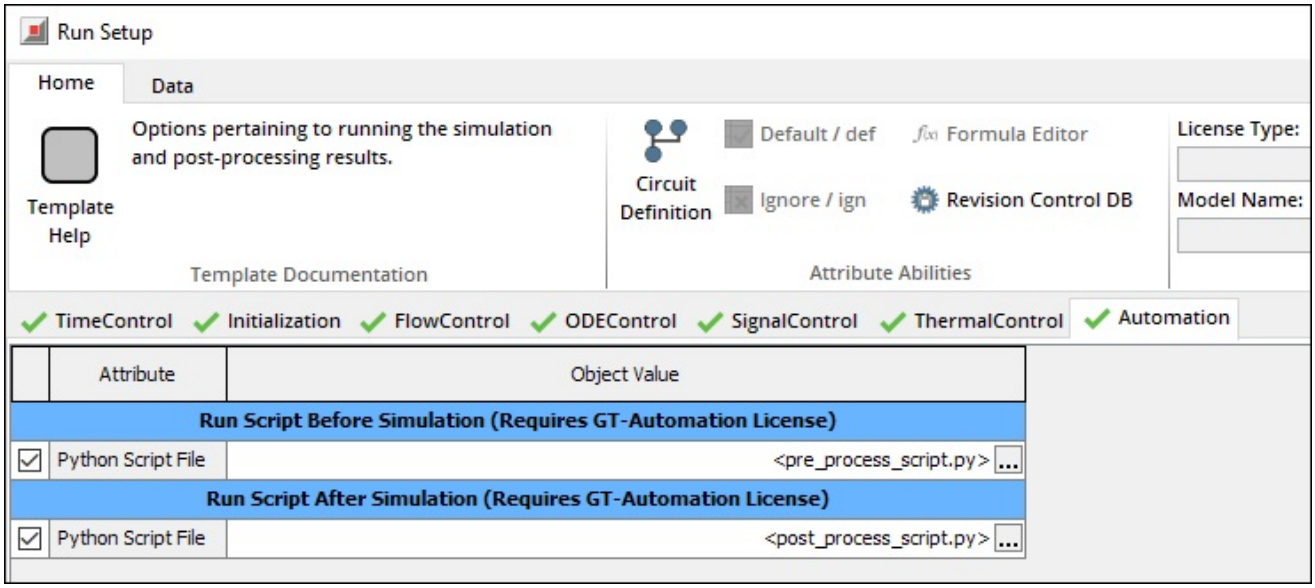
Shortcuts to commonly-used applications and files can be created inside the File > Resources menu of GT-ISE. This includes Python scripts, which can be added to this panel by clicking on the (+) icon and selecing 'Python File (*.py)'. Clicking on a User Shortcut for a Python script will execute that script using the specified Python interpreter in GT-ISE.



User Shortcuts can also be displayed as a toolbar tab. This option can be turned on by clicking on the star icon above the User Shortcuts panel, or inside of File > Options > Customize Ribbon > Add Tab for User Shortcuts.

Automatic Execution from Models

Models can be set up to automatically execute Python scripts each time a simulation is run, either before the simulation starts or after it completes. These settings can be found inside the Run Setup menu of every model:



Run Script Before Simulation

When this option is turned on, the specified script is executed after clicking the "Run Simulation" button (or after launching *gtsuite* from command line). Once the script has completed, the resulting model goes through standard model validation, then a *.dat* file is created and sent to the GT-SUITE solver. It should be noted that the script operates on a *copy* of the model, which is temporarily saved in the model directory with a *.gtgen* file extension. This file gets deleted automatically once the *.dat* file is created. **Modifications are not saved to the original model** and will not be visible in the GT-ISE GUI.

This feature can be useful when it is desirable to have a model-specific script run every time the simulation is started, but without making permanent changes to the original model. Some examples may include scripts that perform additional validation checks or reporting on the model, or scripts that build models on-the-fly using the base model and certain user selections.

When using the GT-SUITE Python API in conjunction with this feature, the *get_active_document()* method should **always** be used to refer to the model. This method will always return the model/document that launched the script, regardless of which document is active in GT-ISE, ensuring that the correct document is accessed. The *get_document(filename)* method may produce unpredictable results and should not be used.

Similarly, the *get_instance()* method will always return the instance of GT-ISE from which the simulation was started (including command line runs), regardless of the port and version arguments specified. Python scripts that are run using this feature cannot connect to a different instance or create a new one.

Additionally, some functions of the GT-SUITE Python API are restricted for scripts that are run using this feature. This includes methods related to opening/closing of models, saving models, and launching other simulations.

Run Script After Simulation

When this option is turned on, the specified script is executed when the simulation completes, after the results (.glx) file has been generated. Note that the script always executes after the simulation completes, even if the simulation fails, or is a pre-processing run.

This feature can be useful for performing post-processing tasks on model results automatically, each time the simulation is run. Some examples may include generating report (.gu) files based on the model results, or extracting data from the results file.

It should be noted that this script is executed by the *solver* process, which is independent of GT-ISE and GT-POST. When using the GT-SUITE Python API in conjunction with this feature, the script behaves as though it is accessing the API "externally", and should be initialized as such. For more information on how to set this up, please see the section **Getting Started > External API Access**.

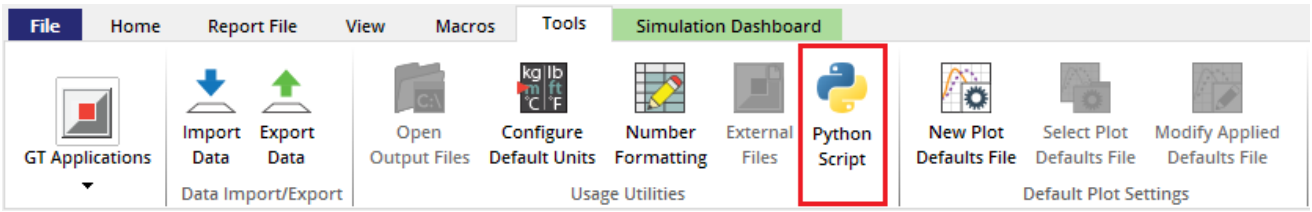
For distributed simulations (local and cluster), the script is executed on the client machine during fetching (downloading) of the result file; the script cannot be executed on the distributed cluster itself.

Python in GT-POST

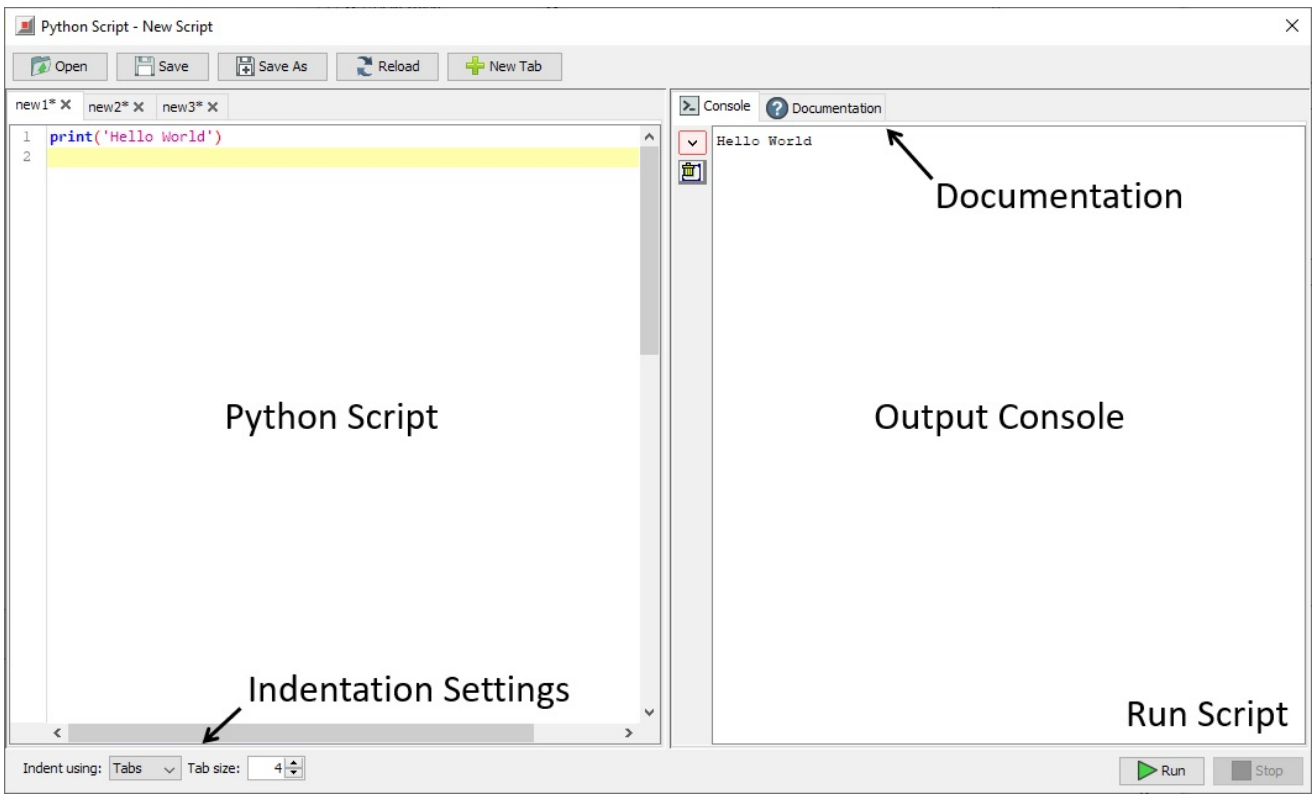
The GT-SUITE Python API can be used to programatically automate post-processing tasks inside GT-POST, such as querying results (.glx) files for instantaneous plots and case/time RLTs, retrieving raw data from results, and copying/formatting plots into custom report (.gu) files. GT-POST contains some additional features to aid developers in utilizing the API.

Built-In Python Editor

Similar to GT-ISE, a Python script editor/interpreter is built directly into GT-POST. It can be accessed from the Tools tab > Usage Utilities.



The built-in Python script editor includes features such as a multi-tab interface, syntax highlighting, an output panel, and the ability to run scripts directly from the editor.



Indentation Settings

The built-in Python editor has an auto-indent feature that automatically adds indentation after lines that require it (ex: *if* statements). This feature allows either tabs or spaces to be used for indentation.

- When set to *Tabs*, the *Tab Size* setting changes the spacing inside the editor for display purposes only. This spacing does not get saved to the Python file, as the indentation is always saved as a tab character. The auto-indent feature will will insert one tab for each indentation level.
- When set to *Spaces*, the *Tab Size* setting determines how many spaces will be used by the auto-indent feature for each indentation level. This spacing will get saved to the Python file, as the indentation is saved as multiple discrete space characters.

These settings should be chosen before starting a new Python script. If an existing Python script is opened, the editor will attempt to detect the indentation scheme that was previously used. If the settings are changed on an existing script (or inconsistencies are detected on opening), the editor will attempt to convert the tabs to spaces or vice versa. Note that Python 3 does not allow mixing the use of tabs and spaces for indentation.

Plot and Dataset Properties

Plots and datasets have various properties that determine how they are displayed in GT-POST (ex: axis labels, line types, legend position). The short names, types, and possible values for each property can be found in in the section **API Documentation > Plot and Dataset Properties**.

Python Environments

The GT-SUITE installation comes with a complete, standard version of Python, which can be found in the **<GTIHOME>/<VERSION>/GTsuite/python##/** folder. By default, Python scripts that are executed within GT-SUITE use this Python installation. However, other Python environments can also be utilized, as described below.

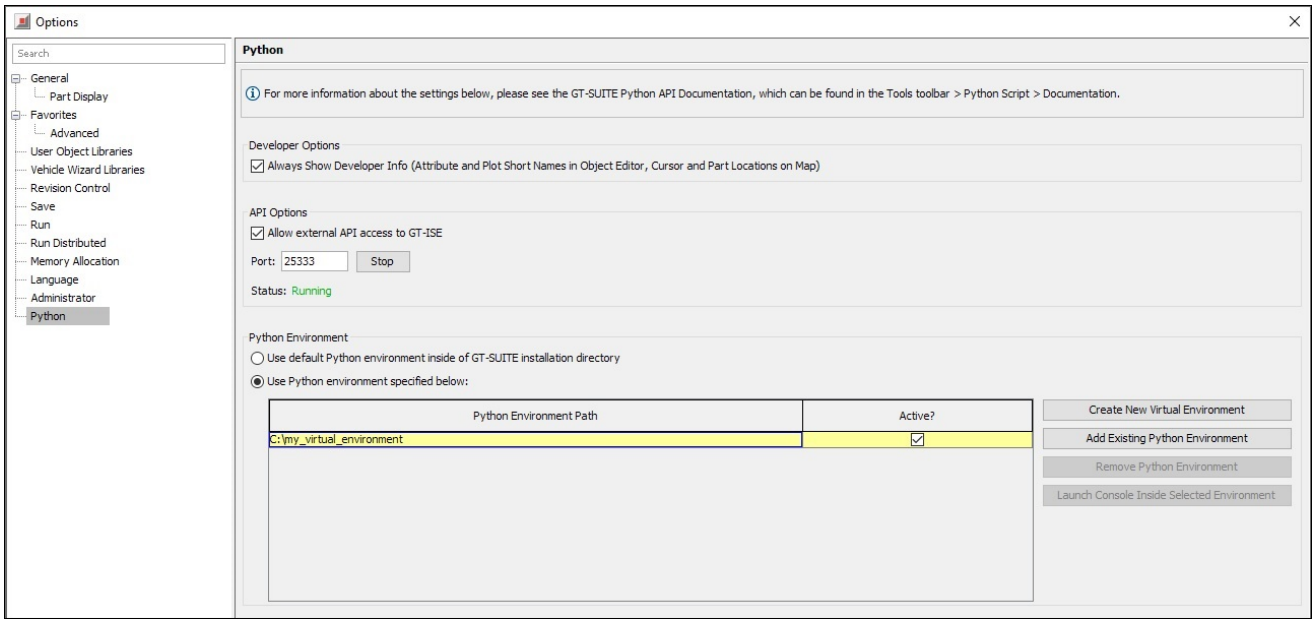
Warning

Certain elements of GT-SUITE rely upon the included Python installation to operate. Because of this, **it is strongly recommended that the base installation of Python (and included packages) not be modified.** If additional modules are required, they should be installed to a separate Python environment using one of the methods below.

Python Virtual Environments

A common method of creating multiple workspaces within a Python installation is to use virtual environments. Each virtual environment has its own copy of the original Python binary, and can have its own independent set of installed Python packages. This allows users to customize each workspace for the task at hand without risk of affecting work in other Python environments. Since virtual environments can also be located in any directory, this can also be helpful in situations where the GT-SUITE installation is placed somewhere that a user does not have adequate permissions to modify.

Virtual environments can be created using the native *venv* Python module. GT-SUITE includes some basic tools to make creating and managing these virtual environments more convenient. These tools can be found inside GT-ISE or GT-POST > File > Options > Python:



Creating A New Virtual Environment

Typically, a virtual environment is created via command line using the *venv* command. Running this command copies (or, on Linux, creates symbolic links to) the necessary files in the target directory. The virtual environment can be placed in any directory, and does not necessarily need to be inside the GT-SUITE installation directory or a Python installation directory.

```
>>> python3 -m venv /path_to_new_virtual_environment
```

Inside the Options dialog, the **Create New Virtual Environment** tool can be used to run this command automatically. Clicking this button opens a prompt where the target directory can be specified, after which the necessary files are copied, and the directory is added to the table in the Options dialog.

Adding An Existing Python Environment

If a virtual environment already exists, the **Add Existing Python Environment** button can be clicked to simply add the directory to the table in the Options dialog. No files are copied in this case.

This option can also be used to point to a standalone Python installation that is not a virtual environment. This can be useful, for example, to allow GT-SUITE to utilize a Python interpreter that is part of a 3rd party IDE. In this case, the directory that contains the Python executable should be specified. Additional setup may be required; for more information, please see the section **Getting Started > External API Access**.

Selecting A Python Environment For Use

Multiple Python environments can be added to the Options dialog in GT-ISE/GT-POST. To select a given environment for use, the 'Active?' checkbox should be clicked for the desired entry in the table. The environment selected here will be used globally for all GT-ISE and GT-POST GUI features that utilize Python, such as the Python Script Editor, automatic execution of Python scripts before simulations, and user shortcuts.

> **Note**
>
> This setting currently does **not** apply to the GT-SUITE solver. The solver always uses the base Python environment inside the GT-SUITE installation.

Installing Packages To A Virtual Environment

In order to install packages to a virtual environment, it must first be activated in a command prompt window. Activating temporarily adds the *python* and *pip* executables inside the virtual environment to *PATH*. On Windows, this can be done using the 'activate.bat' script found inside the virtual environment:

```
> my_env\Scripts\activate.bat
(my_env) >
```

As a shortcut, the **Launch Console Inside Selected Environment** tool can be used from the Options dialog once the desired Python environment is selected in the table. This will automatically launch a command prompt window and run the 'activate.bat' script corresponding to that virtual environment.

Note that this tool is not supported on Linux. However, the virtual environment can be activated similarly via command line using the equivalent 'activate' script for Linux:

```
> source my_env/bin/activate
(my_env) >
```

Notice that the command window is returned with the name of the virtual environment in parenthesis to denote the activated state. From here, standard *pip* commands can be used to install packages to the virtual environment.

```
(my_env) > pip install <package>
```

After the desired packages are installed, the *deactivate* command can be used, or the command prompt can simply be closed.

```
> deactivate
```

Removing A Virtual Environment

The **Remove Python Environment** button can be used in the Options dialog to remove the selected Python environment from the table. This does **not** delete the directory or any files from the disk. The virtual environment files may be kept for future use, or can simply be deleted manually.

External Python Installations

In addition to virtual environments, GT-SUITE can point to a standalone, external Python installation, such as one that may be included as part of a 3rd party Python IDE. Such environments can be added in the Options dialog using the **Add Existing Python Environment** button, as mentioned above. In this case, the directory that contains the Python executable should be specified. Additional setup may be required; for more information, please see the section **Getting Started > External API Access**.

External API Access

The GT Python API can be accessed from an external installation of Python. This can be helpful for users who prefer to utilize a dedicated Python IDE, or for integration of GT-SUITE into larger scripted workflows.

Python Installation

The GT Python API is based on the standard implementation of Python 3. Most variants of **Python 3.7 or newer** should be compatible with the API. Please visit www.python.org for more information and downloads of the latest version.

API Documentation

When using the GT Python API from outside of the GT GUI applications, these API documentation pages can be found in the GT installation directory under **/documents/Graphical_Applications/GT-Automation_Python_API/GT-Automation_Python_API.html**. This file can be opened in any standard web browser, and navigated the same way as in the built-in Python editor.

Py4J

Py4J (**Py**thon for **J**ava) is a library written in Python and Java designed to act as a bridge between the two programming languages. Since GT-SUITE applications are written in Java, this library is required for a Python interpreter to access Java objects inside GT-SUITE and interact through the API. More information about Py4J can be found at www.py4j.org.

The easiest method for installing the Py4J library is through pip (package manager for Python) using the command:

```
>>> pip install py4j
```

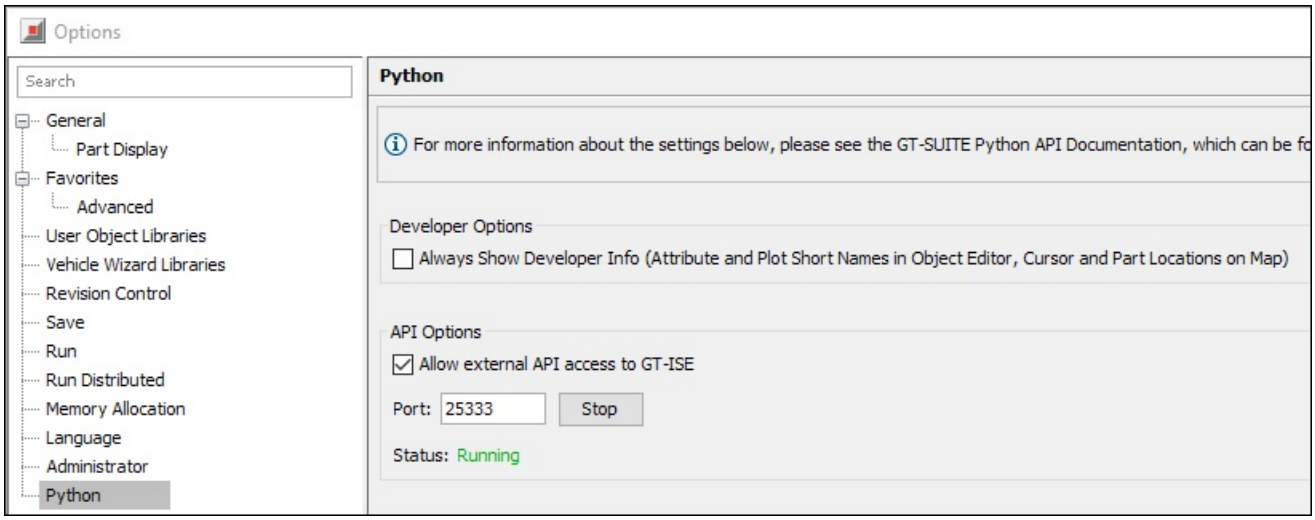
For more detailed instructions and alternative installation methods, please visit the Py4J Installation page.

> **Note**
>
> It is **not** necessary to install a Java environment to access the GT Python API, as this is built into the GT-SUITE installation.

Start API Gateway (GUI)

The GT Python API can be used with both an open GT-ISE/GT-POST GUI as well as with all GT GUI applications closed (headless). In both cases, an API gateway must be enabled and started in order to access it from an external Python interpreter.

The API gateway can be started within GT-ISE and GT-POST via File > Options > Python > Allow external API access to GT-ISE. The default ports (port:25333 for GT-ISE and port:25444 for GT-POST) should be acceptable in most cases, but can be changed if a conflict occurs with another application.



Once an external Python interpreter is connected to the API gateway, any commands sent through the API will be executed in the open instance of GT-ISE/GT-POST.

Start API Gateway (Headless)

The GT Python API can be used without any GT GUI applications running by using a headless instance of the API gateway. To *manually* start a headless gateway for GT-ISE, the following command can be used:

```
>>> gtisec -gateway
GATEWAY_PORT=12345
```

Similarly, to *manually* start a headless gateway for GT-POST, the following command can be used:

```
>>> gtpostc -gateway
GATEWAY_PORT=23456
```

Optional Arguments:

- *-port #####*: Determines the port number to be used for the gateway. If omitted, an ephemeral port will automatically be selected.
- *-v20##*: Specifies the version of GT-SUITE to be used to create the headless instance. If omitted, the default version specified by the *VERSIONS.TXT* file in the *GTIHOME* directory will be used.
- *-persistent*: The headless instance and gateway will continue to run in the background after the command window is closed. To shut down the gateway, an *app.close(force=True)* command must be issued from within the Python API.

> **Note**
>
> This step is **optional**, as external calls to the API via the *Application.get_instance()* command will automatically generate a new headless instance of the API gateway unless otherwise specified. However, it may be advantageous to do so if the API will be accessed multiple times in succession, as this will reduce the overhead associated with starting a new instance of the gateway each time.

External Script Initialization

In order to use the GT Python API, the *Application* module must first be imported. This can be done using the following lines of code:

```
import os
import sys

gtihome = os.environ['GTIHOME']
sys.path.append(os.path.abspath(gtihome + '/v2022/GTsuite/ext/gtsuite.jar')) #path should
be adjusted for desired version

from gtisoft.core.application import Application
```

Next, the following line can be used to attach the Python interpreter to an existing instance of the API gateway, or start a new one automatically:

```
app = Application.get_instance()
```

Similar to starting the API gateway from command line, the *get_instance()* command has optional arguments:

- *port=#####*: If the *port* argument is specified, the API will first attempt to connect to an existing instance of the gateway (GUI or headless) on that port. If no instance exists or the API fails to connect, a new headless instance of the gateway will be started automatically using the specified port.

If *port=None* or the *port* argument is omitted altogether, the API will start a new headless instance of the gateway using an ephemeral port.

This argument is ignored when running scripts from inside the built-in editor, as the script is necessarily attached to the GUI instance in which it is being run in that case.

- *version='-v20##'*: If a new headless instance of the gateway is started as a result of this call and the version is specified, the new instance will be started with that version. If no version is specified, the default version specified by the *VERSIONS.TXT* file in the *GTIHOME* directory will be used.

  This argument is ignored when running scripts from the built-in editor, or if a successful connection was made to an existing gateway via the *port* parameter, as the version is already implied/determined in these cases.
- *application_type='gtise' or 'gtpost'*: If a new headless instance of the gateway is started as a result of this call, this argument determines the GT application (GT-ISE or GT-POST) that will be used. If no application is specified, a headless instance of the GT-ISE gateway will be started by default.

  This argument is ignored when running scripts from the built-in editor, or if a successful connection was made to an existing gateway via the *port* parameter, as the application type is already implied/determined in these cases.

Note

A new application instance is created each time the *get_instance()* command is called. In order to access objects within the same application instance, it should be saved to a variable that can be referred to later in the script (such as *'app'* in the above example).

This is particularly important if the *get_instance()* call is starting a new headless API gateway, as a new gateway will be started each time this command is called.

Additionally, the following lines of code may be added for more convenient access to *UNITS* and *UNIT_CATEGORIES*.

```
UNITS = app.gtt.units
UNIT_CATEGORIES = app.gtt.unit_categories
```

Since the *units* and *unit_categories* mappings can always be accessed directly, this step is not required, but can be helpful for scripts where units are accessed frequently. This initialization occurs automatically when running Python scripts via the built-in editor but not when utilizing the API externally, so it may also be desirable for scripts that are designed to run both inside the built-in Python editor as well as externally.

Licensing

The GT Python API is part of the GT-Automation add-on license package, and requires the GTAutomation license in order to be accessed (both from inside the GT GUI applications, i.e. GT-ISE, as well as externally). This license is checked out during execution of Python code and is checked back in upon completion. No license is required to open the Python Script Editor inside the GT GUI Applications and write Python code or view the API documentation.

Please contact support@gtisoft.com for more information.

Application

The *Application* module represents the top level of the GT-ISE application.

Module Documentation

class **gtisoft.core.application.Application**(*application_wrapper=None, gateway: Optional[py4j.java_gateway.JavaGateway] = None, stdout_deque: Optional[Deque] = None, stderr_deque: Optional[Deque] = None*)

Bases: `abc.ABC`

The Application class represents an instance of the GT-ISE application.

close(*force: bool = False*) → None

Requests that the connected application shut itself down and then closes the *Application* object's connection.

Parameters

force (`bool`) – By default, GUI instances and headless instances started with the *-persistent* argument will ignore calls to *close* and remain open. If the *force* flag is *True* the call to *close* will not be ignored and the application will exit.

static **close_all**(*force: bool = False*) → None

Calls the `gtisoft.core.application.Application.close()` method on all connected *Application* instances.

Parameters

force (`bool`) – The *force* flag to pass when calling *close*.

static **get_instance**(*port: Union[str, int, None] = None, version: Optional[str] = None, application_type: Union[<unknown>.ApplicationType, str] = <ApplicationType.GTISE: 'gtise'>*) → gtisoft.core.application.Application

Gets an instance of an *Application* object that is connected to a running GT application.

Parameters

- **port** (`int`) – If the *port* argument is specified, the API will first attempt to connect to an existing instance of a GT application that is listening on that port (either a GUI instance or a headless version started via command line e.g. *gtisec*

-gateway -port xyz). If no instance exists or if the API fails to connect, a new headless instance of the GT application will be started, listening on the given port.

If the *port* argument is *None* or not specified, the API will start a new headless instance of the GT application using an ephemeral port.

This argument is ignored when running scripts from the editor inside of an application.

- **version** (`str`) – If a new headless instance of the GT application is started as a result of this call, and the version is specified, the new instance will be started with the given version. The format is *-v<version>*, e.g. *-v2020*. If no version is specified, the headless instance will open using the version specified by the *VERSIONS.TXT* file in the *GTIHOME* directory.

  This argument is ignored when running scripts from the editor inside of an application, or if a successful connection was made to an existing application listening on the port specified by the *port* parameter.

- **application_type** (`ApplicationType`) – If a new headless instance is started as a result of this call, the application that is started will be the one with the given type (for example GT-ISE or GT-POST).

Returns

The Application object for the connected GT application.

Return type

`Application`

abstract property **application_type**

property **documents**

`gtisoft.core.documents.Documents` : **Read-Only**; The *Documents* instance for opening and accessing documents in this application.

property **gtt**

`gtisoft.core.gtt.Gtt` : **Read-Only**; The *Gtt* instance for accessing the gtt database. This database provides static information (such as templates and units) for the Application.

property **stderr**

`collections.deque` : **Read-Only**; A deque containing the stderr output from the connected application. May be *None* if the connected application was not started by the script.

property **stdout**

`collections.deque`: **Read-Only**; A deque containing the stdout output from the connected application. May be *None* if the connected application was not started by the script.

property **version**

`GTVersion` **Read-Only**; The *GTVersion* instance that contains the version information for this application.

---

class **gtisoft.core.application.GtiseApplication**(*application_wrapper=None, gateway=None, stdout_deque: Optional[Deque] = None, stderr_deque: Optional[Deque] = None*)

Bases: `gtisoft.core.application.Application`

property **application_type**

property **distributed_queue**

`gtisoft.core.distributed.DistributedQueue`: **Read-Only**; The *DistributedQueue* instance for submitting, fetching, and otherwise interacting with distributed simulations.

property **simulations**

`gtisoft.core.simulations.SimulationManager`: **Read-Only**; The *SimulationManager* instance for starting and otherwise interacting with local simulations for this application.

---

class **gtisoft.core.application.GtpostApplication**(*application_wrapper=None, gateway=None, stdout_deque: Optional[Deque] = None, stderr_deque: Optional[Deque] = None*)

Bases: `gtisoft.core.application.Application`

property **application_type**

---

class **gtisoft.core.application.ExecutionLayerApplication**(*application_wrapper=None, gateway=None, stdout_deque: Optional[Deque] = None, stderr_deque: Optional[Deque] = None*)

Bases: `gtisoft.core.application.Application`

property **application_type**

---

class **gtisoft.core.application.ApplicationType**

Bases: `gtisoft.util.collect_util.EnumMixin`, `enum.Enum`

An enumeration.

get_executable_name()

CENTRALWEBSERVER= *'centralwebserver'*

ELAPP= *'elapp'*

GEM= *'gem'*

GTISE= *'gtise'*

GTPOST= *'gtpost'*

IDO= *'ido'*

---

class **gtisoft.core.application.GtCentralWebServerApplication**(*application_wrapper=None, gateway=None, stdout_deque: Optional[Deque] = None, stderr_deque: Optional[Deque] = None*)

Bases: `gtisoft.core.application.Application`

property **application_type**

Attributes

The *Attribute* module contains functions and classes for modifying the values of attributes in GT-objects and parts.

Module Documentation

exception **gtisoft.core.attributes.SetValueError**(*message*)

Bases: `Exception`

Raised when there is an error attempting to set the value of an *AttributeValue*.

Variables

message (`str`) – A message describing the error.

Parameters

message (`str` , optional) – A message describing the error.

class **gtisoft.core.attributes.Attributes**(*attribute_provider_wrapper*)

Bases: `collections.abc.Mapping`

This class is a collection that belongs to an `gtisoft.core.objects.Object` or `gtisoft.core.parts.Part` and contains the attribute values or part overrides for its parent.

Notes

Attributes is a `Mapping` that behaves similarly to a Python `dict` . The keys to the mapping are an attribute short name (type `str`) and the values are instances of *AttributeValue*.

Many of the `dict` methods are present in the *Attributes* class as well as many of the same capabilities.

For instance, checking for the presence of an attribute in an object or part can be done using the Python membership operators:

```
>>> # Check if the 'length' attribute is in a pipe object
>>> print('PIPEL' in pipe_object.attribute_values)
True
```

```
>>> print('MISSING' not in pipe_object.attribute_values)
True
```

Attributes is “scriptable”, enabling getting attributes from an object or part using brackets:

```
>>> att = pipe_object.attribute_values['PIPEL']
>>> print(att)
AttributeValue(short_name=u'PIPEL', long_name=u'Length', value=[100], unit=mm)
```

Attributes also allows iterating over all the attributes in an object or part using the default iterator:

```
>>> for attribute_name in pipe_object.attribute_values:
>>>         print(attribute_name)
D1
D2
PIPEL
...
```

See also

Documentation for Python `dict` type.

copy_values(*other_attributes*)

Copies all:

```
# Copy all values from PipeRound-1 to PipeRound-2
obj_A = objects.get_object('PipeRound', 'PipeRound-1')
obj_B = objects.get_object('PipeRound', 'PipeRound-2')
obj_B.attribute_values.copy_values(obj_A.attribute_values)
```

or a subset:

```
# Copy diameter values from PipeRound-1 to PipeRound-3
obj_C = objects.get_object('PipeRound', 'PipeRound-3')
obj_C.attribute_values.copy_values(
    {'D1' : obj_A.attribute_values['D1'], 'D2' : obj_A.attribute_values['D2']})
```

of the values from another *Attributes* collection to this collection.

Parameters

other_attributes (`Attributes`) – The other set of attributes to copy.

Returns

Return type

None

Raises

SetValueError – If an error occurs while copying the values.

items()

Gets a list of key-value pairs for all the attributes in this collection where the key is an attribute short name as a *str* and the value is an instance of *Attribute*.

Returns

A list of key-value pairs where the key is a *str* attribute short name and the value is an *Attribute*.

Return type

`list` of (`str`, `Attribute`)

keys()

Gets a new view of the keys for all the attributes in this collection. A key is an attribute short name as a *str*.

Returns

A list of keys for all the attributes in this collection.

Return type

`list` of `str`

values()

Gets a list of *Attribute* instances for all the attributes in this collection.

Returns

A list of all the attributes in this collection.

Return type

`list` of `Attribute`

---

class **gtisoft.core.attributes.AttributeValue**(*value_wrapper*)

Bases: `collections.abc.Sequence`

This class is a collection of zero or more values for an *Attribute*.

Notes

AttributeValue is a `Sequence` that behaves similarly to a Python `list`. When treating this collection like a *Sequence*, values are represented as a single dimensional vector, even for matrix and 3D data.

Much of the behavior that is present in *list* is also present in *AttributeValue*. For instance checking the number of values:

```
>>> value = obj.attribute_values['ARRAY']
>>> print(value)
AttributeValue(short_name=u'USER1', long_name=u'X Data', value=[1.0, 2.0, 3.0], unit=No
Unit)
>>> print(len(value))
3
```

AttributeValue is also "scriptable", enabling getting a value at a specific index using brackets:

```
>>> print(value[0])
1.0
>>> print(value[-1])
3.0
```

and it supports "slicing":

```
>>> print(value[0:2])
[1.0, 2.0]
```

For working with matrix and 3D data in two dimensions, the `number_rows()`, `number_columns()`, `get_value()`, `get_row()`, and `get_column()` methods are provided.

get_column(*column*)

Gets a new view of a single column of the value as a *list*.

For example:

```
>>> values = [
>>>     [1.0, 2.0, 3.0],
>>>     [4.0, 5.0, 6.0],
>>>     [7.0, 8.0, 9.0]
>>> ]
>>> obj.attributes_values['ZDAT'].set_value(values, None)
>>> print(obj.attribute_values['ZDAT'].get_column(1))
[2.0, 5.0, 8.0]
```

Parameters

column (`int`) – the index of the column to get a view of

Returns

A new view of the values in the given column.

Return type

`list`

Raises

exceptions.IndexError – If the given column index is out of bounds.

get_row(*row*)

Gets a new view of a single row of the value as a *list*.

For example:

```
>>> values = [
>>>     [1.0, 2.0, 3.0],
>>>     [4.0, 5.0, 6.0],
>>>     [7.0, 8.0, 9.0]
>>> ]
>>> obj.attributes_values['ZDAT'].set_value(values, None)
>>> print(obj.attribute_values['ZDAT'].get_row(1))
[4.0, 5.0, 6.0]
```

Parameters

row (`int`) – the index of the row to get a view of

Returns

A new view of the values in the given row.

Return type

`list`

Raises

exceptions.IndexError – If the given row index is out of bounds.

get_value(*row=0*, *column=0*)

Gets the value for a specific row and column index.

Parameters

- **row** (`int` , optional) – the row index
- **column** (`int` , optional) – the column index

Returns

The value at the given row and column indices.

Return type

`str` or `float`

Raises

exceptions.IndexError – If the given row or column index is out of bounds.

set_solver_unit(*unit*, *category=None*)

Sets the user-defined unit and category for the attribute, if it is supported by this attribute. The value of the *solver_unit* and *unit* properties will both be set to the new unit. Will not convert or change any values.

Parameters

- **unit** (`Unit` or `list` of `Unit`) – The new unit to use; may be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.
- **category** (`UnitCategory` , optional) – The category for the unit to set. This is required to disambiguate the *unit* parameter in the case it is a *list*.

Raises

SetValueError – If the attribute does not support user-defined units, if the *unit* parameter is a *list* but the *category* parameter is not provided, or if the solver unit cannot be set for any other reason.

set_unit(*unit*)

Sets the unit to use for the value or values of this attribute. The values will be automatically converted from the previous unit the same as if the dropdown were changed in the object editor in the UI.

Parameters

unit (`Unit` or `list` of `Unit`) – The new unit to use; may be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.

Raises

SetValueError – If the unit does not exist or is not valid for the attribute.

set_value(*new_values*, *unit=None*)

Sets the object value for the attribute.

Parameters

- **new_values** (`list` of `str` , `double` , or `int`) – A list of the new value(s) for the attribute.

  Setting values always requires a list because all values are treated as vectors. Single value attributes accept a list of length one:

```
attr.set_value([10.0], UNITS['mm'])
```

Array type values accept a list of one or more values:

```
attr.set_value([1.0, 2.0, 3.0, 4.0], UNITS['deg'])
```

Matrix and 3D type attributes will accept a list of lists:

```
values = [
    [1.0, 2.0, 3.0],
    [4.0, 5.0, 6.0],
    [7.0, 8.0, 9.0]
]
attr.set_value(values, UNITS['N'])
```

- **unit** (`gtisoft.core.units.Unit` , or `list` of `gtisoft.core.units.Unit` , optional) – The unit to use for the new values.

  Defaults to the attribute's current unit if *None* or if the parameter is omitted.

  May be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.

  For example: there are multiple units with the name 'rad':

  ```
  >>> rads = UNITS['rad']
  >>> print(rads)
  [Unit(gtt_id=62, name='rad', unit_category='ANGLE',
  multiplier=57.2957795130823),                Unit(gtt_id=525, name='rad',
  unit_category='ANGLE 3D', multiplier=57.2957795130823)]
  ```

  Instead of requiring selection of the correct unit from the list returned by the *UNITS* dictionary, it can be used directly and GT-ISE will choose the correct unit for the attribute based on the category.

Returns

Return type

None

Raises

SetValueError – If an error occurred when setting the value.

property **is_checkbox_on**

`bool` :

Gets or sets the status of the attribute's checkbox. If the attribute does not have a checkbox, the value of the property is *False* and attempting to set the value will raise a *SetValueError*.

Note that the checkbox referenced here is the so-called 'left checkbox' that activates or deactivates an attribute, not the so-called 'right checkbox' where the selection status is the value of the attribute itself. To set the value of a 'right checkbox' use *set_value* with a value of *['on']* or *['off']*.

> **Raises**
>
> **SetValueError** – If the attribute does not have a check box or if the value cannot be set for any reason.

property **`is_radio_on`**

> `bool`:
>
> Gets or sets the status of the attribute's radio button. If the attribute does not have a radio button, the value of the property is *False* and attempting to set the value will raise a *SetValueError*.
>
> For attributes with a radio button, setting it to *True* will turn off any other radio buttons in the group.

> **Raises**
>
> **SetValueError** – If the attribute does not have a radio button or if the value cannot be set for any reason.

property **`long_name`**

`str`: **Read-Only**; The attribute's long name, which is longer and more descriptive than the short name; the long name is not required to be unique within a template.

property **`number_columns`**

`int`: **Read-Only**; Gets the number of columns for the value. For single value attributes, this will always be *1*. For array type data, this will be the number of values in the array, and for matrix and 3D data it is the number of columns in the matrix.

property **`number_rows`**

`int`: **Read-Only**; Gets the number of rows for the value. For data that is not matrix or 3D, this will always be *1*.

property **`short_name`**

`str`: **Read-Only**; The attribute's short name, which uniquely identifies this attribute within its parent object's or part's template.

property **solver_unit**

`gtisoft.core.units.Unit`:

Read-Only; Gets the unit that is used for writing values to solver for the attribute.

Notes

Certain attributes support a user-defined unit and category. For instance the *XYTable* template allows the user to specify both the unit and unit category for the *X Data* and *Y Data* attributes. For such attributes this property may be changed via the *set_solver_unit* method.

property **unit**

`gtisoft.core.units.Unit`: **Read-Only**; Gets the unit for the value.

Case Setup

The *Case Setup* module contains functions and classes for creating, editing, and deleting cases, folders, and parameters in the Case Setup of a model.

Module Documentation

class **gtisoft.core.case_setup.CaseSetup**(*document_wrapper*)

Bases: `object`

This class contains the collections for run cases, parameter folders, and parameters that allow querying and interacting with its model's case setup.

> *property* **parameter_folders**
>
> `ParameterFolders`: **Read-Only**; Returns the collection for querying and modifying parameter folders.

> *property* **parameters**
>
> `Parameters`: **Read-Only**; Returns the collection for querying and modifying parameters.

> *property* **run_cases**
>
> `RunCases`: **Read-Only**; Returns the collection for querying and modifying run cases.

class **gtisoft.core.case_setup.RunCases**(*case_provider_wrapper*)

Bases: `collections.abc.Sequence`

This class is a collection of one or more cases from Case Setup.

Notes

RunCases is a `Sequence` that behaves similarly to a Python `list`.

Much of the behavior that is present in *list* is also present in *RunCases*. For instance checking the number of cases:

```
>>> print(len(run_cases))
3
```

RunCases is also "scriptable", enabling getting the case at a specific index using brackets:

```
>>> print(run_cases[0])
RunCase(case_index=0, label=u'Case 1', is_on=True)
>>> print(run_cases[-1])
RunCase(case_index=5, label=u'Case 6', is_on=True)
```

and it supports “slicing”:

```
>>> print(run_cases[0:2])
[RunCase(case_index=0, label=u'Case 1', is_on=True), RunCase(case_index=1, label=u'Case 2',
is_on=True)]
```

`append_case`()

Appends a new case to the end of the sequence of cases.

Returns

The run case that was created.

Return type

`RunCase`

Raises

exceptions.RunTimeError – If a case could not be appended for any reason.

`delete_case`(*run_case*)

Deletes the case represented by the given `RunCase` instance or given `int` index.

Parameters

run_case (`RunCase` or `int`) – An instance of *RunCase* that represents the case to be deleted, or else the integer index of the case to delete.

Raises

- **exceptions.IndexError** – If the given integer index is out of bounds, that is, it is *< 0* or *> len(self)*
- **exceptions.ValueError** – If the given case cannot be deleted for any reason.

`insert_case`(*index*)

Inserts a new case at the specified index. Indices start at *0* and increment by one for each case. If a case already exists at the given index, the new case will be inserted and the existing case will be moved to *index+1*.

Parameters

index (`int`) – The position where the new case should be inserted.

Returns

The run case that was created.

Return type

`RunCase`

Raises

- **exceptions.IndexError** – If the given index is out of bounds, that is, it is < *0* or > *len(self)*
- **exceptions.RunTimeError** – If a case could not be appended for any other reason.

---

class **gtisoft.core.case_setup.RunCase**(*case_wrapper*)

Bases: `object`

A *RunCase* instance represents and provides methods for interacting with a single run case from the case setup of a model.

property **case_index**

`int`: **Read-Only**; Gets the 0-based index of this case.

property **is_on**

`bool`:

Gets or sets the on/off status of this parameter.

Raises

exceptions.ValueError – When setting the on/off status, if it cannot be set for any reason.

property **label**

`str`:

Gets or sets the case label for this parameter.

Raises

exceptions.ValueError – When setting the label, if the label cannot be created for any reason.

---

class **gtisoft.core.case_setup.Parameters**(*parameter_provider_wrapper*)

Bases: `collections.abc.Mapping`

This class is a collection that contains all of the parameters in a model's case setup.

Notes

Parameters is a `Mapping` that behaves similarly to a Python `dict`. The keys to the mapping are a parameter's name (type `str`) and the values are instances of *Parameter* that represent parameters in case setup.

Many of the `dict` methods are present in the *Parameters* class as well as many of the same capabilities.

For instance, checking for the presence of a parameter can be done using the Python membership operators:

```
>>> print('RPM' in parameters)
True
```

```
>>> print('Missing Parameter' not in parameters)
True
```

Parameters is "scriptable", enabling getting parameters from case setup using brackets:

```
rpm_param = parameters['RPM']
```

Parameters also allows iterating over all parameters using the default iterator:

```
for param_name in parameters:
    print(param_name)
```

See also

Documentation for Python `dict` type.

`create_parameter`(*parameter_name*, *description*, *folder*, *unit*, *unit_category=None*)

Creates a new *Parameter* and adds it to this collection (and the document that owns it). The created parameter will be a 'Declared Parameter' in Case Setup.

If a parameter with the given name already exists, it will be converted to a 'Declared Parameter'.

Parameters

- **parameter_name** (`str`) – The name of the parameter to create; or the name of an existing parameter to set as 'Declared'.
- **description** (`str`) – A description of the parameter that will be created.

- **folder** (`str` or `gtisoft.core.parameter_folders.ParameterFolder`) – The folder where the new parameter should be added. May be either an instance of *ParameterFolder* or the name of a folder as a *str*.
- **unit** (`Unit` or `list` of `Unit`) – The unit for the new parameter; this will be used as both the 'value' unit that is displayed in the dropdown and the 'solver' unit that is written during .dat file creation. May be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.
- **unit_category** (`UnitCategory` , optional) – The category for the unit. This is required to disambiguate the *unit* parameter in the case it is a *list*.

Returns

The newly created *Parameter*.

Return type

`Parameter`

Raises

RunTimeError – If the parameter could not be created for any reason.

`delete_parameter`(*parameter*)

Deletes a parameter from this collection (and the document that owns it).

Note that deleting a parameter is only allowed for unused 'Declared' parameters.

Parameters

parameter (`Parameter` or `str`) – An instance of 'Parameter' or the name of the parameter to delete.

Raises

- **exceptions.KeyError** – If the given *Parameter* is not in this collection, or if a parameter with the given name does not exist.
- **exceptions.RuntimeError** – If the parameter cannot be deleted because it is not declared, is not unused, or for any other reason.

`items`()

Gets a list of key-value pairs for all the parameters in this collection where the key is a parameter name as a *str* and the value is an instance of *Parameter*.

Returns

A list of key-value pairs where the key is a *str* and the value is a *Parameter*

Return type

`list` of (`str`, `Parameter`)

keys()

Gets a new view of the keys for all the parameters in this collection. A key is a parameter name as a *str*.

Returns

A list of keys for all the parameters in this collection

Return type

`list` of `str`

values()

Gets a list of *Parameter* instances for all the parameters in this collection.

Returns

A list of all the parameters in this collection.

Return type

`list` of `Parameter`

---

class **gtisoft.core.case_setup.Parameter**(*parameter_wrapper*)

Bases: `collections.abc.Sequence`

A *Parameter* instance represents and provides methods for interacting with a parameter from the case setup of a model.

Notes

Parameter is a `Sequence` that behaves similarly to a Python `list`. When treating this collection like a *Sequence*, the parameter's values are represented as a single dimensional vector.

Much of the behavior that is present in *list* is also present in *Parameter*. For instance checking the number of values:

```
>>> parameter = parameters['RPM']
>>> print(parameter[0])
6000
>>> print(len(parameter))
11
```

Parameter is also "scriptable", enabling getting a value at a specific index using brackets:

```
>>> print(parameter[0])
6000
>>> print(parameter[-1])
1000
```

and it supports “slicing”:

```
>>> print(parameter[0:2])
[6000, 5500]
```

Setting the value for a case can also be done using brackets:

```
>>> parameter[0] = 6500
>>> print(parameter[0])
6500
```

get_value(*case_or_option*)

Gets the value for the given case or super parameter option.

Parameters

case_or_option (`RunCase` , `int` , `SuperParameterOption` , or `str`) – For regular parameters, an instance of *RunCase* or an *int* 0-based case index to specify which case to get the value for. For parameters in super parameter folders, an *int* 0-based index for the option, a *SuperParameterOption* or a *str* name of the option to retrieve the value for.

Returns

The value of the parameter for the given case or option as a *str*.

Return type

`str`

Raises

- **exceptions.IndexError** – If *case_or_option* is an *int* that is out of bounds for the values sequence.
- **exceptions.TypeError** – If *case_or_option* is a *str* but the parameter is not inside a super parameter folder OR if *case_or_option* is an instance of *RunCase* and the parameter is inside a super parameter folder.
- **exceptions.KeyError** – If *case_or_option* is a *str* but no super parameter choice with the given name exists for the super parameter folder that the parameter is in.
- **exceptions.ValueError** – If the value cannot be retrieved for any other reason.

set_solver_unit(*unit*, *category=None*)

Sets the user-defined unit and category for this parameter, if it is supported by this parameter. The value of the *solver_unit* and *unit* properties will both be set to the new unit. Will not convert or change any values.

Parameters

- **unit** (`Unit` or `list` of `Unit`) – The new unit to use; may be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.
- **category** (`UnitCategory` , optional) – The category for the unit to set. This is required to disambiguate the *unit* parameter in the case it is a *list*.

Raises

exceptions.ValueError – If this parameter does not support user-defined units, if the *unit* argument is a *list* but the *category* argument is not provided, or if the solver unit cannot be set for any other reason.

set_unit(*unit*)

Sets the unit to use for the value or values of this parameter. The values will be automatically converted from the previous unit the same as if the dropdown were changed in the Case Setup dialog in the UI.

Parameters

unit (`Unit` or `list` of `Unit`) – The new unit to use; may be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.

Raises

exceptions.ValueError – If the unit does not exist or is not valid for this parameter.

set_value(*case_or_option*, *value*)

Sets the value for the given case or super parameter option.

Parameters

- **case_or_option** (`RunCase` , `int` , `SuperParameterOption` , or `str`) – For regular parameters, an instance of *RunCase* or an *int* 0-based case index to specify which case to set the value for. For parameters in super parameter folders, an *int* 0-based index for the option, a *SuperParameterOption* or a *str* name of the option to set the value for.
- **value** (`str`) – The new value to set for the case or option.

Raises

- **exceptions.IndexError** – If *case_or_option* is an *int* that is out of bounds for the values sequence.
- **exceptions.TypeError** – If *case_or_option* is a *str* but the parameter is not inside a super parameter folder OR if *case_or_option* is an instance of *RunCase* and the parameter is inside a super parameter folder.
- **exceptions.ValueError** – If *case_or_option* is a *str* but no super parameter choice with the given name exists for the super parameter folder that the parameter is in, or if the value cannot be set for any other reason.

property **description**

`str` : Gets or sets this parameter's description.

property **folder**

`gtisoft.core.parameter_folders.ParameterFolder` :

Gets or sets the parent folder for this parameter.

Raises

exceptions.ValueError – When setting the folder, if the given folder does not exist or the parameter cannot be moved for any reason.

property **is_super_parameter**

`bool` **Read-Only**; Returns *True* if the parameter is a super parameter, otherwise *False*.

property **parameter_name**

`str` :

Gets or sets the name of this parameter.

Raises

exceptions.ValueError – When setting the name, if the new name is invalid.

property **solver_unit**

`gtisoft.core.units.Unit` : **Read-Only**; Gets the unit that is used for writing values to the solver for this parameter.

property **super_parameter_options**

`SuperParameterOptions` **Read-Only**; Gets the *SuperParameterOptions* collection containing this parameter's options if it is a super parameter, otherwise *None*.

property **unit**

`gtisoft.core.units.Unit`: **Read-Only**; Gets the unit for this parameter.

---

class **gtisoft.core.case_setup.SuperParameterOptions**(*wrapper*)

Bases: `collections.abc.Sequence`

A `Sequence` of *SuperParameterOption* objects that represent the options available for a super parameter.

Options in the sequence appear in the same order as they would in the Case Setup dialog.

---

class **gtisoft.core.case_setup.SuperParameterOption**(*option_wrapper*)

Bases: `object`

An option that is available for a super parameter.

property **option_name**

`str` **Read-Only**; Gets the name of this super parameter option.

---

class **gtisoft.core.case_setup.ParameterFolders**(*parameter_folder_provider_wrapper*)

Bases: `collections.abc.Mapping`

This class is a collection that contains all of the folders in a model's case setup.

Notes

ParameterFolders is a `Mapping` that behaves similarly to a Python `dict`. The keys to the mapping are a folder's name (type `str`) and the values are instances of *ParameterFolder* that represent folders in case setup.

Many of the `dict` methods are present in the *ParameterFolders* class as well as many of the same capabilities.

For instance, checking for the presence of a folder can be done using the Python membership operators:

```
>>> print('Main' in parameter_folders)
True
```

```
>>> print('Missing Folder' not in parameter_folders)
True
```

ParameterFolders is "scriptable", enabling getting folders from case setup using brackets:

```
main_folder = folders['Main']
```

ParameterFolders also allows iterating over all folders using the default iterator:

```
for folder_name in parameter_folders:
    print(folder_name)

.. seealso:: Documentation for Python :class:`dict` type.
```

create_folder(*folder_name*)

Creates a new folder with the given name and adds it to this collection.

Parameters

folder_name (`str`) – The name of the folder to create.

Returns

The folder that was created.

Return type

`ParameterFolder`

Raises

exceptions.ValueError – If a folder with the given name already exists, if the name is not valid, or if the folder cannot be created for any other reason.

delete_folder(*folder*)

Deletes the given folder.

Parameters

folder (`ParameterFolder` or `str`) – The folder instance or the name of the folder to delete.

Raises

exceptions.ValueError – If the given Folder instance does not have a matching folder in case setup, if a folder with the given name does not exist, if the folder contains parameters, or if the folder cannot be deleted for any other reason.

items()

Gets a list of key-value pairs for all the parameter folders in this collection where the key is a parameter folder name as a *str* and the value is an instance of *ParameterFolder*.

Returns

A list of key-value pairs where the key is a *str* and the value is a *ParameterFolder*

Return type

`list` of (`str`, `ParameterFolder`)

keys()

Gets a new view of the keys for all the parameter folders in this collection. A key is a parameter folder name as a *str*.

Returns

A list of keys for all the parameter folders in this collection

Return type

`list` of `str`

values()

Gets a list of *ParameterFolder* instances for all the parameter folders in this collection.

Returns

A list of all the parameter folders in this collection.

Return type

`list` of `ParameterFolder`

---

class **gtisoft.core.case_setup.ParameterFolder**(*parameter_folder_wrapper*)

Bases: `object`

Represents and provides methods for interacting with a parameter folder in case setup.

property **is_super_parameter_folder**

`bool`: **Read-Only**; Returns *True* if the folder belongs to a super parameter, otherwise *False*.

property **parameter_folder_name**

`str`:

Gets or sets the name of the parameter folder.

Raises

exceptions.ValueError – When setting the name, if the new name is invalid.

property **parameters**

`list` of `gtisoft.core.parameters.Parameter`: **Read-Only**; Returns a *list* of *Parameter* instances for the parameters that are in this folder in case setup.

Distributed

The *Distributed* module contains methods and classes for interacting with distributed clusters to submit and fetch simulations.

Module Documentation

class **gtisoft.core.distributed.DistributedQueue**(*distributed_queue_wrapper*)

Bases: `object`

The *DistributedQueue* object provides methods for submitting, fetching, and otherwise interacting with distributed simulations.

check_results(*document*, ***kwargs*)

Checks if all packets of the distributed simulation have completed, meaning that the results are ready to be downloaded.

Parameters

- **document** (`gtisoft.core.documents.Document`) – The document that was submitted to the cluster.
- ****kwargs** – A set of keyword arguments specifying the options for checking the simulation results. See below.

Keyword Arguments

url (`str`) – Scheduler hostname and port number for 2nd generation distributed cluster. The form is: *url='gdc://hostname:portnumber'*. Use *'sgdc'* instead of *'gdc'* for encrypted communication. To submit to the local-distributed service, use hostname *'localhost'*. The default ports are 8970 (Plaintext) and 9970 (encrypted) for network distributed and 8972 for local-distributed. Ports are specified in GT-ISE > File > Options > Run Distributed.

Returns

True if the simulation is complete and results are ready to be fetched, otherwise *False*.

Return type

`bool`

Raises

exceptions.RuntimeError – If a job for the given document does not exist on the cluster or if the results could not be checked for any other reason.

delete_results(*document*, ***kwargs*)

Deletes the results for the given document from the queue without downloading any results. All results will be lost.

Parameters

- **document** (`gtisoft.core.documents.Document`) – The document that was submitted to the cluster.
- ****kwargs** – A set of keyword arguments specifying the options for deleting the simulation results. See below.

Keyword Arguments

url (`str`) – Scheduler hostname and port number for 2nd generation distributed cluster. The form is: *url='gdc://hostname:portnumber'*. Use *'sgdc'* instead of *'gdc'* for encrypted communication. To submit to the local-distributed service, use hostname *'localhost'*. The default ports are 8970 (Plaintext) and 9970 (encrypted) for network distributed and 8972 for local-distributed. Ports are specified in GT-ISE > File > Options > Run Distributed.

Raises

exceptions.RuntimeError – If the results could not be deleted for any reason.

fetch_results(*document*, ***kwargs*)

Fetches the results for the given document from the distributed cluster. This method will not return until the fetch is completed or encounters an error. If packets are still running, the command will wait to attempt a fetch until they are all in a final state.

Parameters

- **document** (`gtisoft.core.documents.Document`) – The document that was submitted to the cluster.
- ****kwargs** – A set of keyword arguments specifying the options for fetching the simulation results. See below.

Keyword Arguments

url (`str`) – Scheduler hostname and port number for 2nd generation distributed cluster. The form is: *url='gdc://hostname:portnumber'*. Use *'sgdc'* instead of *'gdc'* for encrypted communication. To submit to the local-distributed service, use hostname *'localhost'*. The default ports are 8970 (Plaintext) and 9970 (encrypted) for network distributed and 8972 for local-distributed. Ports are specified in GT-ISE > File > Options > Run Distributed.

Raises

exceptions.RuntimeError – If the results could not be fetched for any reason.

submit(*document*, ***kwargs*)

Submits a document to the distributed computing cluster.

Parameters

- **document** (`gtisoft.core.documents.Document`) – The document to submit for simulation.
- ****kwargs** – A set of keyword arguments specifying the options for submitting the simulation. See below.

Keyword Arguments

url (`str`) – Scheduler hostname and port number for 2nd generation distributed cluster. The form is: *url='gdc://hostname:portnumber'*. Use *'sgdc'* instead of *'gdc'* for encrypted communication. To submit to the local-distributed service, use hostname *'localhost'*. The default ports are 8970 (Plaintext) and 9970 (encrypted) for network distributed and 8972 for local-distributed. Ports are specified in GT-ISE > File > Options > Run Distributed.

Raises

- **gtisoft.core.documents.DocumentClosedError** – If the *document* argument is a *Document* instance that is not currently open.
- **exceptions.RuntimeError** – If the simulation could not be submitted for any other reason.

Documents

The *Documents* module contains functions and classes for opening, closing, and saving GT-ISE map documents (gtm, gtsub, gtc, etc).

Module Documentation

exception **gtisoft.core.documents.OpenDocumentError**(*message: Optional[str] = None*)

Bases: `Exception`

Raised when there is an error attempting to open a document.

Variables

message (`str`) – A message describing the error.

Parameters

message (`str` , optional) – A message describing the error.

exception **gtisoft.core.documents.SaveDocumentError**(*message: Optional[str] = None*)

Bases: `Exception`

Raised when there is an error attempting to save a document.

Variables

message (`str`) – A message describing the error.

Parameters

message (`str` , optional) – A message describing the error.

exception **gtisoft.core.documents.DocumentClosedError**(*message: Optional[str] = None*)

Bases: `Exception`

Raised when an operation is attempted on a document that is already been closed via `Document.close()` .

Variables

message (`str`) – A message describing the error.

Parameters

message (`str` , optional) – A message describing the error.

---

class **gtisoft.core.documents.Documents**(*document_manager_wrapper*)

Bases: `collections.abc.Mapping`

The *Documents* object is used for looking up open documents in the application.

Notes

Documents is a `Mapping` that behaves similar to a Python `dict` . The keys to the mapping are document paths and the values are `Document` objects.

Many of the `dict` methods are present as well as many of the same capabilities.

For instance, checking for the presence of a document may be done using the Python membership operators:

```
print('RunMeFirst.gtm' in documents)
print('MissingDocument.gtm' not in documents)
```

Documents is “scriptable”, enabling getting documents using brackets:

```
print(documents['RunMeFirst'])
```

Documents also allows iterating using the default iterator:

```
for name in documents:
    print(name)
```

One difference between a standard `dict` and a *Documents* object is that keys are much more flexible for *get* operations. A *get* operation will attempt to find a matching *Document* using (in order):

1. full path
2. a document name with extension
3. a document name without an extension

Only in the case that all these fail will it raise a `KeyError` .

See also

Documentation for Python `dict` type.

get_active_document() → Optional[gtisoft.core.documents.Document]

Gets the *Document* that is currently active in the application.

If the application has a UI, this is the document that is currently being edited.

If this script is being called from command line GT-ISE to create a .dat file or run a simulation as part of the "Advanced Setup -> Automation -> Run Script Before Simulation" options, the document will be the model that is currently being processed.

Returns

- `Document` – If the application has an active document
- *None* – If the application does not have an active document

get_document(*document_name: str*) → Optional[gtisoft.core.documents.Document]

Gets a *Document* with the given name or path.

Parameters

document_name (`str`) – The name (with or without extension) or full path of the document to get.

Returns

- `Document` – If an open document with the given name or full path was found.
- *None* – If no open document with the given name or full path could be found.

get_version(*file_path: str*) → gtisoft.core.version.GTVersion

Gets a document's version without opening it.

Parameters

file_path (`str`) – The full path of the file for the document.

Returns

- `GTVersion` – The document's version, if an document with the given full path was found.
- *None* – If no document full path could be found.

items() → List[Tuple[str, gtisoft.core.documents.Document]]

Gets a list of the *(name, document)* key-value pairs for all open documents.

Returns

A list of key-value pairs where the key is a document name and the value is a *Document*.

Return type

`list` of (`str` , `Document`)

keys() → List[str]

Gets a new view of the keys (document names) for all open documents.

Returns

A list of the names of open documents

Return type

`list` of `str`

new_document(*document_type: Union[<unknown>.GtDocumentType, str, None] = None*, *license_type: Union[<unknown>.GtLicenseType, str, None] = None*) → gtisoft.core.documents.Document

Creates a new document.

Parameters

- **document_type** (`GtDocumentType` , optional) – The type of the document to create. If *None*, GT-ISE will create a "GT-Model (.gtm)" document and GT-POST will create a "GT Report File (.gu)".
- **license_type** (`GtLicenseType`], optional) – The license to use for the new document. If *None*, GT-ISE will use the "GT-SUITE" license by default. GT-POST does not use this argument.

Returns

The newly created *Document*.

Return type

`Document`

open_document(*file_path: str*, *evolution_option: Optional[str] = None*, *rename_evolved: bool = True*, *evolve_dependencies: bool = True*, *rename_dict: Optional[Dict[str, str]] = None*) → gtisoft.core.documents.Document

Opens the document with the given file name.

Parameters

- **file_path** (`str`) – The full path of the file for the document.
- **evolution_option** (`str` , optional) – Specifies the solution method to use after evolution. Valid values are:
  - `'new'` - Minor bug fixes. Similar results to previous version.
  - `'rec'` - Suggested settings. Significant results change possible

- `'old'` - Reproduce results of previous version

- **rename_evolved** (`bool`, optional) – If *True*, any model that is evolved during opening will be renamed.
- **evolve_dependencies** (`bool`, optional) – If *True*, subassembly models referred to by the model being opened will also be evolved. Note that this does NOT affect compound (.gtc) files referenced by SubassemblyExternal, these must be evolved in order for the parent model to be opened.
- **rename_dict** (`dict`, optional) – A dictionary that maps original model names to the name to use for the evolved version of that file. If *rename_evolved* is *False*, this parameter is not used. If a file is to be renamed and it does not have an entry in this *dict*, the default behavior is to append '*_v<version>*'.

Usage in GT-POST

Starting in v2021, GT-SUITE output files will now have the new extension .glx instead of the old extension of .gdx. This change was required due to adopting a new database storage technology which offers many performance benefits.

Legacy .gdx files can still be accesed via the GT-POST Python API in v2021 using the *open_document()* method. However, they will be silently converted to the new .glx file format prior to being opened. Note that if a .glx version of the .gdx file already exists in the local directory, it will be overwritten by this operation.

All evolution parameters should be omitted when opening output (.glx) files, regardless of version.

Notes

If the model is from a previous version, the *evolution_option* parameter must be specified, otherwise an *OpenDocumentError* will be raised. If the model is not from a previous version, all of the evolution parameters will be ignored.

Returns

A *Document* object for the document that was opened if a document with the given path was found, otherwise raises an *OpenDocumentError*.

Return type

`Document`

Raises

OpenDocumentError – If the file with the given path does not exist, the document at the given path is an unsupported type, or if any other error occurs while opening.

values() → List[gtisoft.core.documents.Document]

Gets a list of *Document* objects for all open documents.

Returns

A list of all open documents.

Return type

`list` of `Document`

---

class **gtisoft.core.documents.Document**(*document_wrapper*)

Bases: `object`

A *Document* object represents and provides methods for interacting with an open document in a GT application.

close() → None

Closes this document in the application.

Returns

No return value.

Return type

None

save() → None

Saves this document to its existing file.

Returns

No return value.

Return type

None

Raises

- **SaveDocumentError** – If there is a problem saving the document.
- **DocumentClosedError** – If `close()` was previously called for the document.

save_as(*file_path: str*) → None

Saves this document to a new file.

Parameters

file_path (`str`) – The full path of the target file.

Returns

No return value.

Return type

None

Raises

- **SaveDocumentError** – If there is a problem saving the document.
- **DocumentClosedError** – If `close()` was previously called for the document.

property **is_closed**

`bool`: **Read-Only**; *True* if *close* has been called on this document.

property **name**

`str`: **Read-Only**; The name of this document is the file path if it has been saved to disk, or else the file name if it has not been saved.

property **version**

`GTVersion` **Read-Only**; The *GTVersion* instance that contains the version information for this document.

---

class **gtisoft.core.documents.GtiseDocument**(*document_wrapper*)

Bases: `gtisoft.core.documents.Document`

close() → None

Closes this document in the application.

Returns

No return value.

Return type

None

property **advanced_setup**

`gtisoft.core.objects.Object`: ***Read-Only**; an *Object* instance for accessing properties of the Advanced Setup dialog. May return *None* if Advanced Setup is not available for this document.

property **case_setup**

`gtisoft.core.case_setup.CaseSetup`: ***Read-Only**; a *CaseSetup* instance for accessing the and modifying run cases, parameter folders, and parameters in this document's Case Setup dialog.

property **design_optimizer**

`gtisoft.core.objects.Object`: ***Read-Only**; an *Object* instance for accessing properties of the Design Optimizer dialog. May return *None* if the Design Optimizer is not available for this document.

property **license_type**

Gets or sets the license type for this document.

`GtLicenseType`:

The license this document is associated with.

Raises

exceptions.ValueError – If license type is incompatible or invalid in document when accessing.

property **links**

`gtisoft.core.links.Links`: **Read-Only**; The *Links* instance that contains the collection of physics links for this document.

property **objects**

`gtisoft.core.objects.Objects`: **Read-Only**; The *Objects* instance that contains the collection of GT-Objects for this document.

property **output_setup**

`gtisoft.core.objects.Object`: ***Read-Only**; an *Object* instance for accessing properties of the Output Setup dialog. May return *None* if Output Setup is not available for this document.

property **parts**

`gtisoft.core.parts.Parts`: **Read-Only**; The *Parts* instance that contains the collection of parts for this document.

property **run_setup**

`gtisoft.core.objects.Object`: ***Read-Only**; an *Object* instance for accessing properties of the Run Setup dialog. May return *None* if the Run Setup is not available for this document.

property **signals**

`gtisoft.core.signals.Signals`: **Read-Only**; The *Signals* instance that contains the collection of controls links for this document.

class **gtisoft.core.documents.GtpostDocument**(*document_wrapper*)

Bases: `gtisoft.core.documents.Document`

get_case_rlts() → EntityList

Gets the case RLTs for this document.

Returns

An *EntityList* instance containing the *TemplateName* groups that are the root entities of the case RLTs tree.

Return type

`EntityList`

get_end_of_run_plots() → EntityList

Gets the end of run plots for this document.

Returns

An *EntityList* object containing the *TemplateName:PartName* groups that are the root entities of the end of run plots tree.

Return type

`EntityList`

get_plots(*case_number: int*) → EntityList

Gets the instantaneous plots for the given case number from this document.

Parameters

case_number (`int`) – The case number for which to get the instantaneous plots.

Returns

An *EntityList* object containing the *TemplateName:PartName* groups that are the root entities of the instantaneous plots tree.

Return type

`EntityList`

get_time_rlts(*case_number: int*) → EntityList

Gets the time RLTs for the given case number from this document.

Parameters

case_number (`int`) – The case number for which to get the time RLTs.

Returns

An *EntityList* object containing the *TemplateName* groups that are the root entities of the time RLTs tree.

Return type

`EntityList`

property **cases**

`EntityList`: ***Read-Only**; an *EntityList* instance containing an *Entity* object for each case.

---

class **gtisoft.core.documents.GuDocument**(*document_wrapper*)

Bases: `gtisoft.core.documents.Document`

property **plots**

`EntityList`: ***Read-Only**; an *EntityList* instance containing group *Entity* objects that are the root of the GU file Plots tree.

---

class **gtisoft.core.documents.GemDocument**(*document_wrapper*)

Bases: `gtisoft.core.documents.Document`

close() → None

Closes this document in the application.

Returns

No return value.

Return type

None

property **case_setup**

`gtisoft.core.case_setup.CaseSetup`: ***Read-Only**; a *CaseSetup* instance for accessing the and modifying run cases, parameter folders, and parameters in this document's Case Setup dialog.

property **objects**

`gtisoft.core.objects.Objects`: **Read-Only**; The *Objects* instance that contains the collection of GT-Objects for this document.

---

class **gtisoft.core.documents.GtLicenseType**

Bases: `gtisoft.util.collect_util.EnumMixin`, `enum.Enum`

An enumeration.

GT_AUTOLION= *'GT-AutoLion'*

GT_DRIVE= *'GT-DRIVE+'*

GT_POWER= *'GT-POWER'*

GT_POWER_LAB= *'GT-POWERLab'*

GT_POWER_XRT= *'GT-POWER-xRT'*

GT_PROCESS= *'GT-ProcessMap'*

GT_SUITE= *'GT-SUITE'*

GT_SUITE_MP= *'GT-SUITEmp'*

GT_SUITE_RT= *'GT-SUITE-RT'*

GT_XLINK= *'GT-xLINK'*

---

class **gtisoft.core.documents.GtDocumentType**

Bases: `gtisoft.util.collect_util.EnumMixin` , `enum.Enum`

An enumeration.

GT_AUTO_LION= *'autolion'*

GT_COMPOUND= *'gtc'*

GT_DRIVE_PLUS= *'gtdrive'*

GT_EXTERNAL_ASSEMBLY= *'gtsub'*

GT_GEM= *'gem'*

GT_GEM_COOL= *'ghx'*

GT_MODEL= *'gtm'*

GT_OBJECT_LIBRARY= *'gto'*

GT_PROCESS= *'gtprocess'*

GT_REPORT_FILE= *'gu'*

GT_XLINK= *'xlink'*

Document Tools

The Document Tools module contains functions and utilities that can be performed at the document level.

Note that when accessing the GT-SUITE Python API externally, the *document_tools* module must be imported prior to using the functions below. To import all functions, the following import statement can be used:

```
from gtisoft.tools.document_tools import *
```

Module Documentation

gtisoft.tools.document_tools.run_xrt_advisor(*document: gtisoft.core.documents.GtiseDocument*) → None

Runs the GT-POWER-xRT Advisor with default arguments. If values are already at their default recommended value, no action is taken.

Returns

No return value.

Return type

None

gtisoft.tools.document_tools.get_unused_objects(*document: gtisoft.core.documents.GtiseDocument*) → Optional[List[gtisoft.core.objects.Object]]

Gets unused objects in a model.

Returns

A list of the unused objects

Return type

`list` of `Object`

gtisoft.tools.document_tools.delete_unused_objects(*document: gtisoft.core.documents.GtiseDocument, force: bool = False*) → None

Deletes unused objects in a model.

Parameters

force (`bool` , optional) – If *True*, forces deletion of objects that may optionally be used without dependency. *False* by default.

Returns

No return value.

Return type

None

---

gtisoft.tools.document_tools.delete_unused_templates(*document: gtisoft.core.documents.GtiseDocument*) → None

Deletes unused templates in a model. Should be called after *unused_objects* when wanting to delete both unused objects and templates.

Returns

No return value.

Return type

None

---

gtisoft.tools.document_tools.combine_flow_volume_to_pipe(*document: gtisoft.core.documents.GtiseDocument, parts: List[gtisoft.core.parts.Part]*) → None

Combines the passed in parts to a pipe. All parts must be in the same assembly.

Parameters

parts (`list` of `Part`) – A list of parts to combine into a pipe. **Note:** the order of the parts in the list may influence certain properties inside the resulting pipe.

---

gtisoft.tools.document_tools.combine_flow_volume_to_flowsplit(*document: gtisoft.core.documents.GtiseDocument, parts: List[gtisoft.core.parts.Part]*) → None

Combines the passed in parts to a flowsplit. All parts must be in the same assembly.

Parameters

parts (`list` of `Part`) – A list of parts to combine into a flowsplit. **Note:** the order of the parts in the list may influence certain properties inside the resulting flowsplit.

---

gtisoft.tools.document_tools.gt_realdrive_calculate_route(*object: gtisoft.core.objects.Object*) → None

Generates a GT-RealDrive profile from a ProfileGpsRoute object with pre-filled route input attributes and stores it in the object.

Note: This method will replace the profile data if it is already present.

Parameters

object (`Object`) – The ProfileGpsRoute object to generate the GT-RealDrive profile for.

Entities

The *Entities* module contains functions and classes for manipulating elements of a GT-POST document (GU and GLX).

Module Documentation

class **gtisoft.core.entities.Entity**(*wrapper*)

Bases: `gtisoft.core.entities.EntitySelectorMixin`, `gtisoft.core.entities.EntityTreeWalkerMixin`, `gtisoft.core.entities.EntityCreatorMixin`

The *Entity* class is the base class for all entities in the GT-POST API. An 'Entity' object represents a database item in a GLX or GU. Most of the elements visible in the GT-POST tree have a corresponding *Entity*: Cases, Groups, Plots, etc.

This class is not explicitly returned by API methods but forms the base functionality for all entities.

property **children**

Gets the children (direct descendants) of this *Entity*.

Returns

An *EntityList* object with this Entity's children (may be empty if the entity has no children)

Return type

`EntityList`

property **entity_name**

`str`:

Gets or sets the name of this *Entity*.

Raises

ValueError – When setting the name, if the new name is invalid.

property **entity_type**

`EntityType`: **Read-Only**; The type of item this *Entity* represents.

property **parent**

`Entity`: **Read-Only**; this *Entity* object's parent *Entity* if it has one, or else *None*.

---

class **gtisoft.core.entities.EntityType**

Bases: `enum.Enum`

This enum contains members representing the valid types of entities in a GLX or GU.

CASE= *2*

DATA= *5*

GROUP= *3*

PLOT= *4*

---

class **gtisoft.core.entities.DefaultEntity**(*wrapper*)

Bases: `gtisoft.core.entities.Entity`

DefaultEntity is the base class for entities that have properties.

This class is not explicitly returned by API methods but forms the base functionality for all entities that contain properties.

property **properties**

GT-POST plot and dataset properties are accessed using the same data structures as part and object attribute values in GT-ISE. The *properties* field is an instance of *Attributes* that provides access to the properties for this entity.

`gtisoft.core.attributes.Attributes`:

Read-Only; The *Attributes* instance that contains the collection of attribute values (properties) for this *Entity*.

---

class **gtisoft.core.entities.GroupEntity**(*wrapper*)

Bases: `gtisoft.core.entities.DefaultEntity`

A *GroupEntity* object represents a generic tree entity in GT-POST, such as a *Template:PartName* node in an instantaneous plot tree or a user-created GU group.

---

class **gtisoft.core.entities.CaseEntity**(*wrapper*)

Bases: `gtisoft.core.entities.DefaultEntity`

CaseEntity objects represent a run case in GT-POST.

property **case_label**

`str` : **Read-Only**; The label for this case as seen in case setup.

property **case_number**

`int` : **Read-Only**; The number for this case as seen in case setup.

Note: This is not necessarily equal to *case_index* if not all cases are active.

property **is_failed**

`bool` : **Read-Only**; *True* if the simulation failed for this case, otherwise *False*.

property **is_skipped**

`bool` : **Read-Only**; *True* if this case was skipped, otherwise *False*.

---

class **gtisoft.core.entities.PlotEntity**(*wrapper*)

Bases: `gtisoft.core.entities.DefaultEntity`

PlotEntity objects represent a plot in GT-POST.

set_contour_unit(*unit: Union[gtisoft.core.units.Unit, List[gtisoft.core.units.Unit]]*)

Sets the unit for this plot's contour axis.

Parameters

unit (`Unit` or `list` of `Unit`) – The new unit to use; may be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.

Raises

exceptions.ValueError – If the given unit is not valid for the contour axis

set_x_unit(*unit: Union[gtisoft.core.units.Unit, List[gtisoft.core.units.Unit]]*)

Sets the unit for this plot's X axis.

Parameters

unit (`Unit` or `list` of `Unit`) – The new unit to use; may be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.

Raises

exceptions.ValueError – If the given unit is not valid for the X axis

set_y2_unit(*unit: Union[gtisoft.core.units.Unit, List[gtisoft.core.units.Unit]]*)

Sets the unit for this plot's Y2 axis.

Parameters

unit (`Unit` or `list` of `Unit`) – The new unit to use; may be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.

Raises

exceptions.ValueError – If the given unit is not valid for the Y2 axis

set_y3_unit(*unit: Union[gtisoft.core.units.Unit, List[gtisoft.core.units.Unit]]*)

Sets the unit for this plot's Y3 axis.

Parameters

unit (`Unit` or `list` of `Unit`) – The new unit to use; may be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.

Raises

exceptions.ValueError – If the given unit is not valid for the Y3 axis

set_y4_unit(*unit: Union[gtisoft.core.units.Unit, List[gtisoft.core.units.Unit]]*)

Sets the unit for this plot's Y4 axis.

Parameters

unit (`Unit` or `list` of `Unit`) – The new unit to use; may be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.

Raises

exceptions.ValueError – If the given unit is not valid for the Y4 axis

set_y_unit(*unit: Union[gtisoft.core.units.Unit, List[gtisoft.core.units.Unit]]*)

Sets the unit for this plot's Y axis.

Parameters

unit (`Unit` or `list` of `Unit`) – The new unit to use; may be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.

Raises

exceptions.ValueError – If the given unit is not valid for the Y axis

`set_z_unit`(*unit: Union[gtisoft.core.units.Unit, List[gtisoft.core.units.Unit]]*)

Sets the unit for this plot's Z axis.

Parameters

unit (`Unit` or `list` of `Unit`) – The new unit to use; may be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.

Raises

exceptions.ValueError – If the given unit is not valid for the Z axis

property `contour_unit`

`Unit` : **Read-Only**; The unit to use for this plot's contour axis. Returns *None* if this plot does not have a contour axis.

property `x_unit`

`Unit` : **Read-Only**; The unit to use for this plot's X axis. Returns *None* if this plot does not have an X axis.

property `y2_unit`

`Unit` : **Read-Only**; The unit to use for this plot's Y2 axis. Returns *None* if this plot does not have a Y2 axis.

property `y3_unit`

`Unit` : **Read-Only**; The unit to use for this plot's Y3 axis. Returns *None* if this plot does not have a Y3 axis.

property `y4_unit`

`Unit` : **Read-Only**; The unit to use for this plot's Y4 axis. Returns *None* if this plot does not have a Y4 axis.

property **y_unit**

`Unit`: **Read-Only**; The unit to use for this plot's Y axis. Returns *None* if this plot does not have a Y axis.

property **z_unit**

`Unit`: **Read-Only**; The unit to use for this plot's Z axis. Returns *None* if this plot does not have a Z axis.

---

class **gtisoft.core.entities.DatasetEntity**(*wrapper*)

Bases: `gtisoft.core.entities.DefaultEntity`

DatasetEntity objects represent a plot's dataset in GT-POST.

set_contour_unit(*unit: Union[gtisoft.core.units.Unit, List[gtisoft.core.units.Unit]], category: gtisoft.core.units.UnitCategory = None*)

Sets the unit for this dataset's contour axis.

Parameters

- **unit** (`Unit` or `list` of `Unit`) – The new unit to use; may be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.
- **category** (`UnitCategory`, optional) – The category for the unit to set. This is required to disambiguate the *unit* parameter in the case it is a *list*.

Raises

exceptions.ValueError – If the dataset does not have a contour unit, or if the given unit list cannot be resolved.

set_x_unit(*unit: Union[gtisoft.core.units.Unit, List[gtisoft.core.units.Unit]], category: gtisoft.core.units.UnitCategory = None*)

Sets the unit for this dataset's X axis.

Parameters

- **unit** (`Unit` or `list` of `Unit`) – The new unit to use; may be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.
- **category** (`UnitCategory`, optional) – The category for the unit to set. This is required to disambiguate the *unit* parameter in the case it is a *list*.

Raises

exceptions.ValueError – If the dataset does not have an X unit, or if the given unit list cannot be resolved.

set_y_unit(*unit: Union[gtisoft.core.units.Unit, List[gtisoft.core.units.Unit]], category: gtisoft.core.units.UnitCategory = None*)

Sets the unit for this dataset's Y axis.

Parameters

- **unit** (`Unit` or `list` of `Unit`) – The new unit to use; may be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.
- **category** (`UnitCategory` , optional) – The category for the unit to set. This is required to disambiguate the *unit* parameter in the case it is a *list*.

Raises

exceptions.ValueError – If the dataset does not have a Y unit, or if the given unit list cannot be resolved.

set_z_unit(*unit: Union[gtisoft.core.units.Unit, List[gtisoft.core.units.Unit]], category: gtisoft.core.units.UnitCategory = None*)

Sets the unit for this dataset's Z axis.

Parameters

- **unit** (`Unit` or `list` of `Unit`) – The new unit to use; may be a single instance of *Unit* or a list. The list facilitates ease of use with a `gtisoft.core.units.Units` dictionary, which may return more than one *Unit* if multiple units share the same name.
- **category** (`UnitCategory` , optional) – The category for the unit to set. This is required to disambiguate the *unit* parameter in the case it is a *list*.

Raises

exceptions.ValueError – If the dataset does not have a Z unit, or if the given unit list cannot be resolved.

property **contour_unit**

`Unit` : **Read-Only**; The contour unit for this dataset. Returns *None* if this dataset does not have a contour unit.

property **data**

`Data` :

Gets or sets the raw data for this dataset.

Parameters

data_map (`Data` or `dict`) – Setting the *data* property for this dataset will accept either another *Data* object to copy, or a dictionary that maps the data axis name (e.g. 'X', 'Y', etc) to a list of *float*, *int*, or *str* values.

Examples:

```
>>> dataset_A = plot_from_glx.dataset
>>> dataset_B = plot_from_gu.create_dataset('xy', 'Example Dataset; copy from GLX
plot')
>>> dataset_B.data = dataset_A.data
```

```
>>> dataset = plot.create_dataset('xy', 'Example Dataset; create from list')
>>> dataset.data = {'x':[1,2,3], 'y':[4,5,6]}
```

property **x_unit**

`Unit` : **Read-Only**; The X unit for this dataset. Returns *None* if this dataset does not have an X unit.

property **y_unit**

`Unit` : **Read-Only**; The Y unit for this dataset. Returns *None* if this dataset does not have a Y unit.

property **z_unit**

`Unit` : **Read-Only**; The Z unit for this dataset. Returns *None* if this dataset does not have a Z unit.

class **gtisoft.core.entities.Data**(*wrapper*)

Bases: `object`

A *Data* object contains the raw data for a *DatasetEntity*. This is the data as it exists in the source file or database.

property **available_data**

`list` of *str*: **Read-Only**; A list of the names of the available data in this *Data* object.

property **data_type**

`str` : **Read-Only**; The type of data represented by this *Data* object.

property **formatted**

`str`: **Read-Only**; A formatted string representation of this *Data* object. Contains metadata on available properties and their values as well as the data itself. Data is tab-delimited for easy pasting into other applications.

---

class **gtisoft.core.entities.EntityList**(*wrapper*)

Bases: `collections.abc.MutableSequence`, `gtisoft.core.entities.EntitySelectorMixin`, `gtisoft.core.entities.EntityTreeWalkerMixin`, `gtisoft.core.entities.EntityCreatorMixin`

This class is a collection of zero or more *Entity* objects.

Notes

EntityList is a `Sequence` that behaves similarly to a Python `list`.

Much of the behavior that is present in *list* is also present in *EntityList*. For instance checking the number of values:

```
>>> entities = parent_entity.children
>>> print(len(entities))
3
```

EntityList is also "scriptable", enabling getting a value at a specific index using brackets:

```
>>> print(entities[0])
Entity(entity_type=<EntityType.GROUP: 3>, entity_name='AcoustExtMicrophone:Exh-Mic')
>>> print(entities[-1])
Entity(entity_type=<EntityType.GROUP: 3>, entity_name='ValveCamConn:InVal1B')
```

and it supports "slicing":

```
>>> print(entities[0:2])
[Entity(entity_type=<EntityType.GROUP: 3>, entity_name='AcoustExtMicrophone:Exh-Mic'),
    Entity(entity_type=<EntityType.GROUP: 3>, entity_name='AcoustExtMicrophone:Int-Mic')]
```

insert(*index: int*, *value: gtisoft.core.entities.Entity*)

S.insert(index, value) – insert value before index

---

class **gtisoft.core.entities.EntitySelectorMixin**

Bases: `object`

This mixin defines methods for navigating an entity tree (e.g. 'Instantaneous Plots') by selecting nodes based on type and optional name filters.

The general tree hierarchy of entities in GT-POST is: *Group* -> *Plot* -> *Dataset*. Some trees may repeat certain levels, and the levels may be selectable by sub-type.

Some *Group* entities may have an implicit sub-type based on what tree type they are a part of. A call to *get_all_groups* may return any of these, but there are specific methods to filter by the sub-type:

- Template - Groups that represent a template grouping (e.g. 'PipeRound')
- Part - Groups that represent a specific part and its plots (e.g. 'EngCylinder-01')
- Folder - Groups that separate RLTs by some scheme (e.g. 'Pressure' or 'Thermal')

The specific hierarchies of the different tree types are:

- Instantaneous Plots: *Group* (a Template:Part pair) -> *Plot* -> *Dataset*
- Case RLTs: *Group* (Template) -> *Group* (Part) -> *Group* (RLT Folder) -> *Plot* -> *Dataset*
- Time RLTs: *Group* (Template) -> *Group* (Part) -> *Group* (RLT Folder) -> *Plot* -> *Dataset*
- GU Plots: *Group* -> *Plot* -> *Dataset*

get_all_datasets(*dataset_name: str = None*) → gtisoft.core.entities.EntityList

Recursively searches all children and returns the set of child *Entity* objects that have type *EntityType.DATA* and match the given name.

This method supports wildcards as part of the *data_name* argument, for instance:

```
>>> datasets = gt_post_doc.get_plots(case_number=1).get_all_datasets('RPM =*')
```

Parameters

dataset_name (`str` , optional) – If present, the child groups will be filtered to those that have the given name, or match a given name with wildcards. If *None*, this method will return all child data entities.

Returns

A new *EntityList* collection containing all matching entities. If no entities match, the *EntityList* will be empty.

Return type

`EntityList`

get_all_folders(*folder_name: str = None*) → gtisoft.core.entities.EntityList

Recursively searches all children and returns the set of child *Entity* objects that have type *EntityType.GROUP*, represent an RLT folder in an RLT tree, and match the given name.

This method supports wildcards as part of the *folder_name* argument, for instance:

```
>>> folders = gt_post_doc.case_rlts.get_all_folders('Timing*')
```

Parameters

folder_name (`str` , optional) – If present, the child groups will be filtered to those that have the given name, or match a given name with wildcards. If *None*, this method will return all child group entities that represent an RLT folder.

Returns

A new *EntityList* collection containing all matching entities. If no entities match, the *EntityList* will be empty.

Return type

`EntityList`

get_all_groups(*group_name: str = None*) → gtisoft.core.entities.EntityList

Recursively searches all children and returns the set of child *Entity* objects that have type *EntityType.GROUP* and match the given name.

This method supports wildcards as part of the *group_name* argument, for instance:

```
>>> groups = gt_post_doc.get_plots(case_number=1).get_all_groups('PipeRound:*')
```

Parameters

group_name (`str` , optional) – If present, the child groups will be filtered to those that have the given name, or match a given name with wildcards. If *None*, this method will return all child group entities.

Returns

A new *EntityList* collection containing all matching entities. If no entities match, the *EntityList* will be empty.

Return type

`EntityList`

get_all_parts(*part_name: str = None*) → gtisoft.core.entities.EntityList

Recursively searches all children and returns the set of child *Entity* objects that have type *EntityType.GROUP*, represent a Part in an RLT or Plot tree, and match the given name.

This method supports wildcards as part of the *part_name* argument, for instance:

```
>>> parts = gt_post_doc.get_plots(case_number=1).get_all_parts('PipeRound-*')
```

Parameters

part_name (`str` , optional) – If present, the child groups will be filtered to those that have the given name, or match a given name with wildcards. If *None*, this method will return all child group entities that represent a part.

Returns

A new *EntityList* collection containing all matching entities. If no entities match, the *EntityList* will be empty.

Return type

`EntityList`

get_all_plots(*plot_name: str = None*) → gtisoft.core.entities.EntityList

Recursively searches all children and returns the set of child *Entity* objects that have type *EntityType.PLOT* and match the given name.

This method supports wildcards as part of the *plot_name* argument, for instance:

```
>>> plots = gt_post_doc.get_plots(case_number=1).get_all_plots('Valve*')
```

Parameters

plot_name (`str` , optional) – If present, the child groups will be filtered to those that have the given name, or match a given name with wildcards. If *None*, this method will return all child plot entities.

Returns

A new *EntityList* collection containing all matching entities. If no entities match, the *EntityList* will be empty.

Return type

`EntityList`

get_all_templates(*template_name: str = None*) → gtisoft.core.entities.EntityList

Recursively searches all children and returns the set of child *Entity* objects that have type *EntityType.GROUP*, represent a Template in an RLT or Plot tree, and match the given name.

This method supports wildcards as part of the *template_name* argument, for instance:

```
>>> templates = gt_post_doc.get_plots(case_number=1).get_all_templates('Pipe*')
```

Parameters

template_name (`str` , optional) – If present, the child groups will be filtered to those that have the given name, or match a given name with wildcards. If *None*, this method will return all child group entities that represent a template.

Returns

A new *EntityList* collection containing all matching entities. If no entities match, the *EntityList* will be empty.

Return type

`EntityList`

get_dataset(*dataset_name: str = None*) → Optional[gtisoft.core.entities.Entity]

Recursively searches all children and returns the first child *Entity* object that has type *EntityType.DATA* and matches the given name.

This method supports wildcards as part of the *dataset_name* argument, for instance:

```
>>> dataset = gt_post_doc.get_plots(case_number=1).get_dataset('RPM =*')
```

Parameters

dataset_name (`str` , optional) – If present, the child dataset will be the first of those that have the given name, or matches a given name with wildcards. If *None*, this method will return the first child dataset.

Returns

- *None* – If there are no child dataset or no dataset that match the criteria.
- `Entity` – The first child entity that matches the criteria.

get_folder(*folder_name: str = None*) → Optional[gtisoft.core.entities.Entity]

Recursively searches all children and returns the first child *Entity* object that has type *EntityType.GROUP*, represents an RLT folder in an RLT tree, and matches the given name.

This method supports wildcards as part of the *folder_name* argument, for instance:

```
>>> folder = gt_post_doc.case_rlts.get_folder('Timing*')
```

Parameters

folder_name (`str` , optional) – If present, the child folder will be the first of those that have the given name, or matches a given name with wildcards. If *None*, this method will return the first child folder.

Returns

- *None* – If there are no child folders or no folders that match the criteria.
- `Entity` – The first child entity that matches the criteria.

get_group(*group_name: str = None*) → Optional[gtisoft.core.entities.Entity]

Recursively searches all children and returns the first child *Entity* object that has type *EntityType.GROUP* and matches the given name.

This method supports wildcards as part of the *group_name* argument, for instance:

```
>>> group = gt_post_doc.get_plots(case_number=1).get_group('PipeRound:*')
```

Parameters

group_name (`str` , optional) – If present, the child group will be the first of those that have the given name, or matches a given name with wildcards. If *None*, this method will return the first child group.

Returns

- *None* – If there are no child groups or no groups that match the criteria.
- `Entity` – The first child entity that matches the criteria.

get_part(*part_name: str = None*) → Optional[gtisoft.core.entities.Entity]

Recursively searches all children and returns the first child *Entity* object that has type *EntityType.GROUP*, represents a Part in an RLT or Plot tree, and matches the given name.

This method supports wildcards as part of the *part_name* argument, for instance:

```
>>> part = gt_post_doc.get_plots(case_number=1).get_parts('PipeRound-*')
```

Parameters

part_name (`str` , optional) – If present, the child part will be the first of those that have the given name, or matches a given name with wildcards. If *None*, this method will return the first child part.

Returns

- *None* – If there are no child parts or no parts that match the criteria.
- `Entity` – The first child entity that matches the criteria.

get_plot(*plot_name: str = None*) → Optional[gtisoft.core.entities.Entity]

Recursively searches all children and returns the first child *Entity* object that has type *EntityType.PLOT* and matches the given name.

This method supports wildcards as part of the *plot_name* argument, for instance:

```
>>> plot = gt_post_doc.get_plots(case_number=1).get_plot('Valve*')
```

Parameters

plot_name (`str` , optional) – If present, the child plot will be the first of those that have the given name, or matches a given name with wildcards. If *None*, this method will return the first child plot.

Returns

- *None* – If there are no child plots or no plots that match the criteria.
- `Entity` – The first child entity that matches the criteria.

get_template(*template_name: str = None*) → Optional[gtisoft.core.entities.Entity]

Recursively searches all children and returns the first child *Entity* object that has type *EntityType.GROUP*, represents a Template in an RLT or Plot tree, and matches the given name.

This method supports wildcards as part of the *template_name* argument, for instance:

```
>>> template = gt_post_doc.get_plots(case_number=1).get_template('Pipe*')
```

Parameters

template_name (`str` , optional) – If present, the child template will be the first of those that have the given name, or matches a given name with wildcards. If *None*, this method will return the first child template.

Returns

- *None* – If there are no child templates or no templates that match the criteria.
- `Entity` – The first child entity that matches the criteria.

class **gtisoft.core.entities.EntityTreeWalkerMixin**

Bases: `object`

This mixin provides a method of walking an entity tree (e.g. 'Instantaneous Plots') and applying a function to each Entity it contains.

get_tree_string(*predicate: Callable[[Entity], bool] = <function EntityTreeWalkerMixin.<lambda>>*) → str

Gets a string representation of the tree rooted with this *EntityList* collection.

Parameters

predicate (`Callable[['Entity'], bool]` , optional) – A function whose argument is the entity being visited and returns a *bool* value for whether or not the entity should be included in the output string. If no predicate is specified, all entities will be included.

Returns

A string representation of the tree for this *EntityList* collection.

Return type

`str`

`walk`(*visitor: Callable[[Entity, int], None], predicate: Callable[[Entity], bool] = <function EntityTreeWalkerMixin.<lambda>>, level: int = 0*) → None

Recursively walks and invokes a visitor on each *Entity* in this collection (if this is an *EntitySet*) or this *Entity* (if this is an *Entity*) and all child entities.

Examples

A simple use case of the *walk* method is to print a string representation of the tree for an Entity and its descendants:

```
>>> part = doc.get_part('EngCylinder-01')
>>> part.walk(lambda e, level: print('  ' * level + e.entity_name))
```

In this example, we are passing in a visitor function via the lambda, that takes an entity *e* and the tree level *level* as arguments. It prints the proper indentation ‘ ‘ * *level* and then the name of the entity.

Suppose we are not interested in printing any datasets, only the names of the plots. We can expand this example to include a predicate function that will filter out datasets:

```
>>> part = doc.get_part('EngCylinder-01')
>>> part.walk(lambda e, level: print('  ' * level + e.entity_name), lambda e:
e.entity_type is not EntityType.DATA)
```

Parameters

- **visitor** (`Callable[['Entity', int], None]`) – A visitor function whose arguments are the entity being visited and the current level of the tree where the entity resides.
- **predicate** (`Callable[['Entity'], bool]` , optional) – A function whose argument is the entity being visited and returns a *bool* value for whether or not the entity should be visited. If no predicate is specified, all entities will be visited.
- **level** (`int`) – The tree level of the Entity objects being visited.

Returns

Return type

None

class **gtisoft.core.entities.EntityCreatorMixin**

Bases: `object`

This mixin provides a method of creating new entities in an entity tree (e.g. the plots tree of a GU file).

create_dataset(*dataset_type: str*, *dataset_name: str = None*) → gtisoft.core.entities.Entity

Creates a new dataset entity and adds it to this *EntityList* collection or to the *children EntityList* if this is an *Entity*.

Parameters

- **dataset_type** (`str`) – The type of dataset to create (e.g. 'xy', 'xyz', 'tr').
- **dataset_name** (`str` , optional) – The new dataset's name.

Returns

The new dataset entity that was created.

Return type

`Entity`

Raises

RuntimeError – If the parent *Entity* of this *EntityList* collection (or this entity if called on an *Entity*) does not support adding a new dataset.

create_group(*group_name: str = None*) → gtisoft.core.entities.Entity

Creates a new group entity and adds it to this *EntityList* collection or to the *children EntityList* if this is an *Entity*.

Parameters

group_name (`str` , optional) – The new group's name.

Returns

The new group entity that was created.

Return type

`Entity`

Raises

RuntimeError – If the parent *Entity* of this *EntityList* collection (or this entity if called on an *Entity*) does not support adding a new group.

create_plot(*plot_type: str*, *plot_name: str = None*) → gtisoft.core.entities.Entity

Creates a new plot entity and adds it to this *EntityList* collection or to the *children EntityList* if this is an *Entity*.

Parameters

- **plot_type** (`str`) – The type of plot to create (e.g. 'xy', 'contour', '3dfe').
- **plot_name** (`str` , optional) – The new plot's name.

Returns

The new plot entity that was created.

Return type

`Entity`

Raises

RuntimeError – If the parent *Entity* of this *EntityList* collection (or this entity if called on an *Entity*) does not support adding a new plot.

create_separator(*separator_name: str = None*) → gtisoft.core.entities.Entity

Creates a new separator entity and adds it to this *EntityList* collection or to the *children EntityList* if this is an *Entity*.

Parameters

separator_name (`str` , optional) – The new separator's name.

Returns

The new separator entity that was created.

Return type

`Entity`

Raises

RuntimeError – If the parent *Entity* of this *EntityList* collection (or this entity if called on an *Entity*) does not support adding a new separator.

Plot and Dataset Types

Plots

	Name	*plot_type*	Allowed Dataset Types
	XY Scatter Plot	*'xy'*	'xy', 'tr', 'text'
	2D Contour Plot	*'contour'*	'xyz', 'xyzz', 'xxyyzz', 'xy', 'text'
	3D Contour Plot	*'3dcontour'*	'xyz', 'xyzz', 'xxyyzz', 'text'
	Polar Scatter Plot	*'polar'*	'tr', 'text'
	Polar Contour Plot	*'polarcontour'*	'trz', 'trzz', 'ttrrzz', 'tr', 'text'
	Carpet Plot	*'carpet'*	'xy', 'text'
	2D Spatial Plot	*'2d-spatial'*	'drawing', '2dfe', '2dfe_elements', 'text'
	3D Spatial Plot	*'3dfe'*	'3dfe', 'text', 'xyz', '3ddrawing'
	Bar Graph	*'bar'*	'cy', 'text'
	XY Bar Graph	*'barxy'*	'xy', 'text'
	Animation 3D	*'anim3d'*	'sphere', 'cylinder', 'box', 'cone', 'clipplane', 'arrow', 'arrows_length', 'extrusion', 'cad', 'sc', 'gear', 'text3d', 'rigidbody3d', 'assembly'
	Animation 3D Vehicle Driving	*'anim3d_reality'*	'road', 'vehicle'
	GPS Map	*'map'*	'gps_route'
	Parallel Coordinates Plot	*'parallel-coordinates'*	'nd', 'text'

Datasets

Name	Description	*dataset_type*
XY	2-D Cartesian Data Set	'xy'
ThetaR	Polar Data Set: Theta - Radius	'tr'
Text	Text Data Set	'text'
XYZ	3-D Cartesian Data Set of Scattered Points	'xyz'
XYZZ	3-D Cartesian Data Set of Structured Points	'xyzz'
XXYYZZ	3-D Cartesian Data Set of Mapped Points	'xxyyzz'
ThetaRZ	Cylindrical Data Set of Scattered Points: Theta-Radius-Z	'trz'
ThetaRZZ	Cylindrical Data Set of Structured Points: Theta-Radius-Z	'trzz'
ThetaThetaRRZZ	Cylindrical Data Set of Mapped Points: Theta-Radius-Z	'ttrrzz'
Drawing	Drawing Data Set	'drawing'
FE2D_Nodes	2-D Finite Element Data Set (triangle, quadrangle)	'2dfe'
FE2D Elements	2-D Finite Element Data Set (element)	'2dfe_elements'
FE3D	3-D Finite Element Data Set (tetrahedral, etc.)	'3dfe'
3DDrawing	3-D Drawing Data Set	'3ddrawing'
CY	String - Y Data Set	'cy'
Sphere	Sphere	'sphere'
Cylinder	Cylinder	'cylinder'
Cuboid	Cuboid	'box'
Cone	Cone	'cone'
Fluid Plane	Fluid Plane	'clipplane'
Arrow	Arrow	'arrow'
Arrows	Arrows	'arrows_length'
Extrusion	Extrusion	'extrusion'

Name	**Description**	***dataset_type***
CAD	CAD	‘cad’
ModelingDimensionGeometry	ModelingDimensionGeometry	‘sc’
Gear	Gear	‘gear’
Text 3D	Text 3D	‘text3d’
Rigid Body 3D	Rigid Body 3D	‘rigidbody3d’
Assembly	Assembly	‘assembly’
Road	Road	‘road’
Vehicle Subassembly	Vehicle Subassembly	‘vehicle’
GPS Route	GPS Route	‘gps_route’
N-Dimensional	N-D Cartesian Data Set of Scattered Points	‘nd’

Gtt

GT-SUITE stores static information about templates and their properties in a file called the GTT. The *Gtt* module provides methods to query some of this information.

Module Documentation

class **gtisoft.core.gtt.Gtt**(*unit_service_wrapper*)

Bases: `object`

This class contains collections for accessing the GTT database. This database provides static information (such and available units and unit categories) for the Application.

> *property* **unit_categories**
>
> `gtisoft.core.units.UnitCategories`: **Read-Only**; A collection of all unit categories available in the application.

> *property* **units**
>
> `gtisoft.core.units.Units`: **Read-Only**; A collection of all units available in the application.

Links

The *Links* module contains functions and classes for creating, editing, and deleting physical links between parts in a model.

Module Documentation

exception **gtisoft.core.links.CreateLinkError**(*message*)

Bases: `Exception`

Raised when there is an error attempting to create a link between parts.

Variables

message (`str`) – A message describing the error.

Parameters

message (`str` , optional) – A message describing the error.

exception **gtisoft.core.links.DeleteLinkError**(*message*)

Bases: `Exception`

Raised when there is an error attempting to delete a *Link* from this collection.

Variables

message (`str`) – A message describing the error.

Parameters

message (`str` , optional) – A message describing the error.

exception **gtisoft.core.links.InvalidPortError**(*message*)

Bases: `Exception`

Raised when there is an error attempting to change the port number or port folder on one side of a *Link*.

Variables

message (`str`) – A message describing the error.

Parameters

message (`str`, optional) – A message describing the error.

---

class **gtisoft.core.links.Links**(*link_provider_wrapper*)

Bases: `collections.abc.Mapping`

This class is a collection that belongs to a `gtisoft.core.documents.Document` object and contains all of the physical links for a model.

Notes

Links is a `Mapping` that behaves similarly to a Python `dict`. The keys to the mapping are a `dict` containing the part name, port number, and port folder name for both the start and end sides of the link:

```
key = {LINK_KEY_START_PART_NAME: start_part,
    LINK_KEY_START_PORT_ID: start_port_id,
    LINK_KEY_START_PORT_FOLDER_NAME: start_port_folder,
    LINK_KEY_END_PART_NAME: end_part,
    LINK_KEY_END_PORT_ID: end_port_id,
    LINK_KEY_END_PORT_FOLDER_NAME: end_port_folder}
```

and the values are instances of *Link* that represent a physical link in the model.

The *LINK_KEY_** string constants are provided at the top level of the module and can be used to construct a dictionary with the proper keys to use as a key in the *Links* collection.

The function `create_link_key()` is provided as a convenience for creating the `dict` keys for an *Links* collection:

```
key = create_link_key(
    start_part='InMan-01',
    start_port_id=3,
    start_port_folder='Flow',
    end_part='Bell1',
    end_port_id=None,
    end_port_folder=None)
```

Many of the `dict` methods are present in the *Links* class as well as many of the same capabilities.

For instance, checking for the presence of a link in a document can be done using the Python membership operators:

```
>>> key = create_link_key(
>>>         start_part='InMan-01',
>>>         start_port_id=3,
>>>         start_port_folder='Flow',
>>>         end_part='Bell1',
>>>         end_port_id=None,
>>>         end_port_folder=None)
>>> print(key in links)
True
```

```
>>> key = create_link_key(
>>>         start_part='Missing',
>>>         start_port_id=1,
>>>         start_port_folder='Does not Exist',
>>>         end_part='Missing-Connection',
>>>         end_port_id=None,
>>>         end_port_folder=None)
>>> print(key not in links)
True
```

Links is “scriptable”, enabling getting links from the model using brackets:

```
key = create_link_key(
    start_part='InMan-01',
    start_port_id=3,
    start_port_folder='Flow',
    end_part='Bell1',
    end_port_id=None,
    end_port_folder=None)
link = links[key]
```

Links also allows iterating over all the links in a model using the default iterator:

```
for key in links:
    print(key)
```

One difference between a standard `dict` and an *Links* object is that the *in*, *not in* and subscript operations will accept either the dictionary key or a `Link` instance:

```
>>> key = create_link_key(
>>>         start_part='InMan-01',
>>>         start_port_id=3,
>>>         start_port_folder='Flow',
>>>         end_part='Bell1',
>>>         end_port_id=None,
>>>         end_port_folder=None)
>>> print(key in links)
True
```

```
>>> link_1 = links[key]
>>> print(link_1 in links)
True
```

```
>>> link_2 = links[link_1]
>>> print(link_1 == link_2)
True
```

See also

Documentation for Python `dict` type.

create_link(*start_part*, *start_port_id*, *start_port_folder*, *end_part*, *end_port_id*, *end_port_folder*)

Creates a physical link between two parts in the model and adds the resulting *Link* instances to this collection (and the document that owns it).

Parameters

start_part (`str`) – The name of the part at the start of the link.

start_port_id: `int`

The ID of the port at the start of the link. May be *None* if there is no port at the start of the link.

start_port_folder: `str`

The name of the port folder at the start of the link. May be *None* if there is no port at the start of the link.

end_part: `str`

The name of the part at the end of the link.

end_port_id: `int`

The ID of the port at the end of the link. May be *None* if there is no port at the start of the link.

end_port_folder: `str`

The name of the port folder at the end of the link. May be *None* if there is no port at the start of the link.

Returns

A list of the newly created *Link* instances.

Return type

`list` of `Link`

Raises

CreateLinkError – If the link could not be created for any reason.

`delete_link`(*start_part*, *start_port_id*, *start_port_folder*, *end_part*, *end_port_id*, *end_port_folder*)

Deletes the link that exactly matches the given parameters.

Parameters

- **start_part** (`str`) – The name of the part at the start of the link.
- **start_port_id** (`int`) – The ID of the port at the start of the link. May be *None* if there is no port at the start of the link.
- **start_port_folder** (`str`) – The name of the port folder at the start of the link. May be *None* if there is no port at the start of the link.
- **end_part** (`str`) – The name of the part at the end of the link.
- **end_port_id** (`int`) – The ID of the port at the end of the link. May be *None* if there is no port at the start of the link.
- **end_port_folder** (`str`) – The name of the port folder at the end of the link. May be *None* if there is no port at the start of the link.

Returns

- `Link` – A *Link* instance representing the deleted link, if one was removed. The returned *Link* is read-only.
- *None* – If no link with the given parameters exists in this collection.

Raises

DeleteLinkError – If the link can not be deleted for any reason.

`find_links`(*start_part=None*, *start_port_id=None*, *start_port_folder=None*, *end_part=None*, *end_port_id=None*, *end_port_folder=None*)

Finds any links in this collection that match the given criteria. All parameters are optional and any parameter that is not provided or is *None* will be ignored when performing the search.

Parameters

- **start_part** (`str` , optional) – The name of the part at the start of the link.
- **start_port_id** (`int` , optional) – The ID of the port at the start of the link.
- **start_port_folder** (`str` , optional) – The name of the port folder at the start of the link.

- **end_part** (`str` , optional) – The name of the part at the end of the link.
- **end_port_id** (`int` , optional) – The ID of the port at the end of the link.
- **end_port_folder** (`str` , optional) – The name of the port folder at the end of the link.

Returns

A list of Links that match the given search criteria. An empty list will be returned if there are no matches.

Return type

`list` of `Link`

`get_link`(*start_part*, *start_port_id*, *start_port_folder*, *end_part*, *end_port_id*, *end_port_folder*)

Gets an instance of *Link* that exactly matches the given parameters.

Parameters

- **start_part** (`str`) – The name of the part at the start of the link.
- **start_port_id** (`int`) – The ID of the port at the start of the link. May be *None* if there is no port at the start of the link.
- **start_port_folder** (`str`) – The name of the port folder at the start of the link. May be *None* if there is no port at the start of the link.
- **end_part** (`str`) – The name of the part at the end of the link.
- **end_port_id** (`int`) – The ID of the port at the end of the link. May be *None* if there is no port at the start of the link.
- **end_port_folder** (`str`) – The name of the port folder at the end of the link. May be *None* if there is no port at the start of the link.

Returns

An instance of *Link* for the link that matches the given parameters.

Return type

`Link`

Raises

exceptions.KeyError – If no link with the given parameters exists in this collection.

`items`()

Gets a list of key-value pairs for all the links in this collection where the key is a *dict* containing the part name, port number, and port folder name for both the start and end sides of the link:

```
key = {
    LINK_KEY_START_PART_NAME: 'InMan-01',
    LINK_KEY_START_PORT_ID: 3,
    LINK_KEY_START_PORT_FOLDER_NAME: 'Flow',
    LINK_KEY_END_PART_NAME: 'Bell1',
    LINK_KEY_END_PORT_ID: None,
    LINK_KEY_END_PORT_FOLDER_NAME: None}
```

and the value is an instance of *Link*.

Returns

A list of key-value pairs where the key is a *dict* and the value is an *Link*

Return type

`list` of (`dict` , `Link`)

keys()

Gets a new view of the keys for all the links in this collection. A key is a *dict* containing the containing the part name, port number, and port folder name for both the start and end sides of the link, for example:

```
key = {
    LINK_KEY_START_PART_NAME: 'InMan-01',
    LINK_KEY_START_PORT_ID: 3,
    LINK_KEY_START_PORT_FOLDER_NAME: 'Flow',
    LINK_KEY_END_PART_NAME: 'Bell1',
    LINK_KEY_END_PORT_ID: None,
    LINK_KEY_END_PORT_FOLDER_NAME: None}
```

Returns

A list of keys for all the links in this collection

Return type

`list` of `dict`

values()

Gets a list of *Link* instances for all the links in this collection.

Returns

A list of all the links in this collection.

Return type

`list` of `Link`

class **gtisoft.core.links.Link**(*link_wrapper*)

Bases: `object`

A *Link* instance represents and provides methods for interacting with a physical link in a GT application.

set_port_at_part(*part_name*, *port_id*, *port_folder*)

Sets both the port id and port folder for the link at the side connected to the given part.

Parameters

- **part_name** (`str`) – The name of the part connected to the link on the side (‘start’ or ‘end’) that should be changed.
- **port_id** (`int`) – The new port ID to set.
- **port_folder** (`str`) – The name of the port folder to set.

Raises

InvalidPortError – If the port ID and port folder name combination is invalid or if the port cannot be changed for any other reason.

property **end_part_name**

`str` : **Read-Only**; Gets the name of the part at the ‘end’ side of the link.

property **end_port_folder_name**

`str` :

Gets or sets the port folder at the ‘end’ side of the link.

Raises

InvalidPortError – When setting the folder name, if the name is invalid, the id and port folder combination is invalid, or the port is unavailable.

property **end_port_id**

`int` :

Gets or sets the id of the port at the ‘end’ side of the link.

Raises

InvalidPortError – When setting the port ID, if the ID is invalid, the id and port folder combination is invalid, or the port is unavailable.

property **start_part_name**

`str` : **Read-Only**; Gets the name of the part at the 'start' side of the link.

property **start_port_folder_name**

`str` :

Gets or sets the port folder at the 'start' side of the link.

Raises

InvalidPortError – When setting the folder name, if the name is invalid, the id and port folder combination is invalid, or the port is unavailable.

property **start_port_id**

`int` :

Gets or sets the id of the port at the 'start' side of the link.

Raises

InvalidPortError – When setting the port ID, if the ID is invalid, the id and port folder combination is invalid, or the port is unavailable.

---

gtisoft.core.links.create_link_key(*start_part, start_port_id, start_port_folder, end_part, end_port_id, end_port_folder*)

Provides a convenient way to create a key for getting values from a *Links* collection.

The resulting dictionary can be used directly with a *Links* collection:

```
key = create_link_key(
    start_part='InMan-01',
    start_port_id=3,
    start_port_folder='Flow',
    end_part='Bell1',
    end_port_id=None,
    end_port_folder=None)
link = links[key]
```

Or it can also be passed in to methods with *start_part*, *start_port_id*, etc parameters:

```
key = create_link_key(
    start_part='InMan-01',
    start_port_id=3,
    start_port_folder='Flow',
    end_part='Bell1',
    end_port_id=None,
    end_port_folder=None)
link = links.get_link(**key)
```

Parameters

- **start_part** (`str`) – The name of the part at the start of the link.
- **start_port_id** (`int`) – The ID of the port at the start of the link. May be *None* if there is no port at the start of the link.
- **start_port_folder** (`str`) – The name of the port folder at the start of the link. May be *None* if there is no port at the start of the link.
- **end_part** (`str`) – The name of the part at the end of the link.
- **end_port_id** (`int`) – The ID of the port at the end of the link. May be *None* if there is no port at the start of the link.
- **end_port_folder** (`str`) – The name of the port folder at the end of the link. May be *None* if there is no port at the start of the link.

Returns

A dictionary of the form:

```
{LINK_KEY_START_PART_NAME: start_part, LINK_KEY_START_PORT_ID: start_port_id,
LINK_KEY_START_PORT_FOLDER_NAME: start_port_fol LINK_KEY_END_PART_NAME: end_part,
LINK_KEY_END_PORT_ID: end_port_id, LINK_KEY_END_PORT_FOLDER_NAME: end_port_folder}
```

Return type

`dict`

---

gtisoft.core.links.convert_to_link_key(*obj*)

Provides a convenient way of converting a *Link* instance into a dictionary key for getting values from a *Links* collection.

While *get* and *delete* operations can be performed on the *Links* collection using a link directly:

```
>>> # assume `link_1` is an existing instance of `Link`
>>> link_2 = links[link_1]
>>> print(link_1 == link_2)
True
```

```
>>> del links[link_1]
>>> print(link_1 in links)
False
```

the *convert_to_link_key* method may still useful for calling methods that take the analogous arguments using “keyword argument unpacking” with **:

```
>>> # assume `link_1` is an existing instance of `Link`
>>> del links[link_1]
>>> print(link_1 in links)
False
```

```
>>> # re-create the link that was just deleted
>>> links.create_link(**link_1)
>>> print(link_1 in links)
True
```

Parameters

obj (`Link` or `dict`) – An instance of *Link* or a *dict* containing all the keys for a *Links* key (see *LINK_KEY_** constants).

Returns

A dictionary of the form:

```
{LINK_KEY_START_PART_NAME: start_part, LINK_KEY_START_PORT_ID: start_port_id,
LINK_KEY_START_PORT_FOLDER_NAME: start_port_fol LINK_KEY_END_PART_NAME: end_part,
LINK_KEY_END_PORT_ID: end_port_id, LINK_KEY_END_PORT_FOLDER_NAME: end_port_folder}
```

Return type

`dict`

Raises

- **TypeError** – If the *obj* parameter is not a *Link* or *dict* instance.
- **KeyError** – If the *obj* parameter is a *dict* instance but does not contain all of the required keys.

---

gtisoft.core.links.LINK_KEY_START_PART_NAME= *'start_part'*

`str` : Key that maps to the name of the part at the 'start' side of a link. Use to construct a dictionary to use as a key into a *Links* collection.

See `create_link_key()` and `Links` .

---

gtisoft.core.links.LINK_KEY_START_PORT_ID= *'start_port_id'*

`str` : Key that maps to the ID of the port at the 'start' side of a link. Use to construct a dictionary to use as a key into a *Links* collection.

See `create_link_key()` and `Links` .

---

gtisoft.core.links.LINK_KEY_START_PORT_FOLDER_NAME= *'start_port_folder'*

`str`: Key that maps to the name of the port folder at the 'start' side of a link. Use to construct a dictionary to use as a key into a *Links* collection.

See `create_link_key()` and `Links`.

---

gtisoft.core.links.LINK_KEY_END_PART_NAME= *'end_part'*

`str`: Key that maps to the name of the part at the 'end' side of a link. Use to construct a dictionary to use as a key into a *Links* collection.

See `create_link_key()` and `Links`.

---

gtisoft.core.links.LINK_KEY_END_PORT_ID= *'end_port_id'*

`str`: Key that maps to the ID of the port at the 'end' side of a link. Use to construct a dictionary to use as a key into a *Links* collection.

See `create_link_key()` and `Links`.

---

gtisoft.core.links.LINK_KEY_END_PORT_FOLDER_NAME= *'end_port_folder'*

`str`: Key that maps to the name of the port folder at the 'end' side of a link. Use to construct a dictionary to use as a key into a *Links* collection.

See `create_link_key()` and `Links`.

Objects

The *Objects* module contains functions and classes for creating, editing, and deleting the GT-objects in a model.

Module Documentation

exception **gtisoft.core.objects.InvalidObjectNameError**(*name*, *message=None*)

Bases: `Exception`

Raised when an *Object* name is set to an invalid value. An invalid name may be one that contains disallowed characters or is already used by another object from the same template.

Variables

message (`str`) – A message describing the error.

Parameters

message (`str` , optional) – A message describing the error.

exception **gtisoft.core.objects.CreateObjectError**(*message=None*)

Bases: `Exception`

Raised when there is an error attempting to create a new *Object*.

Variables

message (`str`) – A message describing the error.

Parameters

message (`str` , optional) – A message describing the error.

exception **gtisoft.core.objects.DeleteObjectError**(*message=None*)

Bases: `Exception`

Raised when there is an error attempting to delete an *Object* from this collection.

Variables

message (`str`) – A message describing the error.

Parameters

message (`str` , optional) – A message describing the error.

class **gtisoft.core.objects.Objects**(*object_provider_wrapper*)

Bases: `collections.abc.Mapping`

This class is a collection that belongs to a `gtisoft.core.documents.Document` object and contains all of the GT-objects for a model.

Notes

Objects is a `Mapping` that behaves similarly to a Python `dict` . The keys to the mapping are a `dict` containing the GT-object's template name and object name:

```
key = { 'template_name': 'PipeRound', 'object_name': 'PipeRound-1' }
```

and the values are instances of *Object* that represent GT-objects in the model.

The function `create_object_key()` is provided as a convenience for creating the `dict` keys for an *Objects* collection:

```
key = create_object_key('PipeRound', 'PipeRound-1')
```

Many of the `dict` methods are present in the *Objects* class as well as many of the same capabilities.

For instance, checking for the presence of an object in a document can be done using the Python membership operators:

```
>>> key = {'template_name' : 'PipeRound', 'object_name': 'PipeRound-1' }
>>> print(key in objects)
True
```

```
>>> key = {'template_name' : 'PipeRound', 'object_name': 'MissingObject' }
>>> print(key not in objects)
True
```

Objects is "scriptable", enabling getting objects from the model using brackets:

```
key = {'template_name' : 'PipeRound', 'object_name': 'MissingObject' }
obj = objects[key]
```

Objects also allows iterating over all the objects in a model using the default iterator:

```
for key in objects:
    print(key)
```

One difference between a standard `dict` and an *Objects* object is that the *in*, *not in* and subscript operations will accept either the dictionary key or an `Object` instance:

```
>>> key = create_object_key('PipeRound', 'PipeRound-1')
>>> print(key in objects)
True
```

```
>>> obj_1 = objects[key]
>>> print(obj_1 in objects)
True
```

```
>>> obj_2 = objects[obj_1]
>>> print(obj_1 == obj_2)
True
```

See also

Documentation for Python `dict` type.

create_object(*template_name*, *object_name*)

Creates a new *Object* and adds it to this collection (and the document that owns it).

Parameters

- **template_name** (`str`) – The name of the template that the new object should be derived from.
- **object_name** (`str`) – The name to use for the new object. If an object from the same template with the given name already exists, an available name will be chosen based on the given name.

Returns

The newly created *Object*.

Return type

`Object`

Raises

CreateObjectError – If the template with the given name does not exist or the object could not be created for any other reason.

create_object_copy(*other_object*)

Creates a copy of the given *Object* and adds it to this collection (and the document that owns it). The new object's name will be the next available name based on the source object's name. All the attribute values from the source object will be copied to the new object.

Parameters

other_object (`Object`) – The *Object* to copy.

Returns

A newly created *Object* that is a copy of *other_object*.

Return type

`Object`

Raises

- **exceptions.TypeError** – If *other_object* is not an instance of *Object*.
- **CreateObjectError** – If the object could not be created for any other reason.

delete_object(*template_name*, *object_name*)

Deletes an object and all of its child parts from this collection (and the document that owns it).

Parameters

- **template_name** (`str`) – The name of the template for the object to delete.
- **object_name** (`str`) – The name of the object to delete.

Returns

- `Object` – The *Object* instance that was removed from this collection, if an object with the given template name and object name was found.
- *None* – If no object with the given template name and object name exists in this collection.

Raises

DeleteObjectError – I the object or any of its children cannot be deleted for any reason.

find_objects(*template_name=None*, *object_name=None*)

Finds any objects in this collection that match the given criteria.

Parameters

- **template_name** (`str` , optional) – The template name to match. If no template name is provided, it will not be used when performing the search and objects from any template may be returned.
- **object_name** (`str` , optional) – The object name to match. If no object name is provided, it will not be used when performing a search and objects with any name may be returned.

Returns

A list of objects that match the given search criteria. An empty list will be returned if there are no matches.

Return type

`list` of `Object`

get_object(*template_name*, *object_name*)

Gets an instance of *Object* with the given template and object name from this collection.

Parameters

- **template_name** (`str`) – The name of the template of the object to get.
- **object_name** (`str`) – The name of the object to get.

Returns

An instance of *Object* for the GT-object with the given template name and object name.

Return type

`Object`

Raises

exceptions.KeyError – If a GT-object with the given template name and object name is not present in this collection.

items()

Gets a list of key-value pairs for all the GT-objects in this collection where the key is a *dict* containing the GT-object's template name and object name:

```
key = { 'template_name': 'PipeRound', 'object_name': 'PipeRound-1' }
```

and the value is an instance of *Object*.

Returns

A list of key-value pairs where the key is a *dict* and the value is an *Object*

Return type

`list` of (`dict` , `Object`)

keys()

Gets a new view of the keys for all the objects in this collection. A key is a *dict* containing the GT-object's template name and object name:

```
key = { 'template_name': 'PipeRound', 'object_name': 'PipeRound-1' }
```

Returns

A list of keys for all the objects in this collection

Return type

`list` of `dict`

values()

Gets a list of *Object* instances for all the GT-objects in this collection.

Returns

A list of all the GT-objects in this collection.

Return type

`list` of `Object`

---

class **gtisoft.core.objects.Object**(*object_wrapper*)

Bases: `gtisoft.core.objects.GtDocumentResourceMixin`

An *Object* instance represents and provides methods for interacting with a GT-object in a GT application.

property **attribute_values**

`gtisoft.core.attributes.Attributes` : **Read-Only**; The *Attributes* instance that contains the collection of attribute values for the object.

property **object_name**

`str` :

Gets or sets the name of the object.

Raises

InvalidObjectNameError – When setting the name, if the new name is invalid.

property **template_name**

`str`: **Read-Only**; The name of the template that this object is derived from.

---

gtisoft.core.objects.create_object_key(*template_name, object_name*)

Provides a convenient way to create a key for getting values from an *Objects* collection.

The resulting dictionary can be used directly with an *Objects* collection:

```
key = create_object_key('PipeRound', 'PipeRound-1')
obj = objects[key]
```

Or it can also be passed in to methods with named *template_name* and *object_name* parameters:

```
key = create_object_key('PipeRound', 'PipeRound-1')
obj = objects.get_object(**key)
```

Parameters

- **template_name** (`str`) – The name of the template of the object to get.
- **object_name** (`str`) – The name of the object to get.

Returns

A dictionary of the form::

{'template_name': template_name, 'object_name': object_name}

Return type

`dict`

Parts

The *Parts* module contains functions and classes for creating, editing, and deleting the parts in a model.

Module Documentation

exception **gtisoft.core.parts.InvalidPartNameError**(*name*, *message*)

Bases: `Exception`

Raised when a *Part* name is set to an invalid value. An invalid name may be one that contains disallowed characters, is blank, contains or starts with reserved words ('ign' or 'def'), is longer than the maximum allowed number of characters, or is already used by another part in the model.

Variables

message (`str`) – A message describing the error.

Parameters

message (`str` , optional) – A message describing the error.

exception **gtisoft.core.parts.CreatePartError**(*message*)

Bases: `Exception`

Raised when there is an error attempting to create a new *Part*.

Variables

message (`str`) – A message describing the error.

Parameters

message (`str` , optional) – A message describing the error.

exception **gtisoft.core.parts.DeletePartError**(*message*)

Bases: `Exception`

Raised when there is an error attempting to delete a *Part* from this collection.

Variables

message (`str`) – A message describing the error.

Parameters

message (`str`, optional) – A message describing the error.

---

class **gtisoft.core.parts.Parts**(*part_provider_wrapper*)

Bases: `collections.abc.Mapping`

This class is a collection that belongs to a `gtisoft.core.documents.Document` object and contains all of the parts for a model.

Notes

Parts is a `Mapping` that behaves similarly to a Python `dict`. The keys to the mapping are a part's name (type `str`) and the values are instances of *Part* that represent parts in the model.

Many of the `dict` methods are present in the *Parts* class as well as many of the same capabilities.

For instance, checking for the presence of a part in a document can be done using the Python membership operators:

```
print('PipeRound-1' in objects)

print('MissingPart-1' not in objects)
```

Parts is "scriptable", enabling getting parts from the model using brackets:

```
part = parts['PipeRound-1']
```

Parts also allows iterating over all the parts in a model using the default iterator:

```
for part_name in parts:
    print(part_name)
```

One difference between a standard `dict` and a *Parts* object is that the *in*, *not in* and subscript operations will accept either the dictionary key or a `Part` instance:

```
>>> print('PipeRound-1' in parts)
True
```

```
>>> part_1 = parts['PipeRound-1']
>>> print(part_1 in parts)
True
```

```
>>> part_2 = parts[part_1]
>>> print(part_1 == part_2)
True
```

See also

Documentation for Python `dict` type.

create_part(*template_name*, *object_name*, *part_name=None*, *assembly_name=None*, *location_x=None*, *location_y=None*, *icon_index=-1*, *icon_width=None*, *icon_height=None*, *icon_orientation='0'*, *is_icon_inverted=False*)

Creates a new *Part* and adds it to this collection (and the document that owns it).

Parameters

- **template_name** (`str`) – The name of the template that the new part should be derived from.
- **object_name** (`str`) – The name of the object that the new part should be a child of.
- **part_name** (`str` , optional) – The name to use for the new part. If not specified, an available part name will be generated based on the parent object's name.
- **assembly_name** (`str` , optional) –

  The name of the internal assembly to place the new part in. If not specified, the part

  will be placed in the *Main* assembly.

- **location_x** (`int` , optional) – The x-coordinate on the map for the new part, in pixels. The origin (0,0) is the top left of the map and positive X is to the right. If a location is not specified, the next available space will be chosen (left to right, top to bottom).
- **location_y** (`int` , optional) – The y-coordinate on the map for the new part, in pixels. The origin (0,0) is the top left of the map and positive Y is down.
- **icon_index** (`int` , optional) – The index of the desired GT-supplied icon. The default map icon is used by default.
- **icon_width** (`int` , optional) – The width of the new part. Parts are sized relative to the center. Defaults to *None*, which specifies that the original width will be used. Icon widths are expected to be a minimum of 16 pixels.
- **icon_height** (`int` , optional) – The height of the new part. Parts are sized relative to the center. Defaults to *None*, which specifies that the original height will be used. Icon heights are expected to be a minimum of 16 pixels.

- **icon_orientation** (`str` , optional) – The orientation of the icon. Accepted strings are “0”, “90”, “180”, and “270”. Each option is a measure of how much the icon will be rotated clockwise, with “0” being the original, upright position.
- **is_icon_inverted** (`bool` , optional) – If *True*, the icon is inverted. *False* by default.

Returns

The newly created *Part*.

Return type

`Part`

Raises

- **InvalidPartNameError** – If the *part_name* parameter was not *None* and is an invalid name. An invalid name may be one that contains disallowed characters, is blank, contains or starts with reserved words (‘ign’ or ‘def’), is longer than the maximum allowed number of characters, or is already used by another part in the model.
- **CreatePartError** – If the template with the given name does not exist, the parent object with the given name does not exist, or if the part could not be created for any other reason.

create_part_copy(*other_part*)

Creates a copy of the given *Part* and adds it to this collection (and the document that owns it). The new part’s name will be the next available name based on the source part’s name. All attribute part override values from the source part will be copied to the new part.

Parameters

other_part (`Part`) – The *Part* to copy.

Returns

A newly created *Part* that is a copy of *other_part*.

Return type

`Part`

Raises

- **exceptions.ValueError** – If *other_part* is not an instance of *Part*.
- **CreatePartError** – If the part could not be created for any other reason.

delete_part(*part_name*)

Deletes a part from this collection (and the document that owns it).

Parameters

part_name (`str`) – The name of the part to delete.

Returns

- `Part` – The *Part* instance that was removed from this collection, if a part with the given name was found.
- *None* – If no part with the given name exists in this collection.

Raises

DeletePartError – If the part with the given name exists but cannot be deleted for any reason.

find_parts(*template_name=None*, *object_name=None*, *part_name=None*)

Finds any parts in this collection that match the given criteria.

Parameters

- **template_name** (`str` , optional) – The template name to match. If no template name is provided, parts from any template may be returned.
- **object_name** (`str` , optional) – The object name to match. If no object name is provided, parts from any object may be returned.
- **part_name** (`str` , optional) – The part name to match. If no part name is provided, parts with any name may be returned.

Returns

A list of parts that match the given search criteria.

Return type

`list` of `Part`

get_part(*part_name*)

Gets an instance of *Part* with the given name from this collection.

Parameters

part_name (`str`) – The name of the part to get.

Returns

An instance of *Part* for the part with the given name.

Return type

`Part`

Raises

exceptions.KeyError – If a part with the given name is not present in this collection.

items()

Gets a list of key-value pairs for all the parts in this collection where the key is a part name as a *str* and the value is an instance of *Part*.

Returns

A list of key-value pairs where the key is a *str* part name and the value is a *Part*.

Return type

`list` of (`str`, `Part`)

keys()

Gets a new view of the keys for all the parts in this collection. A key is the part's name

as a *str*.

Returns

A list of keys for all the parts in this collection.

Return type

`list` of `str`

values()

Gets a list of *Part* instances for all the parts in this collection.

Returns

A list of all the parts in this collection.

Return type

`list` of `Part`

class **gtisoft.core.parts.Part**(*part_wrapper*)

Bases: `object`

A *Part* instance represents and provides methods for interacting with a part in a GT application.

property **assembly_name**

`str`: **Read-Only**; The name of the internal assembly that this part is in.

property **attribute_override_values**

`gtisoft.core.attributes.Attributes`: **Read-Only**; The *Attributes* instance that contains the collection of attribute part override values.

property **attribute_values**

`gtisoft.core.attributes.Attributes`: **Read-Only**; The *Attributes* instance that contains the collection of attribute values for the part's parent object.

property **icon_height**

`int`:

The height of the part, used to resize the icon associated with the part. Icons are sized relative to the center. Defaults to *None*, which specifies that the original height will be used. Icon heights are expected to be a minimum of 16 pixels.

Raises

exceptions.ValueError – If icon is set to be less than 16 pixels high. Icon height must be 16 pixels high or larger.

property **icon_index**

`int`:

The index of the part's GT-supplied icon. 0 is the default map icon.

Raises

exceptions.ValueError – If icon index is not within range of available icons. Icon indexes are 0-based.

property **icon_orientation**

`str`:

The orientation of the icon. Accepted strings are "0", "90", "180", and "270". Each option is a measure of how much the icon will be rotated clockwise, with "0" being the original, upright position. Note that icons are rotated counterclockwise when inverted.

Raises

exceptions.ValueError – If an argument other than "0", "90", "180", or "270" are sent in.

property **icon_width**

`int`:

The width of the part, used to resize the icon associated with the part. Icons are sized relative to the center. Defaults to *None*, which specifies that the original width will be used. Icon widths are expected to be a minimum of 16 pixels.

Raises

exceptions.ValueError – If icon is set to be less than 16 pixels wide. Icon width must be 16 pixels wide or larger.

property **is_icon_inverted**

`bool`: If *True*, the icon is inverted. *False* by default.

property **location_x**

`int`: **Read-Only**; The x-coordinate of the part, in pixels. The origin (0,0) is the top left of the map and positive X is to the right.

property **location_y**

`int`: **Read-Only**; The y-coordinate of the part, in pixels. The origin (0,0) is the top left of the map and positive Y is down.

property **object_name**

`str`: **Read-Only**; The name of the part's parent *Object*.

property **part_name**

`str`:

Gets or sets the name of the part.

Raises

InvalidPartNameError – When setting the name, if the new name is invalid.

property **plots**

`gtisoft.core.plots.Plots`: **Read-Only**; The *Plots* instance that contains the collection of plots, plot requests, and plot request properties for the part.

property **template_name**

`str`: **Read-Only**; The name of the template that this part is derived from.

Plots

The *Plot* module contains functions and classes turning plot requests on and off and setting plot request properties for parts.

Module Documentation

class **gtisoft.core.plots.Plots**(*plot_provider_wrapper*)

Bases: `collections.abc.Mapping`

This class is a collection that belongs to a `gtisoft.core.parts.Part` and contains the available plots and their settings.

Notes

Plots is a `Mapping` that behaves similarly to a Python `dict`. The keys to the mapping are the plots' short name (type `str`) and the values are instances of *Plot*.

Many of the `dict` methods are present in the *Plots* class as well as many of the same capabilities.

For instance, checking for the presence of a plot in a part can be done using the Python membership operators:

```
>>> # Check if the part has an available plot called 'RPM'
>>> print('RPM' in cranktrain_part.plots)
True
```

```
>>> print('MISSING' not in cranktrain_part.plots)
True
```

Plots is "scriptable", enabling getting plots from a part using brackets:

```
>>> plot = cranktrain_part.plots['RPM']
>>> print(plot)
Plot(plot_name=u'RPM', is_on=False, location=None, max_plot_points=None, plot_range=None,
x_axis=None, sampling_type=None, sampling_interval=None)
```

Plots also allows iterating over all the available plots in a part using the default iterator:

```
>>> for plot_name in cranktrain_part.plots:
>>>     print(plot_name)
RPM
TORQ
TORQ2
...
```

See also

Documentation for Python `dict` type.

items()

Gets a list of key-value pairs for all the plots in this collection where the key is a plot short name as a *str* and the value is an instance of *Plot'*.

Returns

A list of key-value pairs where the key is a *str* plot short name and the value is a *Plot*.

Return type

`list` of (`str` , `Plot`)

keys()

Gets a new view of the keys for all the plots in this collection. A key is a plot short name as a *str*.

Returns

A list of keys for all the plots in this collection.

Return type

`list` of `str`

values()

Gets a list of *Plot* instances for all the plots in this collection.

Returns

A list of all the plots in this collection.

Return type

`list` of `Plot`

class **gtisoft.core.plots.Plot**(*value_wrapper*)

Bases: `object`

A *Plot* instance represents and provides methods for interacting with a plot, plot request, and plot request properties for a part in a GT application.

property **is_on**

`bool`: Gets or sets the flag indicating if the plot is active or not. If a plot is active, it will result in a request that the plot be stored when the model is run.

property **location**

`str`:

Gets or sets the *Location* property for the plot.

Notes

This property must be filled in when the data curves are from parts that are discretized into subvolumes (i.e. *Pipe** parts). The positions are normalized so that `'0.0'` selects the inlet end of the pipe:

```
pipe.plots['pres'].location = '0.0'
```

and `'1.0'` selects the outlet:

```
pipe.plots['pres'].location = '0.0'
```

Multiple locations can be selected by adding a blank space between the values. For example, the inlet and outlet may be selected:

```
pipe.plots['pres'].location = '0.0 1.0'
```

A value of `'def'` or `None` will reset the property to the default value, which is specified in 'Plot Setup'.

Raises

SetPlotPropertyError – When setting the *location*, if the given location is invalid, if the plot does not support the location property, or the property cannot be set for any other reason.

property **max_plot_points**

`int`:

Gets or sets the *Max Plot Points* property for the plot.

Notes

The *Max Plot Points* property specifies the maximum number of points in each data set to be plotted. If the maximum number is reached, the data will be thinned to stay at or below the maximum using the algorithm specified in *Output Setup > Data_Storage folder > Thinning Method for Limiting Large Arrays*…. The global and individual plot maximum limits are 40,000 points.

A value of `'def'` or `None` will reset the property to the default value, which is specified in 'Plot Setup'.

Since this property is an `int`, a value of `None` will be returned when the value in the UI is `'def'`.

Raises

SetPlotPropertyError – When setting the *max_plot_points*, if the given value is invalid, out of range, or the property cannot be set for any other reason.

property **plot_name**

`str`: **Read-Only**; The name of the plot.

property **plot_range**

`str`:

Gets or sets the *Plot Range* property for the plot.

Notes

The *Plot Range* property selects the range of values to be plotted. Valid values are:

- `'LastCycle(s)'` or `'LAST_CYCLES'` - indicates that the data points will only be plotted for the last cycle of the simulation. `'LastCycle(s)'` can be used only with periodic simulations.
- `'EntireRun'` or `'ENTIRE_RUN'` - indicates that the data points will be plotted for the entire duration of the simulation.
- `'Automatic'` or `'AUTOMATIC'` - indicates that the solver will select the plot range most appropriate for the data that is being requested. Typically `'Automatic'` for a non-control object will use `'LastCycle(s)'` as the plot range and a control object will use `'EntireRun'`.

A value of `'def'` or `None` will reset the property to the default value, which is specified in 'Plot Setup'.

Raises

SetPlotPropertyError – When setting the *plot_range*, if the value is not valid, or if the property cannot be set for any other reason.

property **sampling_interval**

`str` :

Gets or sets the *Sampling Interval* property for the plot.

Notes

The *Sampling Interval* property selects the interval between which consecutive points are plotted. For example:

- If the *Sampling Type* is set to `'Degrees'` and the *Sampling Interval* is set to `'5'`, the desired value will be plotted once every 5 degrees.
- If `'Timestep'` is selected for *Sampling Type*, the *Sampling Interval* must be a positive integer.
- If *Sampling Interval* is set to `'def'`, the *Sampling Type* must also be set to `'def'`.

A value of `'def'` or `None` will reset the property to the default value, which is specified in 'Plot Setup'.

Raises

SetPlotPropertyError – When setting the *sampling_interval*, if the value is not valid, or if the property cannot be set for any other reason.

property **sampling_type**

`str` :

Gets or sets the *Sampling Type* property for the plot.

Notes

The *Sampling Type* property selects how the data points are sampled. Valid values are:

- `'Timestep'`
- `'Degrees'` - can only be used with periodic simulations.
- `'Seconds'`

A value of `'def'` or `None` will reset the property to the default value, which is specified in 'Plot Setup'.

If *sampling_type* is set to `'def'`, the *sampling_interval* must also be set to `'def'`.

Raises

SetPlotPropertyError – When setting the *sampling_type*, if the value is not valid, or if the property cannot be set for any other reason.

property **x_axis**

`str` :

Gets or sets the *X-Axis* property for the plot.

Notes

The *X-Axis* property selects the x-axis for the data points to be plotted against. Valid values are:

- `'CrankAngle'` - indicates that the x-axis will be set to crank angles in units of degrees. `'CrankAngle'` can be used only with periodic simulations.
- `'Time'` - indicates that the x-axis will be set to time in units of seconds.
- `'Automatic'` - indicates that the solver will select the x-axis most appropriate for the data that is being requested. Typically `'Automatic'` for a non-control object will use `'CrankAngle'` as the x-axis and a control object will use `'Time'`. If `'Automatic'` is selected while *Plot Range* is set to `'EntireRun'`, the x-axis will be set to `'Time'`.

A value of `'def'` or `None` will reset the property to the default value, which is specified in 'Plot Setup'.

Raises

SetPlotPropertyError – When setting the *x_axis*, if the value is not valid, or if the property cannot be set for any other reason.

Signals

The *Signals* module contains functions and classes for creating, editing, and deleting signal and controls links between parts in a model.

Module Documentation

exception **gtisoft.core.signals.CreateSignalError**(*message*)

Bases: `Exception`

Raised when there is an error attempting to create a signal link between parts.

Variables

message (`str`) – A message describing the error.

Parameters

message (`str` , optional) – A message describing the error.

exception **gtisoft.core.signals.DeleteSignalError**(*message*)

Bases: `Exception`

Raised when there is an error attempting to delete a *Signal* from this collection.

Variables

message (`str`) – A message describing the error.

Parameters

message (`str` , optional) – A message describing the error.

exception **gtisoft.core.signals.InvalidSignalIdError**(*message*)

Bases: `Exception`

Raised when there is an error attempting to change the signal ID on one side of a *Signal*.

Variables

message (`str`) – A message describing the error.

Parameters

message (`str` , optional) – A message describing the error.

class **gtisoft.core.signals.Signals**(*signal_provider_wrapper*)

Bases: `collections.abc.Mapping`

This class is a collection that belongs to a `gtisoft.core.documents.Document` object and contains all of the signal links for a model.

Notes

Signals is a `Mapping` that behaves similarly to a Python `dict`. The keys to the mapping are a `dict` containing the part name and signal ID for both the start and end sides of the signal link:

```
key = {SIGNAL_SIGNAL_KEY_START_PART_NAME: start_part,
    SIGNAL_KEY_START_SIGNAL_ID: start_signal_id,
    SIGNAL_KEY_SIGNAL_KEY_END_PART_NAME: end_part,
    SIGNAL_KEY_END_SIGNAL_ID: end_signal_id}
```

and the values are instances of *Signal* that represent a signal link in the model.

The *SIGNAL_KEY_** string constants are provided at the top level of the module and can be used to construct a dictionary with the proper keys to use as a key in the *Signals* collection.

The function `create_signal_key()` is provided as a convenience for creating the `dict` keys for an *Signals* collection:

```
key = create_signal_key(
    start_part='Int-Vel',
    start_signal_id=None,
    end_part='Int-Mic',
    end_signal_id=1)
```

Many of the `dict` methods are present in the *Signals* class as well as many of the same capabilities.

For instance, checking for the presence of a signal link in a document can be done using the Python membership operators:

```
>>>  key = create_signal_key(
>>>     start_part='Int-Vel',
>>>     start_signal_id=None,
>>>     end_part='Int-Mic',
>>>     end_signal_id=1)
>>> print(key in signals)
True
```

```
>>>  key = create_signal_key(
>>>     start_part='Missing-Part-1',
>>>     start_signal_id=None,
>>>     end_part='Missing-Part-1',
>>>     end_signal_id=1)
>>> print(key not in links)
True
```

Signals is “scriptable”, enabling getting signal links from the model using brackets:

```
key = create_signal_key(
    start_part='Int-Vel',
    start_signal_id=None,
    end_part='Int-Mic',
    end_signal_id=1)
signal = signals[key]
```

Signals also allows iterating over all the signal links in a model using the default iterator:

```
for key in signals:
    print(key)
```

One difference between a standard `dict` and a *Signals* object is that the *in*, *not in* and subscript operations will accept either the dictionary key or a `Signal` instance:

```
>>>  key = create_signal_key(
>>>     start_part='Int-Vel',
>>>     start_signal_id=None,
>>>     end_part='Int-Mic',
>>>     end_signal_id=1)
>>> print(key in signals)
True
```

```
>>> signal_1 = signals[key]
>>> print(signal_1 in signals)
True
```

```
>>> signal_2 = signals[signal_1]
>>> print(signal_1 == signal_2)
True
```

> **See also**
>
> Documentation for Python `dict` type.

`create_signal`(*start_part*, *start_signal_id*, *end_part*, *end_signal_id*)

Creates a signal link between two parts in the model and adds the resulting *Signal* instances to this collection (and the document that owns it).

Parameters

- **start_part** (`str`) – The name of the part at the start of the signal link.
- **start_signal_id** (`int`) – The ID of the signal at the start of the signal link. May be 'None' if there is no signal at the start of the link (for instance a signal link starting at a *SensorConn* part).
- **end_part** (`str`) – The name of the part at the end of the signal link.
- **end_signal_id** (`int`) – The ID of the signal at the end of the signal link. May be 'None' if there is no signal at the end of the link (for instance a signal link terminating at a *SensorConn* part).

Returns

A list of the newly created *Signal* instances.

Return type

`list` of `Signal`

Raises

CreateSignalError – If the signal link could not be created for any reason.

`delete_signal`(*start_part*, *start_signal_id*, *end_part*, *end_signal_id*)

Deletes the signal link that exactly matches the given parameters.

Parameters

- **start_part** (`str`) – The name of the part at the start of the signal link.
- **start_signal_id** (`int`) – The ID of the signal at the start of the signal link. May be 'None' if there is no signal at the start of the link (for instance a signal link starting at a *SensorConn* part).
- **end_part** (`str`) – The name of the part at the end of the signal link.
- **end_signal_id** (`int`) – The ID of the signal at the end of the signal link. May be 'None' if there is no signal at the end of the link (for instance a signal link terminating at a *SensorConn* part).

Returns

- `Signal` – A *Signal* instance representing the deleted signal link, if one was removed. The returneed *Signal* is read-only.
- *None* – If no signal link with the given parameters exists in this collection.

Raises

DeleteSignalError – If the signal link can not be deleted for any reason.

find_signals(*start_part=None*, *start_signal_id=None*, *end_part=None*, *end_signal_id=None*)

Finds any signal links in this collection that match the given criteria. All parameters are optional and any parameter that is not provided or is *None* will be ignored when performing the search.

Parameters

- **start_part** (`str` , optional) – The name of the part at the start of the signal link.
- **start_signal_id** (`int` , optional) – The ID of the signal at the start of the signal link.
- **end_part** (`str` , optional) – The name of the part at the end of the link.
- **end_signal_id** (`int` , optional) – The ID of the signal at the end of the signal link.

Returns

A list of Signals that match the given search criteria. An empty list will be returned if there are no matches.

Return type

`list` of `Signal`

get_signal(*start_part*, *start_signal_id*, *end_part*, *end_signal_id*)

Gets an instance of 'Signal' that exactly matches the given parameters.

Parameters

- **start_part** (`str`) – The name of the part at the start of the signal link.
- **start_signal_id** (`int`) – The ID of the signal at the start of the signal link. May be 'None' if there is no signal at the start of the link (for instance a signal link starting at a *SensorConn* part).
- **end_part** (`str`) – The name of the part at the end of the signal link.
- **end_signal_id** (`int`) – The ID of the signal at the end of the signal link. May be 'None' if there is no signal at the end of the link (for instance a signal link terminating at a *SensorConn* part).

Returns

An instance of *Signal* for the signal link that matches the given parameters.

Return type

`Signal`

Raises

excepctions.KeyError – If no signal link with the given parameters exists in this collection.

items()

Gets a list of key-value pairs for all the signal links in this collection where the key is

a *dict* containing the part name and signal ID for both the start and end sides of the signal link, for example:

```
key = {
    SIGNAL_SIGNAL_KEY_START_PART_NAME: 'Int-Vel',
    SIGNAL_KEY_START_SIGNAL_ID: None,
    SIGNAL_KEY_SIGNAL_KEY_END_PART_NAME: 'Int-Mic',
    SIGNAL_KEY_END_SIGNAL_ID: 1}
```

and the value is an instance of *Signal*.

Returns

A list of key-value pairs where the key is a *dict* and the value is a *Signal*

Return type

`list` of (`dict` , `Signal`)

keys()

Gets a new view of the keys for all the signal links in this collection. A key is a *dict* containing the containing the part name and signal id for both the start and end sides of the signal link, for example:

```
key = {
    SIGNAL_SIGNAL_KEY_START_PART_NAME: 'Int-Vel',
    SIGNAL_KEY_START_SIGNAL_ID: None,
    SIGNAL_KEY_SIGNAL_KEY_END_PART_NAME: 'Int-Mic',
    SIGNAL_KEY_END_SIGNAL_ID: 1}
```

Returns

A list of keys for all the signal links in this collection

Return type

`list` of `dict`

values()

Gets a list of *Signal* instances for all the signal links in this collection.

Returns

A list of all the signal links in this collection.

Return type

`list` of `Signal`

---

class **gtisoft.core.signals.Signal**(*signal_wrapper*)

Bases: `object`

A *Signal* instance represents and provides methods for interacting with a signals or controls link in a GT application.

set_signal_id_at_part(*part_name*, *signal_id*)

Sets the signal id for the signal link at the side connected to the given part.

Parameters

- **part_name** (`str`) – The name of the part connected to the signal link on the side ('start' or 'end') that should be changed.
- **signal_id** (`int`) – The new signal ID to set.

Raises

InvalidSignalIdError – If the signal ID is invalid or if the signal ID cannot be changed for any other reason.

property **attribute_values**

`gtisoft.core.attributes.Attributes` : **Read-Only**; The *Attributes* instance that contains the collection of attribute values for the signal link.

property **end_part_name**

`str` : **Read-Only**; Gets the name of the part at the 'end' side of the signal link.

property **end_signal_id**

`int` :

Gets or sets the id of the signal at the 'end' side of the signal link.

Raises

InvalidSignalIdError – When setting the signal ID, if the ID is invalid.

property **end_signal_unit**

`Unit` :

Gets or sets the unit for the signal at the 'end' side of the signal link. The setter accepts either a single instance of *Unit* or a *list* of *Unit* to facilitate using the `gtisoft.core.units.UNITS` collection:

```
# Using a single unit:
a_unit = signal_a.end_signal_unit
signal_b.end_signal_unit = a_unit

# Passing in a list using the UNIT collection:
signal_c.end_signal_unit = UNITS['rad']
```

Raises

InvalidSignalUnitError – When setting the signal Unit, if the Unit is invalid

property **start_part_name**

`str` : **Read-Only**; Gets the name of the part at the 'start' side of the signal link.

property **start_signal_id**

`int` :

Gets or sets the id of the signal at the 'start' side of the signal link.

Raises

InvalidSignalIdError – When setting the signal ID, if the ID is invalid.

property **start_signal_unit**

`Unit` :

Gets or sets the unit for the signal at the 'start' side of the signal link. The setter accepts either a single instance of *Unit* or a *list* of *Unit* to facilitate using a `gtisoft.core.units.Units` collection:

```
# Using a single unit:
a_unit = signal_a.start_signal_unit
signal_b.start_signal_unit = a_unit

# Passing in a list using the UNIT collection:
signal_c.start_signal_unit = UNITS['rad']
```

Raises

InvalidSignalUnitError – When setting the signal Unit, if the Unit is invalid

---

gtisoft.core.signals.create_signal_key(*start_part, start_signal_id, end_part, end_signal_id*)

Provides a convenient way to create a key for getting values from a *Signals* collection.

The resulting dictionary can be used directly with a *Signals* collection:

```
key = create_signal_key(
    start_part='Int-Vel',
    start_signal_id=None,
    end_part='Int-Mic',
    end_signal_id=1)
signal = signals[key]
```

Or it can also be passed in to methods with *start_part*, *start_signal_id*, etc parameters:

```
key = create_signal_key(
    start_part='Int-Vel',
    start_signal_id=None,
    end_part='Int-Mic',
    end_signal_id=1)
signal = signals.get_signal(**key)
```

Note that calling the method using named parameters as in the above examples adds additional clarity but is not strictly necessary:

```
key = create_signal_key('Int-Vel', None, 'Int-Mic', 1)
```

Parameters

- **start_part** (`str`) – The name of the part at the start of the signal link.
- **start_signal_id** (`int`) – The ID of the signal at the start of the signal link. May be 'None' if there is no signal at the start of the link (for instance a signal link starting at a *SensorConn* part).
- **end_part** (`str`) – The name of the part at the end of the signal link.
- **end_signal_id** (`int`) – The ID of the signal at the end of the signal link. May be 'None' if there is no signal at the end of the link (for instance a signal link terminating at a *SensorConn* part).

Returns

A dictionary of the form:

```
{SIGNAL_SIGNAL_KEY_START_PART_NAME: start_part,
SIGNAL_KEY_START_SIGNAL_ID: start_signal_id,
SIGNAL_KEY_SIGNAL_KEY_END_PART_NAME: end_part,
SIGNAL_KEY_END_SIGNAL_ID: end_signal_id}
```

Return type

`dict`

---

gtisoft.core.signals.convert_to_signal_key(*obj*)

Provides a convenient way of converting a *Signal* instance into a dictionary key for getting values from a *Signal* collection.

While *get* and *delete* operations can be performed on the *Signal* collection using a signal directly:

```
>>> # assume `signal_1` is an existing instance of `Signal`
>>> signal_2 = signals[signal_1]
>>> print(signal_1 == signal_2)
True
```

```
>>> del signals[signal_1]
>>> print(signal_1  in signals)
False
```

the *convert_to_signal_key* method may still useful for calling methods that take the analogous arguments using “keyword argument unpacking” with **:

```
>>> # assume `signal_1` is an existing instance of `Signal`
>>> del signals[signal_1]
>>> print(signal_1 in signals)
False
```

```
>>> # re-create the signal that was just deleted
>>> signals.create_signal(**signal_1)
>>> print(signal_1 in signals)
True
```

Parameters

obj (`Signal` or `dict`) – An instance of *Signal* or a *dict* containing all the keys for a *Signal* key (see *SIGNAL_KEY_** constants).

Returns

A dictionary of the form:

```
{SIGNAL_SIGNAL_KEY_START_PART_NAME: start_part,
SIGNAL_KEY_START_SIGNAL_ID: start_signal_id,
SIGNAL_KEY_SIGNAL_KEY_END_PART_NAME: end_part,
SIGNAL_KEY_END_SIGNAL_ID: end_signal_id}
```

Return type

`dict`

Raises

- **TypeError** – If the *obj* parameter is not a *Signal* or *dict* instance.
- **KeyError** – If the *obj* parameter is a *dict* instance but does not contain all of the required keys.

---

gtisoft.core.signals.SIGNAL_KEY_START_PART_NAME= *'start_part'*

`str`: Key that maps to the name of the part at the 'start' side of a signal link. Use to construct a dictionary to use as a key into a *Signals* collection.

See `create_signal_key()` and `Signals`.

---

gtisoft.core.signals.SIGNAL_KEY_START_SIGNAL_ID= *'start_signal_id'*

`str`: Key that maps to the ID of the signal at the 'start' side of a signal link. Use to construct a dictionary to use as a key into a *Signals* collection.

See `create_signal_key()` and `Signals`.

---

gtisoft.core.signals.SIGNAL_KEY_END_PART_NAME= *'end_part'*

`str`: Key that maps to the name of the part at the 'end' side of a signal link. Use to construct a dictionary to use as a key into a *Signals* collection.

See `create_signal_key()` and `Signals`.

---

gtisoft.core.signals.SIGNAL_KEY_END_SIGNAL_ID= *'end_signal_id'*

`str`: Key that maps to the ID of the signal at the 'end' side of a signal link. Use to construct a dictionary to use as a key into a *Signals* collection.

See `create_signal_key()` and `Signals`.

Simulations

The *Simulations* module contains methods and classes for running local simulations.

Module Documentation

class **gtisoft.core.simulations.SimulationManager**(*simulation_manager_wrapper*)

Bases: `object`

The *SimulationManager* object provides methods for creating .dat files and running local simulations.

create_dat_file(*document*, ***kwargs*)

Creates a .dat file for the document.

Notes

This will save any unsaved changes in the document, effectively overwriting it, and also overwrite any existing .dat file for the model without warning.

Parameters

- **document** (`gtisoft.core.documents.Document`) – The document for which to create the .dat file.
- ****kwargs** – A set of keyword arguments specifying the options for creating the .dat file. See below.

Keyword Arguments

- **implicitConflict** (`str`) – Valid values: *update*, *keep*, *fail*. Defines what action to take if an implicit object in the model differs from the parent implicit object in the Object Library. If the model had been run from GT-ISE, a dialog would ask what to do with the items which are in conflict. The default behavior if no argument is given is “keep”.
- **SAObjectConflict** (`str`) – Valid values: *keep*, *override*, *rename*, *fail*. Defines what action to take if an object in an external subassembly has the same name as an object in the main model but has differences in its data. If the model was run from GT-ISE, a dialog would ask what to do with the items which are in conflict. The default behavior is to KEEP the existing objects.

- **doe** (`str`) – Valid values: *on*, *off*. For model files that have a DOE configured, this argument gives options to activate the DOE cases (*on*) or run ignore the DOE and run the values in Case Setup (*off*). If this argument is omitted, the default setting is *on*.

> **Raises**

- **gtisoft.core.documents.DocumentClosedError** – If the *document* argument is a *Document* instance that is not currently open.
- **exceptions.RuntimeError** – If the .dat file could not be created for any other reason.

`run_simulation`(*document*, ***kwargs*)

Starts running a local simulation.

> **Parameters**

- **document** (`gtisoft.core.documents.Document`) – The document to run.
- ****kwargs** – A set of keyword arguments specifying the options for the simulation. See below.

> **Keyword Arguments**

- **p** (`str`) – Valid values: *on*, *off*, *fmu*. Specifies whether the run is a preprocess run. Default=`off`, which means regular, *on`= preprocess only*, *`fmu* is a special preprocess run for creating *.fmu files.
- **g** (`str`) – Valid values: *on*, *off*. Specifies the location of the solver output. If *on* the output will be displayed in the Simulation Dashboard. If *off* the the output will be displayed in the console window.
- **m** (`str`) – Valid values: *on*, *off*. Specifies whether the monitors should be enabled.
- **implicitConflict** (`str`) – Valid values: *update*, *keep*, *fail*. Defines what action to take if an implicit object in the model differs from the parent implicit object in the Object Library. If the model had been run from GT-ISE, a dialog would ask what to do with the items which are in conflict. The default behavior if no argument is given is “keep”.
- **SAObjectConflict** (`str`) – Valid values: *keep*, *override*, *rename*, *fail*. Defines what action to take if an object in an external subassembly has the same name as an object in the main model but has differences in its data. If the model was run from GT-ISE, a dialog would ask what to do with the items which are in conflict. The default behavior is to KEEP the existing objects.

- **doe** (`str`) – Valid values: *on*, *off*. For model files that have a DOE configured, this argument gives options to activate the DOE cases (*on*) or run ignore the DOE and run the values in Case Setup (*off*). If this argument is omitted, the default setting is *on*.

Raises

- **gtisoft.core.documents.DocumentClosedError** – If the *document* argument is a *Document* instance that is not currently open.
- **exceptions.RuntimeError** – If the simulation could not be started for any other reason.

Units

The *Units* module contains dictionaries that provide access to all the valid units and unit categories for GT Applications.

Module Documentation

class **gtisoft.core.units.Unit**(*unit_wrapper*)

Bases: `object`

Represents the unit of a physical quantity.

property **gtt_id**

`int` : **Read-Only**; The database (gtt) ID that is used by GT Applications to refer to this unit.

property **multiplier**

`float` : **Read-Only**; The multiplier to convert the unit to the base unit of the category.

property **unit_category**

`str` : **Read-Only**; The upper-case, human-readable name of this unit's parent category. Corresponds to the *category_name* attribute in the *UnitCategory* class.

property **unit_name**

`str` : **Read-Only**; The name of this unit (e.g. *mm* or *m/s*).

class **gtisoft.core.units.UnitCategory**(*wrapper*)

Bases: `collections.abc.Sequence`

A *UnitCategory* is a sequence of units that all represent the same type of physical quantity, for instance *DISTANCE* contains the units *mm*, *m*, *ft*, and others.

property **category_name**

`str` : **Read-Only**; An upper-case, human-readable name for this unit category.

property **gtt_id**

`int`: **Read-Only**; The database (gtt) ID that is used by GT Applications to refer to this unit category.

---

class **gtisoft.core.units.Units**(*wrapper*)

Bases: `collections.abc.Mapping`

A dictionary that contains all valid units for GT Applications. The keys to the dictionary are `str` that correspond to the unit's names, and the values are `list` of `Unit` instances that have that name. Values are a list of units instead of a single instance because it is possible for multiple units to share the same name as long as they belong to different categories. For example, 'deg' and 'rad' are used by both the 'ANGLE' and 'ANGLE 3D' categories:

```
>>> units = Application.get_instance().gtt.units
>>> rads = units['rad']
>>> print(rads)
[Unit(gtt_id=62, name='rad', unit_category='ANGLE', multiplier=57.2957795130823),
Unit(gtt_id=525, name='rad', unit_category='ANGLE 3D', multiplier=57.2957795130823)]
```

get_unit(*unit_name: str*) → gtisoft.core.units.Unit

Gets an instance of *Unit* with the given name from this collection.

Parameters

unit_name (`str`) – The name of the unit to get.

Returns

A list of units with the given name. Return value is a list instead of a single instance because it is possible for multiple units to share the same name as long as they belong to different categories. For example, 'deg' and 'rad' are used by both the 'ANGLE' and 'ANGLE 3D' categories:

Return type

`Unit`

Raises

exceptions.KeyError – If a unit with the given name is not present in this collection.

get_unit_by_id(*gtt_id: int*) → Optional[gtisoft.core.units.Unit]

Gets the *Unit* instance with the given gtt_id from this collection.

Parameters

gtt_id (`int`) – The ID of the unit to get.

Returns

- `Unit` – If a *Unit* with the given ID was found.
- *None* – If no *Unit* with the given ID exists in this collection.

get_unit_by_name_category(*unit_name: str*, *category_name: str*)

Gets an instance of *Unit* with the given name and category from this collection.

Parameters

- **unit_name** (`str`) – The name of the unit to get.
- **category_name** (`str`) – The name of the category the unit is in.

Returns

The unit with the given name and category.

Return type

`Unit`

Raises

exceptions.KeyError – If a unit with the given name and category is not present in this collection.

items()

Gets a list of key-value pairs for all the units in this collection where the key is a unit name as a *str* and the value is an instance of *Unit*.

Returns

A list of key-value pairs where the key is a *str* unit name and the value is a *Unit*.

Return type

`list` of (`str` , `Unit`)

keys() → List[str]

Gets a new view of the keys for all the units in this collection. A key is the unit's name

as a *str*.

Returns

A list of keys for all the units in this collection.

Return type

`list` of `str`

values() → List[gtisoft.core.units.Unit]

Gets a list of *Unit* instances for all the parts in this collection.

Returns

A list of all the units in this collection.

Return type

`list` of `Unit`

---

class **gtisoft.core.units.UnitCategories**(*wrapper*)

Bases: `collections.abc.Mapping`

A dictionary that contains all valid unit categories for GT Applications. The keys to the dictionary are `str` that corresponds to the upper-case, human-readable name for a category, and the values are instances of `UnitCategory`. For example:

```
>>> categories = Application.get_instance().gtt.unit_categories
>>> cat = categories['DISTANCE'] # get the category for distance, using its string name
>>> print(cat.gtt_id, cat.category_name, cat.units)
DISTANCE 1 [Unit(gtt_id=4, name='in', unit_category='DISTANCE', multiplier=0.0254),
Unit(gtt_id=5, name='ft', unit_category='DISTANCE', multiplier=0.3048), Unit(gtt_id=3,
name='mm', unit_category='DISTANCE', multiplier=0.001), Unit(gtt_id=394, name='dm',
unit_category='DISTANCE', multiplier=0.1), Unit(gtt_id=277, name='km',
unit_category='DISTANCE', multiplier=1000), Unit(gtt_id=345, name='mi',
unit_category='DISTANCE', multiplier=1609.33), Unit(gtt_id=1, name='m',
unit_category='DISTANCE', multiplier=1), Unit(gtt_id=648, name='nm',
unit_category='DISTANCE', multiplier=1e-09), Unit(gtt_id=6, name='micron',
unit_category='DISTANCE', multiplier=1e-06), Unit(gtt_id=2, name='cm',
unit_category='DISTANCE', multiplier=0.01)]
```

get_unit_category(*category_name: str*) → gtisoft.core.units.UnitCategory

Gets an instance of *UnitCategory* with the given name from this collection.

Parameters

category_name (`str`) – The name of the unit category to get.

Returns

An instance of *UnitCategory* for the category with the given name.

Return type

`UnitCategory`

Raises

exceptions.KeyError – If a category with the given name is not present in this collection.

items()

Gets a list of key-value pairs for all the unit categories in this collection where the key is a category name as a *str* and the value is an instance of *UnitCategory*.

Returns

A list of key-value pairs where the key is a *str* unit name and the value is a *UnitCategory*.

Return type

`list` of (`str` , `UnitCategory`)

keys() → List[str]

Gets a new view of the keys for all the unit categories in this collection. A key is the unit category's name as a *str*.

Returns

A list of keys for all the unit categories in this collection.

Return type

`list` of `str`

values() → List[gtisoft.core.units.UnitCategory]

Gets a list of *UnitCategory* instances for all the parts in this collection.

Returns

A list of all the unit categories in this collection.

Return type

`list` of `UnitCategory`

Version

The *Version* module provides methods to query version information for models and applications.

Module Documentation

class **gtisoft.core.version.GTVersion**(*version_wrapper*)

Bases: `object`

An object that contains version information for a document or GT Application.

> *property* **build_number**
>
> `str` : ***Read-Only**; this version's build number (e.g. 1 in 2020.1)

> *property* **version_number**
>
> `str` : ***Read-Only**; this version's number (e.g. 2020 in 2020.1)

1. Build Spring Mass Damper

This example will go through some of the basics of building a model in GT-ISE using the GT Python API and covers the following topics:

- Opening a model in GT-ISE
- Creating new objects
- Defining attribute values
- Placing parts on the map
- Creating links between parts
- Modifying Run Setup
- Saving the resulting model

The files associated with this example can be found in **/examples/z_Interfaces_and_Co-Simulation_UserCode/GT_Automation_Python_Scripting_API/01-Build_Spring_Mass_Damper**.

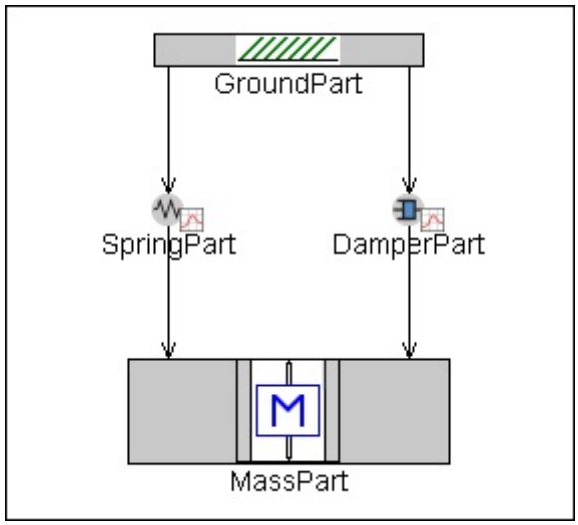
Note

The Python script featured in this example is designed to be run from inside the built-in Python editor in GT-ISE. For a version that can be run external to GT-ISE, please see example **4. External API Access**.

Overview

The purpose of this example is to show how a model can be built from scratch using various commands within the GT Python API. The script will ultimately result in a simple mass-spring-damper model containing four components:

- Mass
- Spring
- Damper
- Ground



A blank starter model and example Python script can be found in the example directory.

Initialization

This section is present at the beginning of each of the example Python scripts. The first line of the script simply defines the model name that it will operate on: Start_Model.gtm.

```
1   modelname = 'Start_Model.gtm'
```

The next line of the script connects the Python interpreter with the GT application within which the script will operate. This instance is saved to the variable *app* so that it can be referred to later for other application-level commands. When running this script from the built-in Python editor in GT-ISE, it is necessarily implied that it will operate within the same instance of GT-ISE, so all arguments for *get_instance()* can be omitted (they are ignored if provided).

This is followed by an *open_document()* call to open the starter model defined above. This model resides in the same working directory as the script, so the full absolute path is not needed. Similar to *get_instance()*, this is saved to the variable *doc* so that it can be referred to later.

```
1   app = Application.get_instance()
2   doc = app.documents.open_document(modelname)
```

Note

In addition to opening models, the *open_document()* method can also be used to evolve models and dependencies from previous versions. By default, opening a model from a previous version will result in a Python error unless other arguments are specified. For more information, please see the **API Documentation > Documents**.

Creating a New Object

New objects can be created in a model using the *documents.objects.create_object(template, object)* method. This command has two required aruments: the template to be used, and the desired name for the resulting object.

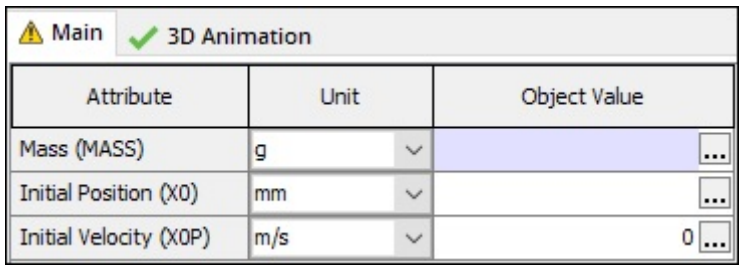
```
1   massobj = doc.objects.create_object('Mass', 'MassObj')
```

Since it is common to perform other actions to an object after creating it, such as defining attribute values, it is generally desirable to save/return the object to a variable (*massobj* in the above example) so that it can be referenced later.

Defining Attribute Values

When defining values for specific attributes in an object, attributes should generally be referred to by their “short names”, which uniquely identify each attribute within a given object. This contrasts with the attribute names normally displayed in the object editor (also known as the “long name”), as these may be duplicated folder to folder and therefore may not be unique.

Attribute short names are displayed in the object editor (in parentheses after the attribute long name) any time the built-in Python script editor is open in GT-ISE. It can also be displayed at all times using the setting in File > Options > Python. Please see **Getting Started > Python in GT-ISE > Python Developer Info** for more information.



The *object.attribute_values* collection can be accessed using square brackets, i.e. *object.attribute_values[‘short_name’]*. This can be combined with the *set_value(value, unit)* method to define the attribute value and unit for a given attribute in a single line:

```
1   massobj.attribute_values['MASS'].set_value('0.1', UNITS['kg'])
2   massobj.attribute_values['X0'].set_value('0', UNITS['mm'])
3   massobj.attribute_values['X0P'].set_value('1', UNITS['m/s'])
```

Note

Units[] (and similarly, *Unit_Categories[]*) is a dictionary that contains all valid units for GT Applications, and allows convenient specification of unit by name. It is a saved instance of *Application.gtt.units* (set up automatically when running from the built-in Python script editor in GT-ISE, and optionally when accessed externally) and should generally be used as a lookup for input into any unit arguments.

Warning

Though the unit argument in the *set_value()* method is optional, it is strongly recommended that units always be included when defining a numeric value. If omitted, the currently set units will be used, which may produce unexpected results if they do not match with the intended units.

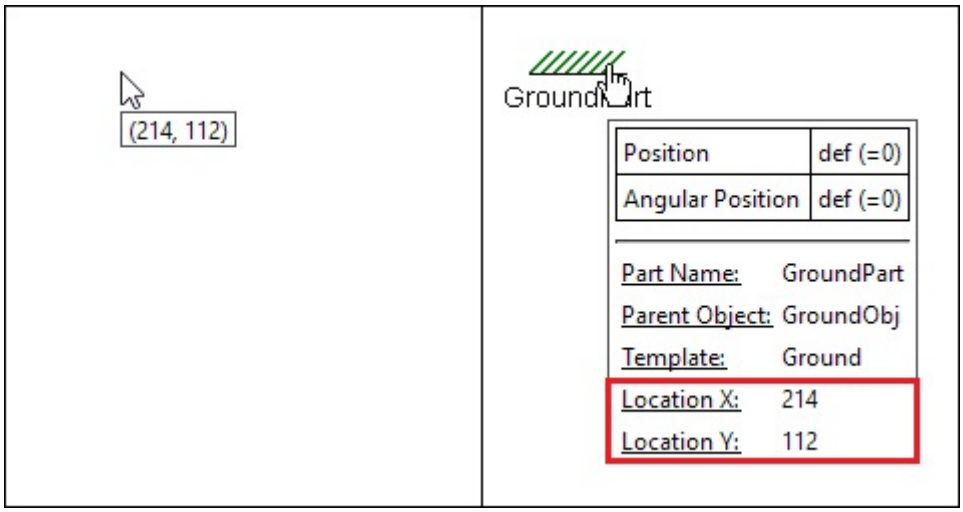
Placing Parts On The Map

With the objects defined, the next step is to place parts on the map. This can be done using the *documents.parts.create_part(template, object, part)* method:

```
groundpart = doc.parts.create_part('Ground', 'GroundObj', 'GroundPart', location_x=240,
location_y=120)
springpart = doc.parts.create_part('Spring', 'SpringObj', 'SpringPart', location_x=180,
location_y=200)
damperpart = doc.parts.create_part('Damper', 'DamperObj', 'DamperPart', location_x=300,
location_y=200)
masspart = doc.parts.create_part('Mass', 'MassObj', 'MassPart', location_x=240,
location_y=300)
```

The *create_part()* method has three required arguments, the source template name, the source object name, and the desired part name. Note that all part names must be unique for a given model.

There are also three optional arguments for the target internal subassembly (if applicable), and for the X and Y location on the map where the part will be placed. To aid in determining part locations, a tool tip with the current X/Y position of the cursor is displayed any time the built-in Python script editor is open in GT-ISE. Additionally, the X/Y position of existing parts on the map is displayed in the tool tip when hovering over parts.



Note

If the X/Y location argument is omitted, the part will be placed in the next available space on the map (left to right, top to bottom). However, in most cases, it is desirable to explicitly define the X/Y location so that the model map is laid out in a logical way.

Linking Parts Together

Links can be created between parts using the *documents.links.create_link()* method. This method has six required arguments: the part name where the link starts, the port ID number on the start part, the port folder name on the start part, the part name where the link ends, the port ID number on the end part, and the port folder name on the end part.

Since most links in GT-SUITE have a directionality associated with them, it is generally important that the start part and end part are defined in the correct order so that the link is pointing in the correct direction.

```
link1 = doc.links.create_link('GroundPart', None, None, 'SpringPart', None, None)
link2 = doc.links.create_link('SpringPart', None, None, 'MassPart', None, None)
link3 = doc.links.create_link('GroundPart', None, None, 'DamperPart', None, None)
link4 = doc.links.create_link('DamperPart', None, None, 'MassPart', None, None)
```

Some templates in GT-ISE do not require specific port ID numbers/folders, typically if there is only one type of port or if it does not affect modeling. This is the case for the parts above, where all of the links are 1D-mechanical connections that do not necessarily need to go to a specific port. In these cases, *None* can be specified for the port ID and folder.

Note

When creating links in the GUI, certain connection parts are generated automatically when linking between two components directly. This is also true when creating links from Python. The *create_link()* method will return a list of two links in these scenarios.

Modifying Run Setup

The Setup menus (Run, Output, Advanced) in GT-ISE are treated similarly to standard objects, except they are accessed at the *document* level, i.e. *documents.run_setup*. Attributes in these setup menus are accessed in the same way as standard objects, via short name using the *attribute_values* collection.

```
doc.run_setup.attribute_values['DURATN2'].is_radio_on = True
doc.run_setup.attribute_values['DURATN2'].set_value('1', UNITS['s'])
```

Notice that the *DURATN2* attribute has both a radio button and numeric attribute value associated with it. Radio buttons can be set using the separate attribute property *is_radio_on = True*. This will automatically de-select any current radio selections in the group, as would be the case in the GUI.

Saving Models

Models can be saved using the *documents.save()* or *documents.save_as(filepath)* methods. In this example, the completed model is saved in the same example directory as the original model, with a new file name **Spring_Mass_Damper_System_From_Python.gtm**.

```
doc.save_as('Spring_Mass_Damper_System_From_Python.gtm')
```

2. Case Setup Parameters

This example will go through modifying plots in GT-ISE and starting a simulation using the GT Python API and covers the following topics:

- Retrieving existing object data from the model and printing it to the output console
- Creating a new parameter in Case Setup
- Appending new cases to Case Setup
- Assigning case values for a parameter
- Defining case labels
- Turning cases on/off
- Using a parameter in an attribute

The files associated with this example can be found in **/examples/z_Interfaces_and_Co-Simulation_UserCode/GT_Automation_Python_Scripting_API/02-Case_Setup_Parameters**.

Note

The Python script featured in this example is designed to be run from inside the built-in Python editor in GT-ISE. To modify it for running externally to GT-ISE, please see the documentation section **Getting Started > External API Access > External Script Initialization** or refer to example **4. External API Access**.

Initialization

This section is present at the beginning of each of the example Python scripts. The first line of the script simply defines the model name that it will operate on: Start_Model.gtm.

```
1  modelname = 'Start_Model.gtm'
```

The next line of the script connects the Python interpreter with the GT application within which the script will operate. This instance is saved to the variable *app* so that it can be referred to later for other application-level commands. When running this script from the built-in Python editor in GT-ISE, it is necessarily implied that it will operate within the same instance of GT-ISE, so all arguments for *get_instance()* can be omitted (they are ignored if provided).

This is followed by an *open_document()* call to open the starter model defined above. This model resides in the same working directory as the script, so the full absolute path is not needed. Similar to *get_instance()*, this is saved to the variable *doc* so that it can be referred to later.

```
app = Application.get_instance()
doc = app.documents.open_document(modelname)
```

Note

In addition to opening models, the *open_document()* method can also be used to evolve models and dependencies from previous versions. By default, opening a model from a previous version will result in a Python error unless other arguments are specified. For more information, please see **API Documentation > Documents**.

Retrieving Object Data

To retrieve an object that already exists in a model, the *objects.get_object(template, object)* method can be used. As in previous examples, it is generally desirable to save/return the object to a variable (*massobj* in the below example) so that it can be referenced later.

```
massobj = doc.objects.get_object('Mass', 'MassObj')
```

Object attribute values can be retrieved using the *get_value()* method, which accesses the *object.attribute_values* collection similarly to the *set_value()* method covered in the previous example.

```
mass = massobj.attribute_values['MASS'].get_value()
```

Note

The *get_value()* method is designed generically for both single value attributes as well as table/matrix attributes. It can therefore take two optional arguments for row and column position to extract any value out of a table. By default, it takes *row=0* and *column=0*, which also corresponds to the position for single value attributes. The *get_row()* and *get_column()* methods can also be used to extract entire rows and columns of data, respectively.

The *attribute_values* class has several useful properties, such as *long_name* and *unit*, which can also be accessed as follows (see **API Documentation > Attributes** for a full list of properties):

```
mass_attr_name = massobj.attribute_values['MASS'].long_name
mass_unit = massobj.attribute_values['MASS'].unit.unit_name
```

Printing Output

Information extracted from the GT Python API can be printed to the output console using the standard Python *print()* command. In this example, the Mass object name, mass, and initial velocity are printed:

```
print('Current values for object: ', massobj.object_name)
print('-----------------------------------------')
print(mass_attr_name, '=', mass, mass_unit)
print(x0p_attr_name, '=', x0p, x0p_unit)
```

Creating a New Parameter

Similar to the other Setup menus, Case Setup is accessed at the *documents* level. To create a new parameter, the *case_setup.parameters.create_parameter()* method can be used. This method has four required arguments: the parameter name, parameter description, Case Setup folder where the parameter will be placed, and the parameter unit.

```
mass_param = doc.case_setup.parameters.create_parameter('mass', 'Part Mass', 'Main',
UNITS['kg'])
```

Appending Cases

Cases can be added to Case Setup using the *case_setup.run_cases.append_case()* method. This method simply appends a single new case and does not take any arguments. To add multiple cases at one time, a *for* loop can be used. The example script shows three cases being added:

```
for i in range(0,3):
        doc.case_setup.run_cases.append_case()
```

Cases can also be inserted using the *case_setup.run_cases.insert_case(index)* method. This method takes one argument for the case index before which the new case will be inserted.

Assigning Case Values

Parameter case values can be assigned using the *parameters.set_value(index, value)* method. This method takes two arguments, the index of the case being set, and the value corresponding to that index. To assign multiple case values at one time, a *for* loop can be used. The example script first defines the case values as a list, then loops over them to assign values to the mass parameter for all four cases:

```
mass_param_values = [0.1, 0.2, 0.5, 1.0]
for i in range(0,4):
        mass_param.set_value(i, mass_param_values[i])
```

Warning

When referring to cases from the GT Python API, a **base-0** index value is always used. This differs from the case numbers shown in the GUI, which is **base-1**. Therefore, the index value will always be one less than the case value, i.e. to refer to case #2 as shown in the GUI, *index=1* should be used.

Note

Similar to Case Setup in the GT-ISE GUI, if any case values are not defined via the Python API, the previous case value is propagated forward automatically. For example, if the model had six cases and the mass parameter only had values for cases 1-4 defined, the value for case #4 would be propagated to the remaining two cases.

Defining Case Labels

Case labels can be defined via the *run_cases.label* property. Similar to attributes, specific cases can be referred to by index within the *run_cases* collection.

```
doc.case_setup.run_cases[0].label = 'Mass = [mass]kg'
```

Similar to the GUI, formulas and parameters may be used, and labels are automatically propagated to all cases unless otherwise defined.

Turning Cases On/Off

Cases can be turned on and off by setting the *run_cases.is_on* property to *True* or *False*. Specific cases can be referred to by index within the *run_cases* collection. The example script turns off cases #2 and #3, leaving cases #1 and #4 on to be run.

```
doc.case_setup.run_cases[1].is_on = False
doc.case_setup.run_cases[2].is_on = False
```

Using Parameters in Attributes

Parameters can be used in object attributes using the standard *attribute_values.set_value()* method. Parameters should be passed in as a string (using 'quotes') along with the traditional square bracket [parameter] notation used in GT-SUITE. Since parameter units are defined from Case Setup and not at the attribute level, the unit argument may be omitted when assigning a parameter to an attribute.

```
massobj.attribute_values['MASS'].set_value('[mass]')
```

Note

In this example, the parameter was first *declared* in Case Setup, then used in an object attribute. Similar to the GUI, parameters can also be defined from object attributes directly, at which point they will be added to Case Setup automatically, with the unit associated with the attribute.

3. Plots and Run Simulation

This example will go through some of the basics of building a model in GT-ISE using the GT Python API and covers the following topics:

- Turning plots on/off
- Setting plot properties
- Starting a simulation

The files associated with this example can be found in **/examples/z_Interfaces_and_Co-Simulation_UserCode/GT_Automation_Python_Scripting_API/03-Plots_And_Run_Simulation**.

> **Note**
>
> The Python script featured in this example is designed to be run from inside the built-in Python editor in GT-ISE. To modify it for running externally to GT-ISE, please see the documentation section **Getting Started > External API Access > External Script Initialization** or refer to example **4. External API Access**.

Initialization

This section is present at the beginning of each of the example Python scripts. The first line of the script simply defines the model name that it will operate on: Start_Model.gtm.

```
1  modelname = 'Start_Model.gtm'
```

The next line of the script connects the Python interpreter with the GT application within which the script will operate. This instance is saved to the variable *app* so that it can be referred to later for other application-level commands. When running this script from the built-in Python editor in GT-ISE, it is necessarily implied that it will operate within the same instance of GT-ISE, so all arguments for *get_instance()* can be omitted (they are ignored if provided).

This is followed by an *open_document()* call to open the starter model defined above. This model resides in the same working directory as the script, so the full absolute path is not needed. Similar to *get_instance()*, this is saved to the variable *doc* so that it can be referred to later.

```
app = Application.get_instance()
doc = app.documents.open_document(modelname)
```

Note

In addition to opening models, the *open_document()* method can also be used to evolve models and dependencies from previous versions. By default, opening a model from a previous version will result in a Python error unless other arguments are specified. For more information, please see **API Documentation > Documents**.

Retrieving Parts

To retrieve a part that already exists in a model, the *parts.get_part(part_name)* method can be used. As in previous examples, it is generally desirable to save/return the part to a variable (*masspart* and *springpart* in the below example) so that it can be referenced later.

```
masspart = doc.parts.get_part('MassPart')
springpart = doc.parts.get_part('SpringPart')
```

Turning Plots On/Off

Similar to object attributes, part plots can be accessed via the *documents.parts.plots* collection by plot short name. Plots can be turned on/off by setting the *plots.is_on* property to *True* or *False*.

```
springpart.plots['FRCX'].is_on = True
```

A *for* loop can be used to set the *is_on* property for all plots within a given part. The example below turns on all available plots for the *MassPart* part:

```
for plot in masspart.plots.values():
        plot.is_on = True
```

Warning

For very large models or ones containing many cases, it is generally recommended that only the plots required for a given analysis be turned on, as they do contribute to both simulation runtime as well as results file size.

Modifying Plot Properties

Plots have additional properties, such as location, sampling type, x-axis type, etc. that can be set via the API. The example below shows how the spring's Force plot can be set to have a sampling type and interval of 0.05 seconds. More information about plot properties can be found in **API Documentation > Plots**.

```
springpart.plots['FRCX'].sampling_type = 'Seconds'
springpart.plots['FRCX'].sampling_interval = 0.05
```

Running a Simulation

Simulations can be started with the GT Python API at the application level using the *Application.simulations.run_simulations()* method. This method has one required argument, the document for which the simulation will be run, and several keyword arguments that control simulation options. The keyword arguments are similar to the options available when running a GT-SUITE simulation from command line.

The example script passes in the current document object (Start_Model.gtm) along with one keyword argument, *g=on*, for turning the Simulation Dashboard GUI on:

```
app.simulations.run_simulation(doc, g="on")
```

Note

Before starting a simulation, the document must be opened and passed into *run_simulations()* as an object.

4. External API Access

This example is functionally identical to example **1. Build Spring Mass Damper**, however it has been set up for accessing the GT Python API from an external Python installation. The following topics are covered:

- Initializing the script for external API access
- Using the *get_instance()* command to connect to an existing gateway, or start a new headless instance of the gateway.

The files associated with this example can be found in **/examples/z_Interfaces_and_Co-Simulation_UserCode/GT_Automation_Python_Scripting_API/04-External_API_Access**.

For more information on setting up an external Python installation for use with the GT Python API, please see the section **Getting Started > External API Access**.

Note

The Python script featured in this example can be run both from inside the built-in Python editor in GT-ISE as well as externally.

Initialization

The initialization step of this example script is the only section that differs from the example **1. Build Spring Mass Damper**, and largely follows the steps described in **Getting Started > External API Access > External Script Initialization**.

The first few lines of the script add a file from the GT-SUITE installation into the Python path so that the GT Python API library can be imported. *GTIHOME* is an OS-level environment variable corresponding to the GT-SUITE installation location. By retrieving this variable from the OS, we can automatically find and set the correct directory for the necessary file, independent of where GT-SUITE was installed. After this, the *Application* module can be imported.

```
import os
import sys

gtihome = os.environ['GTIHOME']
sys.path.append(os.path.abspath(gtihome + '/v2022/GTsuite/ext/gtsuite.jar'))

from gtisoft.core.application import Application
```

The next line attaches the external Python interpreter to a GT-SUITE application using the *Application.get_instance()* method:

```
app = Application.get_instance(port=25333, version='-v2022')
```

The resulting behavior of this line will differ depending on a few factors:

- If the GT-ISE GUI is open and the API gateway is enabled at port:25333 (default), it will connect to that GUI instance. The remainder of the script will be executed within the GT-ISE GUI.
- If a headless API gateway has already been started from command line at port:25333, it will connect to that headless instance and execute the remainder of the script.
- If no API gateway can be found at port:25333, a headless instance will automatically be created. It will then connect to the new headless instance and execute the remainder of the script.

More information about the API gateway and *get_instance()* arguments can be found in **Getting Started > External API Access**.

The final two lines of this section save an instance of the *units* and *unit_categories* mappings, which contain all valid units and unit categories available in GT-SUITE. This step is optional, but allows for more convenient access to units later in the script.

```
UNITS = app.gtt.units
UNIT_CATEGORIES = app.gtt.unit_categories
```

Note

The built-in Python editor in GT-ISE automatically performs this step before running any scripts, so it is not necessary for scripts designed to run only inside GT-ISE. However, it may be helpful to include it for scripts that may also be run externally.

Please refer to example **1. Build Spring Mass Damper** for a description of the remainder of the example script.

v2020

Notable changes to the GT-SUITE API and related features are listed below. Please see the relevant sections of the documentation for more detailed information, or contact support@gtisoft.com.

RC

Initial release of the GT-SUITE Python API.

Build 1

What's New:

- The GT-SUITE Python API can now be utilized externally to GT-ISE without the GUI open via a new 'headless gateway' option.
- An option to display Python Developer Info in the GT-ISE GUI was added.
- The *open_document()* method can now be used to evolve models and dependencies from previous versions.
- The units argument when using the *attribute_values.set_value()* method is now optional, as it serves no purpose in certain cases, such as when using a parameter. However, it is strongly recommended that a unit always be specified when working with numeric values.
- The Python Script Editor was enabled in the Autolion GUI mode of GT-ISE.

Bugfixes:

- Parameters defined directly inside of an attribute using the *attribute_values.set_value()* method were not appearing in Case Setup immediately.
- Using the *create_parameter()* method with *None* specified for the parameter description was preventing the Value Selector from opening thereafter. *None* is now converted to an empty string.
- Specifying a parameter in an attribute with user-definable "dynamic units" was not propagating the Case Setup units to the attribute until after the model was closed and reopened.
- The *parameters.set_solver_unit()* method was incorrectly resetting the display unit to be the same as the solver unit in Case Setup when used. The display unit will now remain consistent unless the unit category is changed.
- The part location coordinates used by the *parts.create_part()* and *parts.get_part()* were using different conventions (icon center vs. corner), resulting in conflicting values.

- The *document.close()* method previously allowed models to be closed despite having dialog windows open in the GUI, causing those windows to be orphaned. This situation will now result in a Python error.
- Python error messages displayed in the Python Script Editor output panel were reporting incorrect line numbers.
- Python error messages and standard output were both displayed in black text in the Python Script Editor. Error messages are now shown in red text.

Build 2

What's New:

- A setting has been added to File > Options for pointing GT-ISE to other Python environments. Tools have also been included for creating and managing Python Virtual Environments, which can be helpful for installing additional Python packages.
- Models can now be set up to run an associated Python script each time a simulation is launched. This Python script is specified via an attribute in the Advanced Setup menu of any model.
- The User Shortcuts panel in File > Resources can now be displayed as a toolbar tab, and now supports Python scripts. Clicking on a Python script in this panel executes it automatically inside GT-ISE.

Bugfixes:

- When using the *objects.create_object_copy()* method, radio button and checkbox attribute values were not being copied to the new object.
- The following methods related to units erroneously included an *Application.get_instance()* call within the API: *attribute_values.unit* and *solver_unit*, *parameter.unit* and *solver_unit*, *signal.start_signal_unit* and *end_signal_unit*. This resulted in additional GT-ISE instances to be spawned when accessing these methods externally to GT-ISE, causing some scripts to fail. Scripts run inside the GT-ISE GUI were unaffected.
- Setting *plots.sampling_interval = 'def'* was not resulting in any change to the attribute. Both *'def'* and *None* are now accepted.

Build 3

What's New:

- The GTPythonExternal license is no longer required to access the GT Python API from outside the GT GUI applications. All usage of the GT Python API is now unified under the GTAutomation license for v2020 build 3, and v2021 build 1 and onward.
- When working with parts via the GT Python API, the *location_x* and *location_y* properties now refer to the center of the part icon instead of the upper-left corner. This was changed to make part placement more consistent, independent of the part's icon size.

Bugfixes:

- Stability of GTAutomation license checkout has been improved, particularly when connecting to a license server over a network with high latency (such as over a VPN connection).
- Opening the File > Options menu sometimes resulted in an unanticipated error due to settings for Python Virtual Environments.

v2021

Notable changes to the GT-SUITE API and related features are listed below. Please see the relevant sections of the documentation for more detailed information, or contact support@gtisoft.com.

RC

What's New:

- The GT-SUITE Python API has been extended to GT-POST, where it can be used to query instantaneous plots and RLTs, extract result data, and create custom report (.gu) files.
- Models can now be set up to run an associated Python script after each time the simulation runs. This Python script is specified via an attribute in the Run Setup menu of any model. The attribute for running a Python script before the simulation has also been moved to the Run Setup menu.
- The GT-SUITE Python API documentation is now available in the **/documents/Graphical_Applications/GT-Automation_Python_API** folder of the GT-SUITE installation for access outside of the built-in Python editor.
- Properties for controlling the icon, size, and orientation of parts on the GT-ISE map have been added. These properties can also be set when using the *create_part()* method.
- Methods have been added for the following actions in GT-ISE:
    - Querying and deleting unused objects/templates from a model
    - Changing a document's license type
    - Creating a new blank document (.gtm, .gtsub, etc.)
    - Querying the version of an application instance or a document
    - GT-POWER-xRT Advisor

Bugfixes:

- Models with the Design Optimizer turned on can now be run as a distributed simulation via the Python API. This previously resulted in running the simulation locally as a serial run.

Build 1

What's New:

- Explicit plots and datasets can now be created from scratch inside report (.gu) files via the GT Python API.
- Result files with the .gdx file extension can now be opened via the *documents.open_document()* method. Doing so will automatically create a converted copy of the result file in the new .glx file format.
- The GTPythonExternal license is no longer required to access the GT Python API from outside the GT GUI applications. All usage of the GT Python API is now unified under the GTAutomation license for v2020 build 3, and v2021 build 1 and onward.
- The current working directory of the Python script now gets passed into the GT Python API, allowing relative paths to be used when working with GT model files.
- A *force* flag has been added to the *document_tools.delete_unused_objects()* method for deleting objects that can be used without any dependency.

Bugfixes:

- The *attribute_values.get_value()* method was always returning an error when used in a "setup" object, i.e. run_setup, output_setup, advanced_setup, and design_optimizer.
- In some cases, while accessing the GT Python API externally, closing a document via the *documents.close()* method was resulting in an error.
- When running inside the built-in Python Script Editor, certain types of Python errors caused an incorrect line number to be shown in the error message.
- For attributes where both the solver and display unit can be user-defined (such as in data reference objects), certain combinations of the *set_value()* and *set_solver_unit()* methods were causing the unit to be reset to 'No Unit'.
- The *multiplier* property of units inside the *gtt.units* module was erroneously returning the unit ID number instead.
- Handling of parts inside internal subassemblies when using the *document_tools.combine_flow_volume_to_*()* methods has been improved.

Build 2

Bugfixes:

- Launching a headless API gateway for GT-POST via command line was incorrectly checking out a GUI license in addition to the GT-Automation license. The headless API gateway for GT-ISE was not affected.
- In certain situations, using the *combine_flow_volume_to_*()* methods was causing GUI dialogs to show, prompting the user for confirmation. These dialogs are now suppressed.
- The entity filtering methods in GT-POST have been improved to better handle plot group names that do not fall under the standard Template:Part format.

Build 3

What's New

- Units for GT-POST results are now more accessible via dedicated properties and methods in the plot/dataset classes for retrieving and setting units.

v2022

Notable changes to the GT-SUITE API and related features are listed below. Please see the relevant sections of the documentation for more detailed information, or contact support@gtisoft.com.

RC

What's New:

- The built-in Python editor now has an auto-indent feature to match indentation of previous lines, or add indentation after certain lines like if statements, for loops, etc. As part of this feature, the editor can also detect whether tabs or spaces have been used and convert between them if needed.
- The API documentation tab in the built-in Python editor now has a find (ctrl-f) feature for text search.
- New methods have been added to the plot and dataset classes in GT-POST for querying and setting units.
- When data is printed using the *data.formatted* method for datasets in GT-POST, units are now included in the column headers for reference.

Bugfixes:

- Deleting parts via Python script would occasionally cause unanticipated errors to occur inside of GT-ISE.
- When executing a Python script for the first time for a given application instance, if a license could not be checked out successfully, all subsequent attemps would also fail to check out a license even if one became available.
- Attempting to launch a headless instance of GT-POST (without GUI) and open a result (.glx) file was incorrectly leading to a "No valid license" error, even if a license was available.
- Launching a simulation with the Simulation Dashboard GUI turned off (*g='off'*) and simulation monitors turned on (*m='on'*) was leading to a startup delay due to this being an incompatible combination. Monitors will now automatically be turned off by default if the Simulation Dashboard is turned off.

Build 1

What's New:

- The GT-SUITE Python API is now available in GEM3D/COOL3D for editing Case Setup and object attributes.
- A new function has been added to the document tools module for automating updates of the ProfileGPSRoute reference object for GT-RealDrive.
- Process Map (.gtprocess) files can now be opened and edited via the Python API.

Bugfixes:

- When changing the port of an existing signal link via Python script, the signal was not updating with the port, causing unit conflicts in some situations.
- Templates that only allow 'def' objects could not be used via Python unless the 'def' object was already present in the model.
- Printing *documents* would sometimes return an empty list when called from an external Python interpreter. This could also result in a *keyerror* due to object library (.gto) files not being supported via the API.

Build 2

Bugfixes:

- The "Stop" button in the built-in Python editor for halting a running script was not working.
- New, unsaved documents were not appearing in the *Application.documents* list until after they were saved to disk.
- Existing report (*.gu) files could not be opened in GT-POST via the Python API due to an *IndexOutOfBoundsException* error.
- Creating new groups and separators within a report (*.gu) file via the Python API was causing GT-POST to freeze.

Build 3

Bugfixes:

- The *parts.create_part_copy()* was not functioning and would always result in an error.
- Default 'def' library objects could not be used via the Python API unless the Template Library was open in the GT-ISE GUI.