POWERFACTORY

Scripting with Python in *PowerFactory*

This is a step-by-step tutorial to get you started in writing and running Python scripts. Before going through this tutorial, you should already have a basic grasp of *PowerFactory* handling and be familiar in performing tasks such as load flows and short circuit calculations. Furthermore the licence for scripting and automation is necessary to perform this tutorial.

Python is not directly installed on the host computer by installing *PowerFactory*. This means you have to install Python separately before starting this tutorial. Details on how to install Python may be found in the User Manual, section Installation of a Python Interpreter.

Tutorial overview

A variety of small examples including code snippets are shown for you to understand basic *PowerFactory* functions such as working with parameters, navigating folders, executing calculations and plotting results. More advanced topics are then introduced. At the end of this tutorial additional exercises are given to test your Python skills.

Possible code solutions to the examples and exercises are shown in this tutorial document. The tutorial will focus mainly on *PowerFactory* specific scripting methods. Before starting this tutorial, basic Python syntax such as if-conditions, loops, working with lists and strings should be known.

1 Introduction to Scripting with Python

1.1 Import project

Firstly, we will import a small sample project as a basis for the first exercises. Click on the icon , to import and activate the project for this exercise.

POWER SYSTEM SOLUTIONS
MADE IN GERMANY

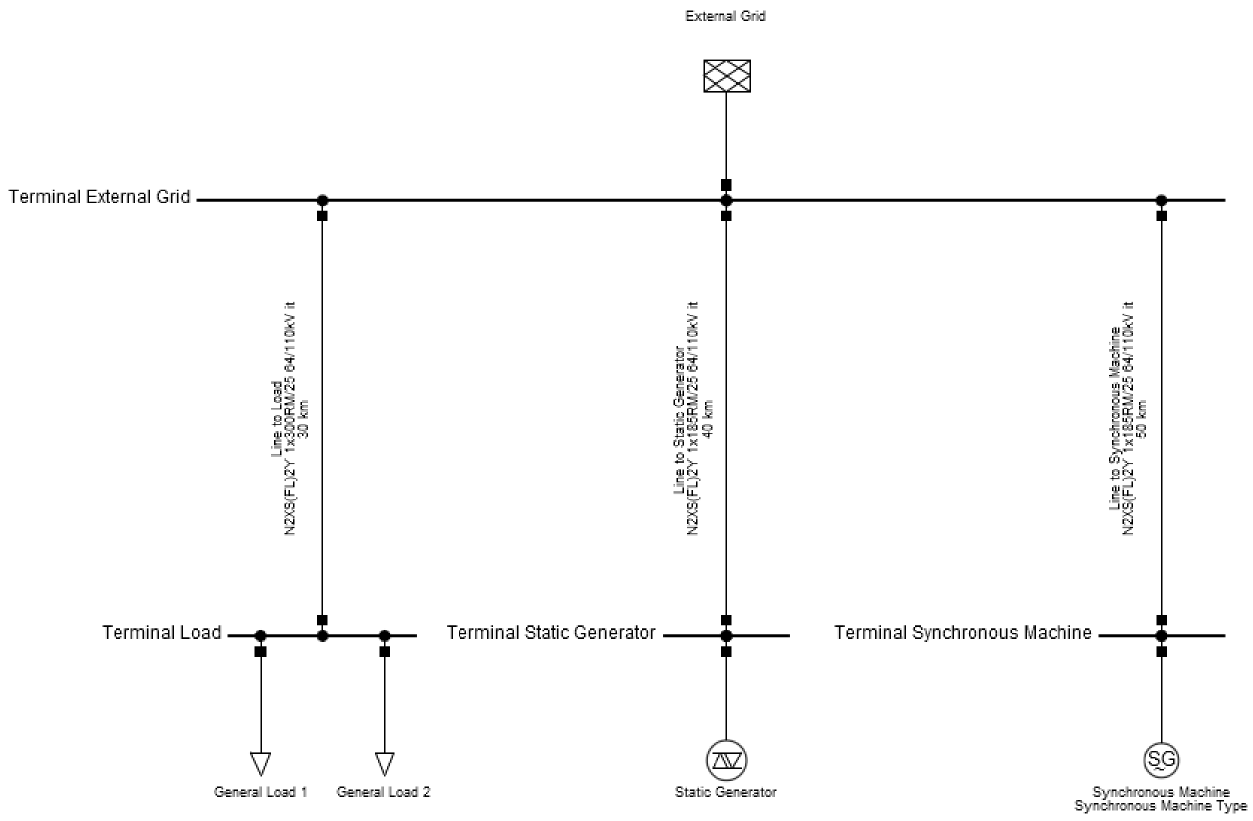
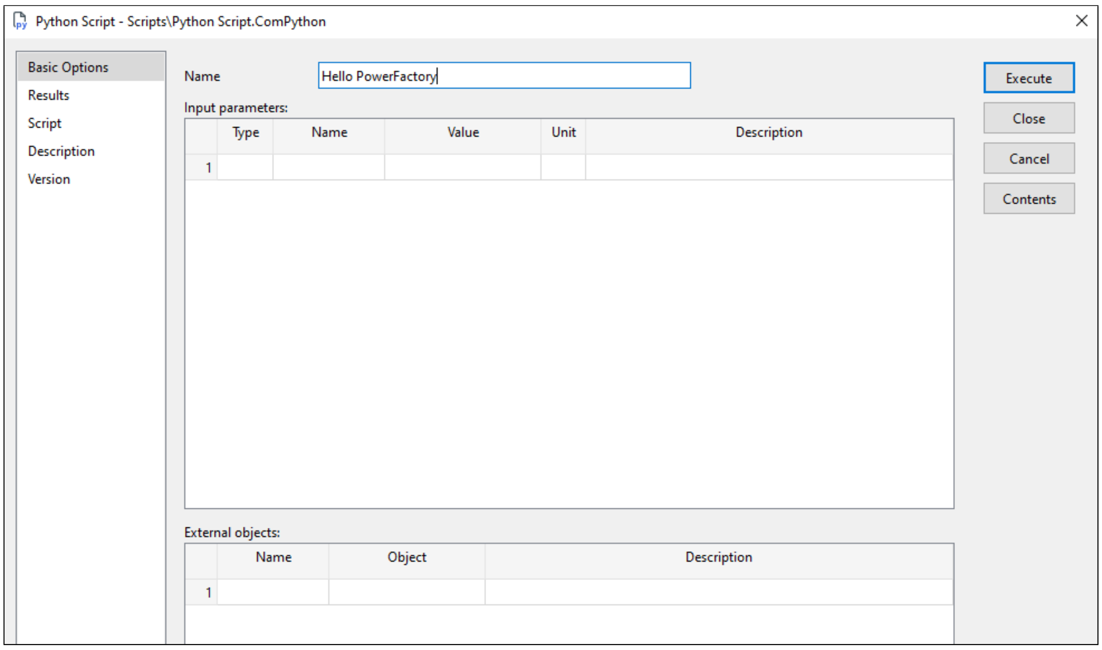


Figure 1.1: Single line diagram of the example network

1.2 Create a Python command object

The next step is to create and set up a Python command script object (*ComPython*) that contains the dedicated code and can be executed:

- Open the Data Manager and navigate to the *Scripts* folder within the project library.
- Click on the *New Object* icon ( ).
- Select *Python Script* and press **OK**.
- The dialog shown in figure 1.2 will be displayed.
- Change the name to “Hello PowerFactory”.



Name: Hello PowerFactory

Input parameters:

	Type	Name	Value	Unit	Description
1					

External objects:

	Name	Object	Description
1			

Figure 1.2: Setup dialog from a *ComPython* object

1.3 Write a Python Script

On the *Script* page of the Python command, the actual code has to be given to the Python command object. There are two options:

- Write embedded code directly into the Python command, similar to DPL (DIgSILENT Programming Language) scripts.
- Link a Python script file (.py) via the file path of the script. So the code itself is maintained outside of *PowerFactory* and only accessed when the Python command object is executed in *PowerFactory*.

In this first step we will write the script as embedded code:

- On the *Script* page of the command, change the *Python Script*-selection to *Embedded*.
- The editor field will now be shown in the command.

1.3.1 How to start a Python script

To allow Python to have access to *PowerFactory*, the “powerfactory” module must be imported. That means, the code has to contain the import command for this “powerfactory” module (usually in the first line):

```
import powerfactory
```

The “powerfactory” module interfaces with the *PowerFactory* API (Application Programming Interface). This solution enables a Python script to have access to a comprehensive range of data available in *PowerFactory*:

- All objects
- All attributes (element data, type data, results)
- All commands (load flow calculation, etc)

- Special built-in functions

To gain access to the *PowerFactory* environment the command GetApplication() must be added:

```
app = powerfactory.GetApplication()
```

These are the two lines of code that each Python script that is interfacing with *PowerFactory* has to contain.

- Write the two code lines into the *Embedded Code* field of the recently created Python script object.

1.3.2 Write messages to the output window

The Python script object (*ComPython*) is a *PowerFactory* command and can be executed. As in other commands, different types of messages can be written to the output window in *PowerFactory* to communicate with the user. The possible messages are:

- Plain text:

  ```
  app.PrintPlain("Text")
  ```

- Information messages:

  ```
  app.PrintInfo("Text")
  ```

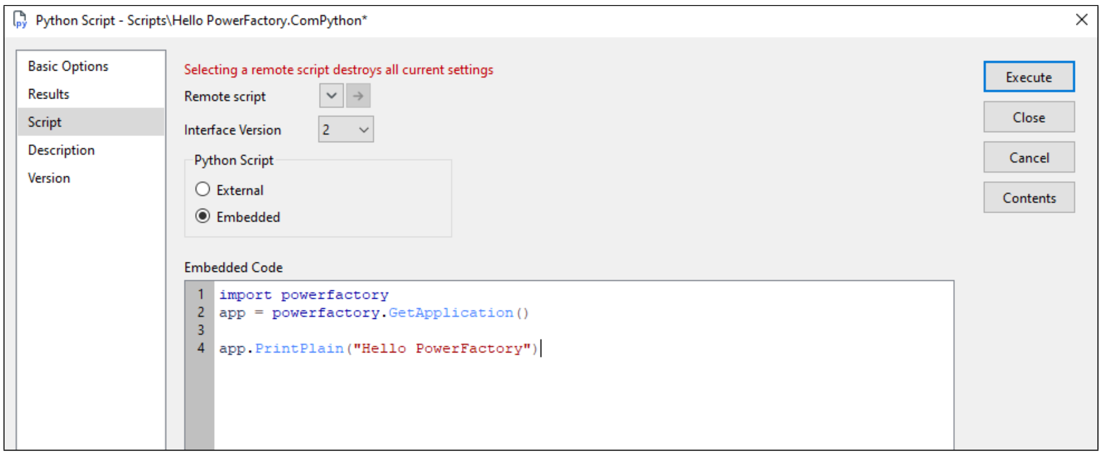
- Warning messages:

  ```
  app.PrintWarn("Text")
  ```

- Error messages:

  ```
  app.PrintError("Text")
  ```

Extend the code in the Hello PowerFactory-Python script object to write the plain text “Hello PowerFactory” into the output window. Execute the Python script object and check the output window.



Selecting a remote script destroys all current settings

Remote script

Interface Version: 2

Python Script
- External
- Embedded [✓]

Embedded Code

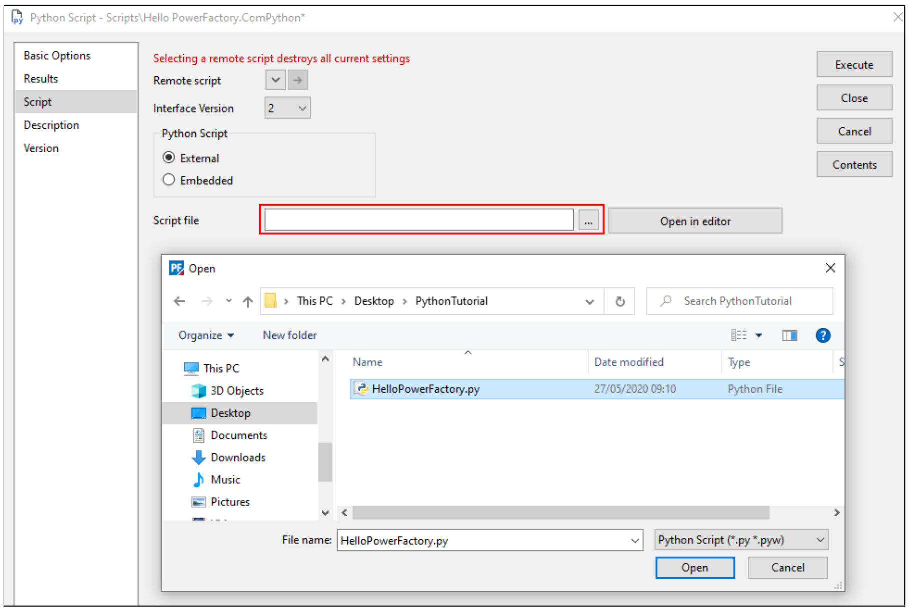
```
import powerfactory
app = powerfactory.GetApplication()

app.PrintPlain("Hello PowerFactory")
```

Figure 1.3: Solution of the “Hello PowerFactory” script as embedded code

1.3.3 Link an external .py-file

Now we will copy the embedded code to an external .py-file and link it to the Python script object.

- Copy the written code from the Python script object.
- Open a text editor of you choice and paste the code.
- Save the file as a Python file (.py) and name it "HelloPowerFactory.py".
- Go back to *PowerFactory* and open the *Hello PowerFactory* Python script object and go to the *Script* page.
- Select *External* on the *Python Script* field.
- Press the button "...", a new window will appear.
- Select the *HelloPowerFactory.py* file.



Remote script:
Interface Version: 2
Python Script: (● External / ○ Embedded)
Script file:

This PC › Desktop › PythonTutorial
Search PythonTutorial

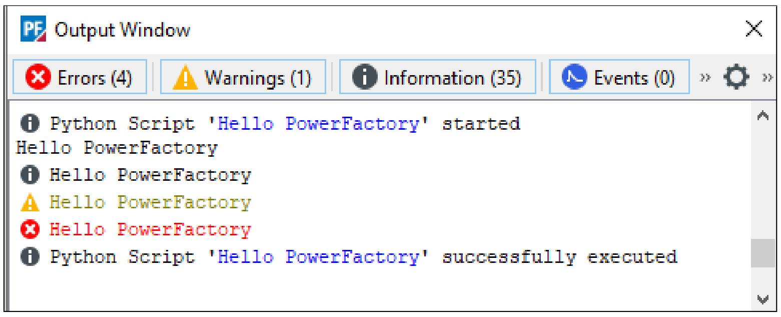
Name	Date modified	Type
HelloPowerFactory.py	27/05/2020 09:10	Python File

File name: HelloPowerFactory.py
Python Script (*.py *.pyw)

Figure 1.4: Linking a Python file to the *ComPython* object in *PowerFactory*

Execute the script in *PowerFactory*. The output should be the same as before when embedded code was used.

Extend the code in the external .py file to print "Hello PowerFactory" as information, warning and error messages as well. Save the .py file and execute the script again.



```
Python Script 'Hello PowerFactory' started
Hello PowerFactory
Hello PowerFactory
Hello PowerFactory
Hello PowerFactory
Python Script 'Hello PowerFactory' successfully executed
```

Figure 1.5: Output of the extended "Hello PowerFactory" script

1.3.4 Available Functions

All the available functions (e.g. PrintPlain, PrintInfo...) are listed in the *Python Function Reference*, a pdf document with a dedicated description for each function. The document can be found via *Help* → *Scripting References*→ *Python*.

The functions can be differentiated between:

- **Application methods:** Functions which are called directly on the application interface.

```
#The printing functions are called on the application interface directly
app.PrintInfo("Every application method is called with app.FunctionName()")

# Getting the load flow command from the active study case
LoadFlowCommand = app.GetFromStudyCase("ComLdf")

# Getting all calculation relevant lines
Listlines = app.GetCalcRelevantObjects("ElmLne")

# To get the line Data object we need to extract one item via indexing
# indexing in Python works with squared brackets:
# item = list[index]
line = Listlines[0]
```

- **General object methods:** Functions which work on all *PowerFactory* objects.

```
# Getting the line length with the GetAttribute function and
# the attribute name "dline"
length = line.GetAttribute("dline")

# Getting the name of the load flow command with the GetAttribute function
# and the attribute  "loc_name"
name = LoadFlowCommand.GetAttribute("loc_name")
```

- **Specific object Methods:** Functions which are designed for a specific object class.

```
# Executing the load flow command to run a load flow calculation in PowerFactory
LoadFlowCommand.Execute()
```

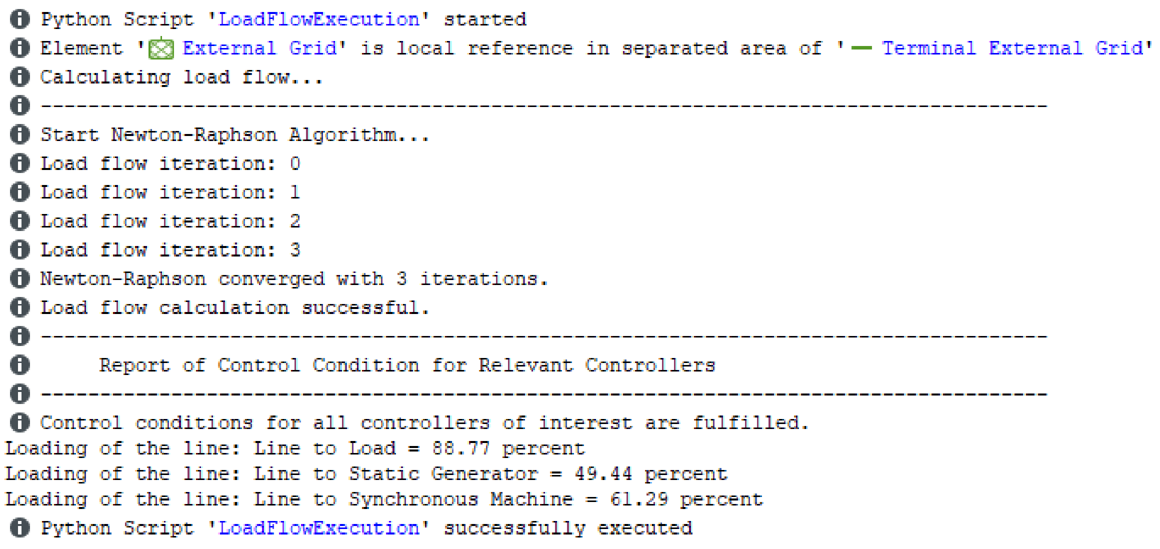
The required functions for the tutorial will be introduced and shown in this tutorial document. But they can also be found in the Scripting reference.

1.3.5 Example: Execute a load flow calculation via script

In this section you will develop a small script that will execute a load flow calculation and print out the name of each line and its corresponding loading in the output window. The used functions are shown in the previous section and will be explained in detail in the following exercises. The code can be written externally and the .py script file linked to the Python command object, or the embedded code functionality can be used. The final script can be found as embedded code in the folder *Solutions* within the *Scripts* library.

- Create a new Python script object in the *Scripts* folder in the project library and name it “LoadFlowExecution”.

- Write the required code into the *Embedded Code* field on the *Script* page. A possible code solution is presented below. The following steps are needed:
  - Import the powerfactory module
  - Get the application object (app)
  - Use the `GetFromStudyCase()` function to get the load flow calculation command
  - Execute the load flow calculation command
  - Get the list of all calculation relevant lines with the `GetCalcRelevantObjects` function
  - Loop through the list of lines and get the name (Parameter: `loc_name`) and the loading value (Parameter `c:loading`) with the `GetAttribute()` function for each line and print the information to the output window with the `app.PrintPlain()` function
- Execute the script. The following results should be seen in the output window:



```
Python Script 'LoadFlowExecution' started
Element 'External Grid' is local reference in separated area of 'Terminal External Grid'
Calculating load flow...
------------------------------------------------------------------------------------
Start Newton-Raphson Algorithm...
Load flow iteration: 0
Load flow iteration: 1
Load flow iteration: 2
Load flow iteration: 3
Newton-Raphson converged with 3 iterations.
Load flow calculation successful.
------------------------------------------------------------------------------------
     Report of Control Condition for Relevant Controllers
------------------------------------------------------------------------------------
Control conditions for all controllers of interest are fulfilled.
Loading of the line: Line to Load = 88.77 percent
Loading of the line: Line to Static Generator = 49.44 percent
Loading of the line: Line to Synchronous Machine = 61.29 percent
Python Script 'LoadFlowExecution' successfully executed
```

Possible solution:

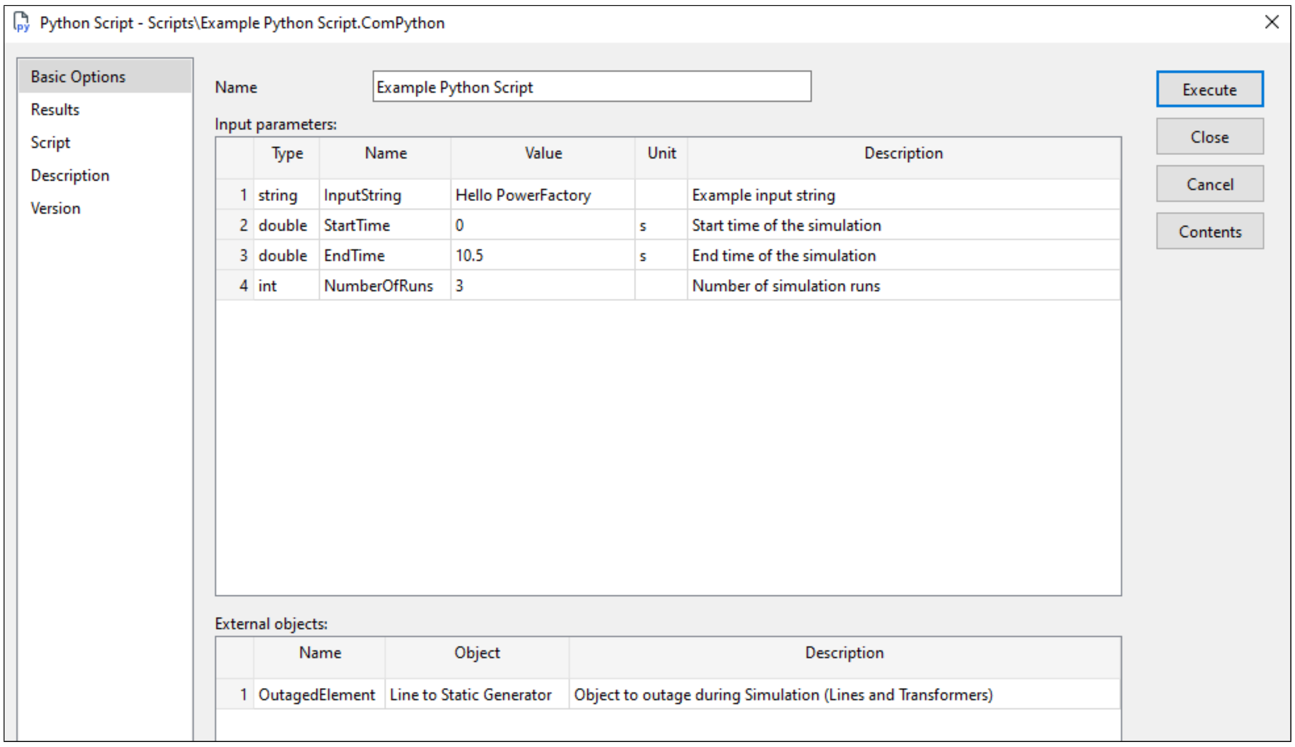
```python
import powerfactory #importing of pf module
app = powerfactory.GetApplication() # Calling app Application object
ldf = app.GetFromStudyCase('ComLdf') #Calling ldf Command object (ComLdf)
ldf.Execute() #executing the load flow command

# Get the list of lines contained in the project
Lines = app.GetCalcRelevantObjects('*.ElmLne')
for line in Lines: #get each element out of list
  name = line.loc_name  # get name of the line
  loading = line.GetAttribute('c:loading') #get value for the loading
  #print results
  app.PrintPlain('Loading of the line: %s = %.2f percent' %(name,loading))
```

1.4 Anatomy of a Python Object

To understand how Python scripting works in *PowerFactory*, it is important to understand the general structure of a complete Python object. The actual script or code is only a part of the story and this script is contained inside an object called the “Python Command” object, which you already created in section 1.2. Now we will have a look at what else is inside the Python command object.

Python command objects and DPL command objects are normally located within the project library under the subfolder *Scripts*.

Whenever you open a Python command object, you will see a window with a number of pages: *Basic Options*, *Results*, *Script*, *Description* and *Version*. We will go through each page and the available functions.



Name: Example Python Script

Input parameters:

	Type	Name	Value	Unit	Description
1	string	InputString	Hello PowerFactory		Example input string
2	double	StartTime	0	s	Start time of the simulation
3	double	EndTime	10.5	s	End time of the simulation
4	int	NumberOfRuns	3		Number of simulation runs

External objects:

	Name	Object	Description
1	OutagedElement	Line to Static Generator	Object to outage during Simulation (Lines and Transformers)

Figure 1.6: Python command object *Basic Options* page

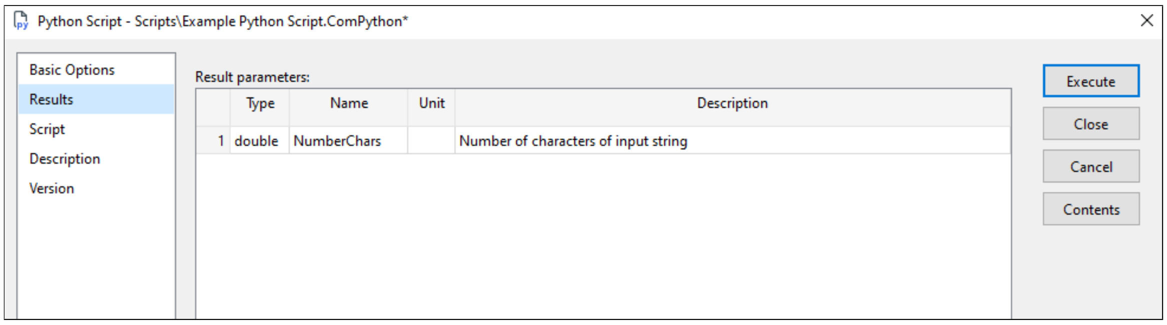
The *Basic Options* page (shown in figure 1.6) has the following functions:

- Name field - Changes the name of the Python command object.
- Input parameters (string, integer or double)- Offer the possibility to define the parameters that can be called by the script. This enables the script user to change the inputs without working in the actual code.
- External objects (*PowerFactory* objects) - Are similar to input parameters, but offer the possibility to preselect objects that can be called in the script. Any object in the *PowerFactory* database can be selected.

Important: To access an external object or input parameter inside of a Python script you have to call the script object (*ComPython*) first and then access the object/parameter as attribute of the script object:

```
script = app.GetCurrentScript() #to call the current script
extObj = script.NameOfExternObj #to call the External Object
inpPar = script.NameOfInpParam #to call Input Parameter
```

Note: The input parameters and external objects as attributes of the script can only be accessed by the currently executed script. To access the input parameters and external objects of the other script objects, the special functions in the scripting reference have to be used. (e.g. `GetExternalObject()` and `SetExternalOject()`).


Python Script - Scripts\Example Python Script.ComPython*

Result parameters:

	Type	Name	Unit	Description
1	double	NumberChars		Number of characters of input string

Figure 1.7: *ComPython* Object, *Results* page

The *Results* page (shown in figure 1.7) has the function to define result parameters - these are essentially output parameters that are stored inside the Python command object (even after script execution). You can access these parameters as if they were from any other type of object, i.e. either by another Python script or as a variable to display on a plot. The defined variables can be found in the variable selection (On the *ComPython* object) under *Basic Data* → *Calculation Parameter*.

In the *Script* page (shown in figure 1.3) the following functions are found:

- Define a remote script - rather than use a script defined locally in the Python command object, you can select a script that has been defined in a separate object.
  One reason to employ a remote script is if you have multiple study cases using the same script code, but different input parameters. If you were to use a remote script, then modifying the master script will affect all the Python objects that refer to it. If you had locally defined scripts, then you would need to change the code in every single local Python object individually.
- Set the interface version - this option depends on the different Python versions. In version 1 (old interface) some data object methods returned a list with additional information. Version 2 (new interface) returns only the results without the input parameter. In this tutorial only the new interface version will be used.

- Python Script:
  - *External:* Define the script file path - the Python script is an external file which is only linked to the Python command object. Here you can define the path where your file .py is located and also, if an external application is set, it is possible to open the linked file directly in an editor.
  - *Embedded:* Write the code into the *Embedded Code* field.

The *Description* and *Version* pages are informational pages for script descriptions and revision information.

The Python command object may contain objects or references to other objects available in the *PowerFactory* database. These can be accessed by clicking on the **Contents** button. New objects are defined by first clicking the *New Object* icon in the toolbar of the Python script contents dialog and then by selecting the required object from the *New Object* pop-up window which appears. References to other objects are created by defining a "IntRef" reference object. Contents can be accessed with the `GetContents()` function. An example showing the possible contents of a Python command object is shown in Figure 1.8.

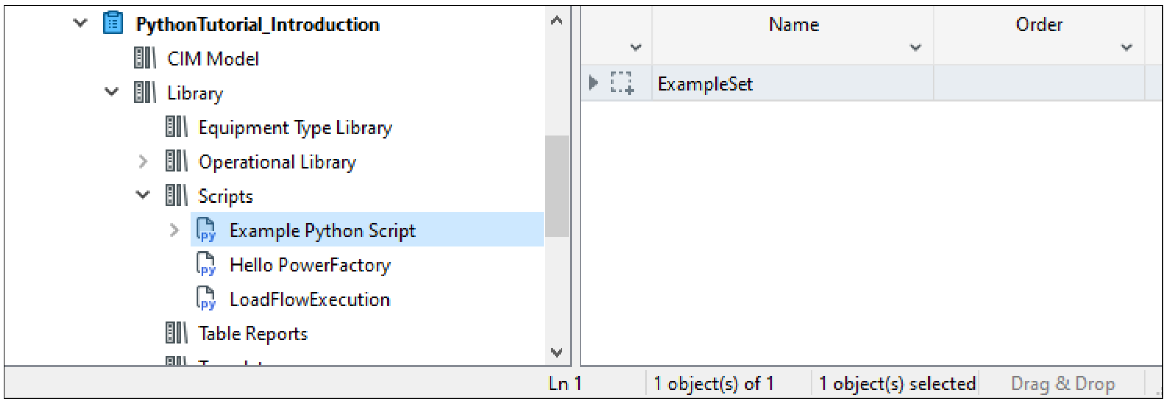


Figure 1.8: Contents of a Python command object

1.4.1 Example Script

- Open the "Example Python Script" provided in the project and check the setup of:
  - Input parameter
  - External object
  - Result parameter
  - Content
- Analyse the embedded code and the included comments.
- Execute the script.
- Check the output window to see that the inputs are used as intended.
- Check the flexible data page for the result parameter of the script.
- Adapt the input parameters and execute the script again. Check the changed outputs of the script.

2 Basic Python Scripting

In this section the basic interaction between the Python code and *PowerFactory* is introduced. Click on the icon from the exercise “Basic Python Scripting” of the “Scripting with Python in *PowerFactory*” tutorial window, to import and activate the project.

The code examples from the individual subsections are available in the folder *Solutions* within the *Scripts* library.

2.1 Access Network Objects

The general approach for accessing network objects in Python purely through code is as follows:

- Get a list of the relevant network objects that you are looking for (using the `GetCalcRelevantObjects()` command), based on a specific element type, e.g. lines, transformers, motors, etc.
- Get an object within this list through indexing (`list[x]`) or by using a `for` loop, etc.

The code snippet below gets the set of all lines, cycles through each line object and prints out its name:

```python
import powerfactory
app = powerfactory.GetApplication()
Lines = app.GetCalcRelevantObjects('*.ElmLne') #get list of all lines
# "*" is a placeholder.
for line in Lines:
    app.PrintPlain(line.loc_name)
    #instead of the name the object itself can be printed
    app.PrintPlain(line)
```

The code snippet below gets the set of all objects, and tries to find the particular line called “Line1”:

```python
import powerfactory
app = powerfactory.GetApplication()
AllObj = app.GetCalcRelevantObjects() #get list of all objects without a filter

listLine = app.GetCalcRelevantObjects('Line1.ElmLne') #get a specific line object ('Line1')
#if there is no Line1 the list will be empty. If conditions will help to check the returns
if listLine != []: #if not empty list -> line found
  Line = listLine[0] #get the actual line object from the list via indexing
  app.PrintPlain("Found line object: %s"%Line)
```

The code snippet below gets the list of all objects, filters for all lines starting with “Line”, cycles through each line object in the filtered set of lines and prints out its full name.

```python
import powerfactory
app = powerfactory.GetApplication()
#Get all lines with names starting with 'Line'
Lines = app.GetCalcRelevantObjects('Line*.ElmLne')
for Line in Lines:
    app.PrintPlain(Line.loc_name)
```

2.2 Identify, Access and Modify Object Parameters

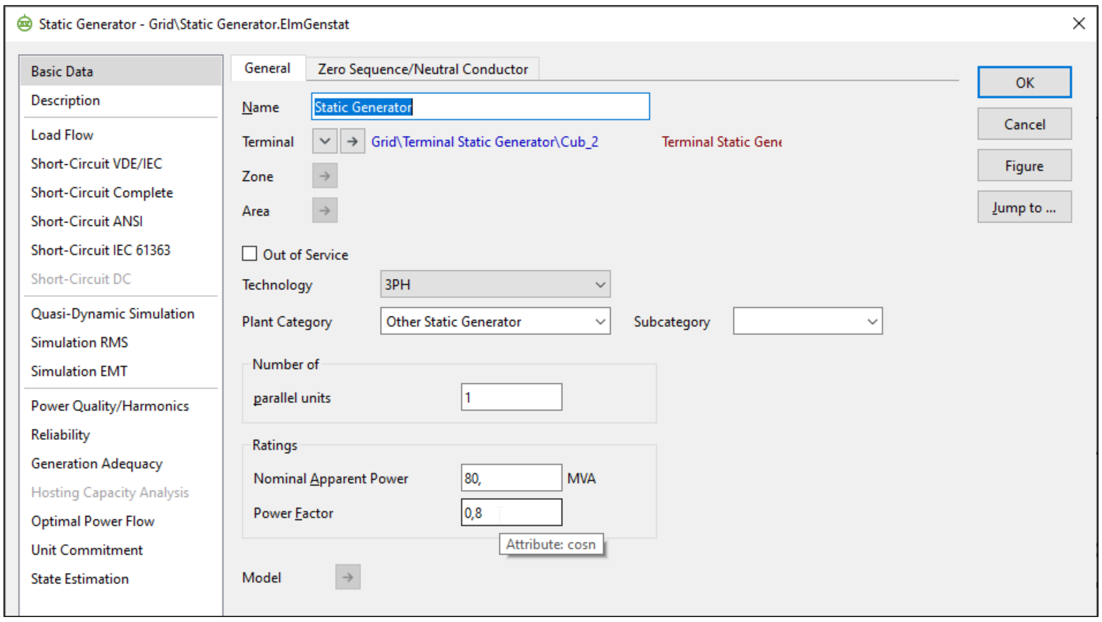
Once a specific object has been selected, the way to access the object parameters or variables is by the variable name separated by a point ".", e.g. `Object.Variable_name`.

For example:

```
NameOfALine = Line.loc_name
```

2.2.1 Identify Variable Names for a Parameter

Variable names can often be found in the manual and in the technical references, but the easiest way to identify variable names is to open the edit dialog of the relevant object and hover the mouse over the field of interest. A tooltip will appear with the corresponding variable name. For example, hovering over the power factor field in the static generator element yields the variable name: "cosn":


Static Generator - Grid\Static Generator.ElmGenstat

Name: Static Generator
Terminal: Grid\Terminal Static Generator\Cub_2   Terminal Static Gene
Zone
Area
☐ Out of Service
Technology: 3PH
Plant Category: Other Static Generator   Subcategory:
Number of
parallel units: 1
Ratings
Nominal Apparent Power: 80, MVA
Power Factor: 0,8
Attribute: cosn
Model

Figure 2.1: Static Generator *Basic Data* page

This means that to get the value of the power factor for this one static generator you could use following code:

```
import powerfactory
app = powerfactory.GetApplication()

staticGeneratorObj = app.GetCalcRelevantObject('*.ElmGenstat')[0]
cosFi = staticGeneratorObj.cosn
```

2.2.2 Access Parameter

Suppose we have a line object “Line” and we want to save the length of the line (variable name = dline) to an internal Python variable dLength. The instruction below may be used.

```
dLength = Line.dline
```

2.2.3 Modify Parameters

Suppose we have a line object “Line” and we want to change the length of the line (variable name = `dline`) to 2 km. The instruction below may be used.

```
Line.dline = 2
```

2.2.4 Access Result Parameters

The direct access via `object.variableName` is only possible for parameters which are directly contained by the object or element (element parameter in the variable selection). If other attributes such as results should be accessed, the Python syntax does not allow access with a colon as DPL does. The function `object.GetAttribute(“variableName”)` must be used instead. The following lines of code will each return the correct value for the line length:

```
length = Line.dline
length = Line.GetAttribute("dline")
length = Line.GetAttribute("e:dline")
```

But when accessing the result parameter `c:loading` after a load flow calculation, the `GetAttribute()` function has to be used:

```
loading = Line.GetAttribute("c:loading") # This will work
loading = Line.GetAttribute("loading") # won't work. the 'c' is mandatory.
#the prefix can only be neglected for element parameters
loading = Line.loading # won't work. The variable (e:)loading does not exist.
```

The `object.SetAttribute(“variableName”, value)` is the corresponding function for setting parameters.

2.3 Navigate Folders and Object Contents

In order to access certain object or folders it is often necessary to navigate through the *PowerFactory* data structure. There are several functions available to support this.

2.3.1 Project Folders

As an entry or starting point into the project folder hierarchy the `GetProjectFolder(string)` function can be useful. This function will return an object with a reference to the top level of project folders, e.g. the folders containing study cases, scripts, libraries, diagrams, etc.

For example, the following line puts a reference to the equipment type library folder into object “oFold”:

```
oFold = app.GetProjectFolder('equip')
```

A selected list of the project folders available and the corresponding string is shown below:

String	Folder Description
equip	Equipment type libary
netmod	Network model
oplib	Operational libary
scen	Operational scenario
script	Script
study	Study Case
templ	Template
netdat	Network data
dia	Diagram
scheme	Variation
cbrat	CB rating
therm	Thermal rating
ra	Running arrangement
mvar	Mvar limits curve
outage	Outage
fault	Fault

Table 2.1: Available project folders

The complete list can be found in the scripting reference.

2.3.2 Object Contents

The `GetContents(string)` function is a generic way of navigating through objects and finding the list of objects contained within them. The function returns a list of objects and a string can be used to filter the content for certain objects.

Some examples:

- Return all objects contained in “oObj” into the list “Contents”

```
Contents = oObj.GetContents()
```

- Return “ElmTerm” type objects (terminals) contained in “oObj” into the list “Contents”

```
Contents = oObj.GetContents('*.ElmTerm')
```

- Return the specific object “T2.ElmTerm” contained in “oObj” into the list “Contents”

```
Contents = oObj.GetContents('T2.ElmTerm')
```

- Return all “ElmTerm” type objects that have names starting with “T” contained in “oObj” into the list “Contents”

```
Contents = oObj.GetContents('T*.ElmTerm')
```

- The GetContents function only checks the direct children by default. If all contained sub folders/objects shall be search the recursive search can be enabled by giving the optional argument recursive = 1 to the function call.

  Return all terminals that are contained in the object and its sub folders:

```
Contents = oObj.GetContents('*.ElmTerm', 1)
```

2.3.3 Object Parent

The functions `GetContents()` and `GetChildren()` are designed to get the contents of an object. If the parent of an object is needed the `object.GetParent()` function can be used.

Example: Return the parent of a study case (study case folder):

```
import powerfactory
app = powerfactory.GetApplication()
activeStudyCase = app.GetActiveStudyCase() #function to get the active study case
fStudyCases = activeStudyCase.GetParent()
```

2.3.4 Objects in a Study Case

In order to access objects within the active study case (e.g. calculation command objects, simulation events, graphics boards, sets, outputs of results, title blocks, etc), you can use the function `GetFromStudyCase(string)`.

This function is essentially a shortcut to accessing objects inside a study case, which is used to navigate to the study case project folder, selecting a study case and then selecting an object. The other advantage of `GetFromStudyCase()` is that if the object does not exist inside the study case, the function will create it.

Note that this function only works with the active study case. The code snippet below gets the load flow command object from the active study case.

```
import powerfactory
app = powerfactory.GetApplication()

ComLdf = app.GetFromStudyCase('ComLdf')
app.PrintPlain(ComLdf)
```

2.4 Access Study Cases

Study cases are *IntCase* objects that are stored in the *Study Cases* project folder. In order to access a study case, you must first access the study case folder.

The code snippet below does the following:

- Gets the list of all study cases.
- Counts the number of study cases.
- Activates each study case and prints its full name.

```python
import powerfactory
app = powerfactory.GetApplication()

#Get the study case folder and its contents
fStudy = app.GetProjectFolder('study')
StudyCases = fStudy.GetContents('*.IntCase',1)

#Counts the number of study cases
iCases=len(StudyCases)
if iCases==0:
    app.PrintError('There is no study case in the project')
else:
    app.PrintPlain('Number of cases: %i' %(iCases))

#Cycle through all study cases and activate each
for Case in StudyCases:
    app.PrintPlain(Case.loc_name)
    Case.Activate
    #here could be the code for executing a load flow calculation in each study case
```

Note: Instead of printing the `loc_name` parameter of an object, you can just print the object itself. For example, to print a study case object you can use `app.PrintPlain(activeStudyCase)`. This way, the object will appear in output window in blue and can be accessed by clicking on the text.

```python
import powerfactory
app = powerfactory.GetApplication()
activeStudyCase = app.GetActiveStudyCase()
app.PrintPlain(activeStudyCase.loc_name)
app.PrintPlain(activeStudyCase)
```

2.5 Execute Calculations

The `GetFromStudyCase(string)` can be used to get an existing or create a new calculation command object. The `Execute()` function can then be used to execute the calculation.

The code snippet below executes a load flow in the active study case:

```python
import powerfactory
app = powerfactory.GetApplication()

```

```
#Get load flow object from study case
ldf = app.GetFromStudyCase('ComLdf')
#Execute load flow calculation
ldf.Execute()
```

2.6 Access Results

2.6.1 Static Calculations

The results of static calculations (Load Flow, Short Circuit, etc.) are stored as parameters in the objects themselves. To get the value of these results you can use the `GetAttribute` method.

`Object.GetAttribute('Result_variable_name')`

For example, suppose you have a line object “Line” and you want to save the loading of the line to an internal Python variable called “LineLoad”:

`LineLoad = Line.GetAttribute('c:loading')`

The simple example below runs a load flow for the active study case, gets all the lines and prints out the name and loading of each line.

```
import powerfactory
app = powerfactory.GetApplication()

#Get load flow object from study case
ldf = app.GetFromStudyCase('ComLdf')
#Execute load flow calculation
ldf.Execute()
lines = app.GetCalcRelevantObjects('*.ElmLne')
for line in lines:
    name = line.loc_name
    LineLoad = line.GetAttribute('c:loading')
    app.PrintPlain('Loading of the : %s = %.2f percent'%(name,LineLoad))
```

2.6.2 Result files (Dynamic Simulations)

Some calculation results such as those of dynamic simulations are not stored as part of the object parameter, but in a separate results file “.ElmRes”. Refer to the section 3.1 for more information.

2.7 Exercise

- Create a new Python command object and name it “StudyCaseSweep”
- Write a script with the following functionality (Some code examples from sections above may be helpful):
  - Import the `powerfactory` module
  - Get the application object
  - Get the study case project folder

- Use the `GetContents()` function to get a list of all study cases
- Get a list all calculation relevant lines.
- Use a for loop to go though the study cases and:
  - Activate each study case and print its name to the output window.
  - Get the load flow command from the active study case.
  - Execute the load flow command.
  - Loop through the list of lines and print the line name and its loading in the output window.

- Execute the script and check the output. You should see the line loading information for each study case.
- A possible solution is provided in the project.

3 Advanced Python Scripting

Click on the icon from the exercise “Advanced Python Scripting” of the “Scripting with Python in *PowerFactory*” tutorial window, to import and activate the project.

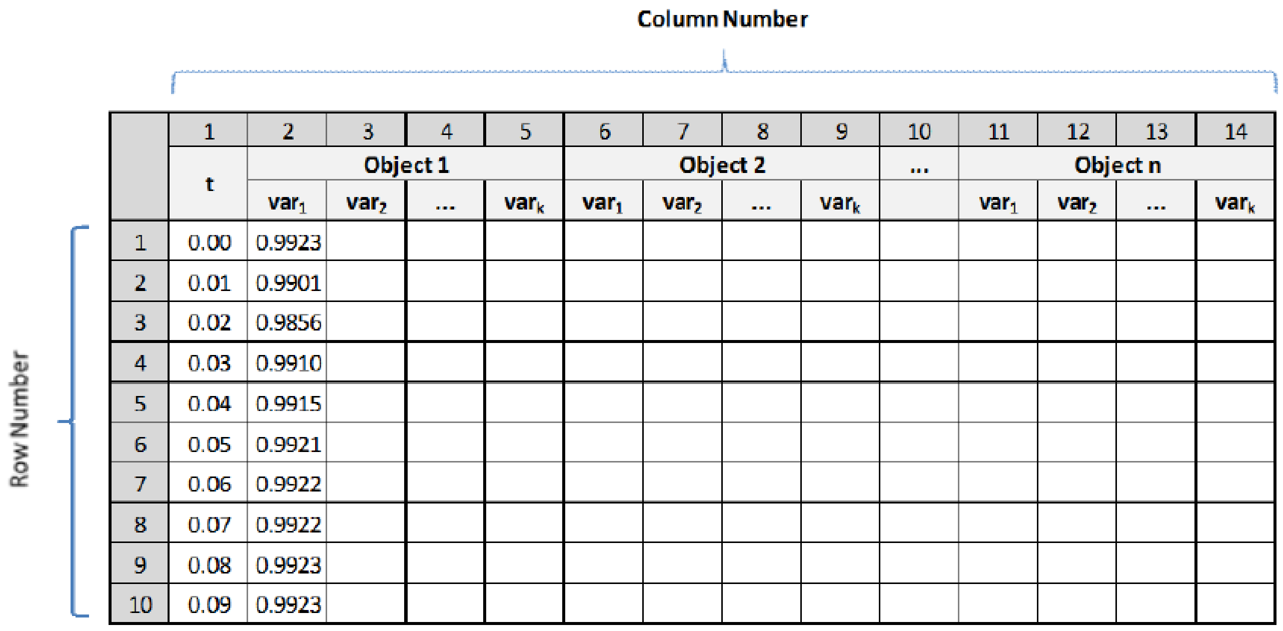
The code examples from the individual subsections are available in the folder *Solutions* within the *Scripts* library.

3.1 Working with Results Files

Suppose you want to get a *PowerFactory* results file (of type *ElmRes*) from a dynamic simulation, look into it, pull out a set of relevant values, perform some calculations on the data and generate some outputs. How do you do that?

3.1.1 Structure of Results Files

In order to manipulate the data in a results file, it is important to understand the structure of the file and how data is stored inside it. The results file is structured as a 2d matrix as shown in the figure below.


Column Number
Row Number

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	t	Object 1				Object 2				...	Object n			
		var$_1$	var$_2$	...	var$_k$	var$_1$	var$_2$	...	var$_k$		var$_1$	var$_2$	...	var$_k$
1	0.00	0.9923												
2	0.01	0.9901												
3	0.02	0.9856												
4	0.03	0.9910												
5	0.04	0.9915												
6	0.05	0.9921												
7	0.06	0.9922												
8	0.07	0.9922												
9	0.08	0.9923												
10	0.09	0.9923												

Figure 3.1: Structure of an ElmRes object

The number of rows represents the total number of time intervals for the simulation. For example, if there is 10 s of simulation with a time step of 0.01 s, then there would be 1000 rows. Each row represents a point in time.

Column 1 in the results file is the **time** of the simulation.

Columns 2 onward represent the **objects** (shown in the diagram as object 1 to object n) and their respective **variables** (shown in the diagram as var_1 to var_k). An object can be any *PowerFactory* object (e.g. a line, motor, terminal, load, etc.) and a variable is any element defined in a variable set for the specific object that is to be measured during the simulation (e.g. m:u1, s:speed, etc.)

Note that in the diagram above, each object has k variables, but this is not necessarily the case in practice as each object can be defined with an arbitrary number of variables in the set.

It is important to know that accessing data from a specific object and variable in the results file hinges on knowing the relevant column index. Similarly, accessing data for a particular time requires the row number for the relevant time point.

3.1.2 Loading a Results File into Memory

Getting the results file object with a Python script does not directly enable access to the data contained within it. You must first load the results file into memory using the command:

```
ResultFile.Load()
```

With this call the contents of a results file are loaded into memory and therefore allows you to begin accessing the data.

3.1.3 Getting the Relevant Column Number

In order to access the object and variable of interest, you need to know the right column number in the results file. To do this, use the following command to search for the relevant column number:

```
ResIndex = ResultFile.FindColumn(Object, Variable Name)
```

`FindColumn` returns an integer, which is the column number for the object and variable you have searched for. “Object” is the specific *PowerFactory* object of interest (which can be defined explicitly or by a general selection). “Variable Name” is the variable of interest (e.g. `m:u1`, `s:speed`, etc.)

3.1.4 Finding the Number of Time Intervals

To find the number of time intervals in the simulation (i.e. number of rows), use the command:

```
ResultFile.GetNumberOfRows()
```

The function returns an integer with the number of time intervals for a specific column. Using Column index = 0 gives the total number of rows.
A similar function `GetNumberOfColumns` returns number of columns.

```
ResultFile.GetNumberOfColumns()
```

3.1.5 Getting Data from the Result File

Once you know the relevant column index (object / variable) and row index (time), you can start getting data from the results file. Data points can be accessed one at a time:

```
# One at a time
[success, value] = ResultFile.GetValue(<row index>,  <column index>)
```

The row and column indices are integers. The return value is a list with an integer indicating whether the call was successful or not and, in case of `GetValue`, also the value.

If all data from a column is relevant the `GetColumnValues` function can be used. If does not directly write the data to a list in Python but into a given vector object. The column data can be obtained from this vector.

```
# Get a vector object (IntVec) for example from the study case
vectorObject = app.GetFromStudyCase("VectorForRedingResults.IntVec")
#write the column data to the vector
ResultFile.GetColumnValues(<column index>)
#get the data from the vector to a list in python
listOfColumnResults = vectorObject.GetAttibute("e:V")
```

Alternatively, exporting results with the Results Export Command (ComRes) to an external file or database is possible.

3.1.6 A Simple Example

This example opens a results file, prints number of rows and columns of it, and all values for `c:loading` variable of an specific line object. In order for the script to work, the result file has to contain the results for the loading of this line. You can try this in the given example project for this exercise.

```
import powerfactory
app = powerfactory.GetApplication()

#Get the Quasi-Dynamic Simulation AC Result file (ElmRes) from the study case
resultFile = app.GetFromStudyCase("Quasi-Dynamic Simulation AC.ElmRes")

#Loading of Result file
resultFile.Load()

#Get object(s) for which results are to be extracted
line = app.GetCalcRelevantObjects('ElmLne')[0]


#Get number of rows and columns
NumCol = resultFile.GetNumberOfColumns()
NumRow  =resultFile.GetNumberOfRows()

#print results
app.PrintPlain('The result file has %i rows and %i Columns' %(NumRow,NumCol))

#Get index of variable of interest
ColIndex = resultFile.FindColumn(line,'c:loading')

#print results
app.PrintPlain("Line loadings of %s:"%line)
for i in range(NumRow):
  value = resultFile.GetValue(i,ColIndex)[1]
  app.PrintPlain(value)
```

3.1.7 Writing Data to the Result File

It is also possible to write data from several simulations (e.g. several load flows) to the result file. To this end, result variables from a calculation relevant object (e.g. a line) must be added to the result file and the writing must be initialized. Then the results of several simulations can be written to the file as illustrated in the following code snippet:

```python
import powerfactory
app = powerfactory.GetApplication()

# Get line
oLine = app.GetCalcRelevantObjects('Line to Load.ElmLne')[0]
# Get result object and delete stored data
oRes = app.GetFromStudyCase('All calculations.ElmRes')
oRes.Clear()
# Add loading variable
oRes.AddVariable(oLine,'c:loading')
# Initialize writing
oRes.InitialiseWriting()

# Calculate load flow for various loadings by varying load active power
oLdf=app.GetFromStudyCase('ComLdf');
oLoad = app.GetCalcRelevantObjects('General Load 1.ElmLod')[0]
for load_active_power in range(50,100,10):
  # Set load active power
  oLoad.plini = load_active_power
  # Execute load flow command
  oLdf.Execute()
  # Write to result object
  oRes.Write()
oRes.FinishWriting()
```

3.2 Plotting Results

Plots can be created using Python.

3.2.1 Creating a Plot Page

You can create a diagram or plot page based on a graphics board object (*.SetDesktop). The function `GetPage (string name, int create)` will create a new page provided the name does not refer to an existing page and the create flag is activated (=1).

For example, the snippet below uses the graphics board object “GraphicsBoard” to create a new page called “Plots”:

```python
GraphicsBoard = app.GetFromStudyCase('SetDesktop')
page = GraphicsBoard.GetPage('Plots',1)
```

3.2.2 Creating a Plot

Similar to creating a new page, you can use the page object (*.GrpPage) to create a new plot or diagram with the function `GetOrInsertCurvePlot(string name, int create)`. To create a new plot, the name should not be the same as another existing one and the create flag should be activated (=1).

For more information about plots and the different classes, please refer to User Manual, section Plots.

```
GraphicsBoard = app.GetFromStudyCase('SetDesktop')
# Alternative: app.GetGraphicsBoard()
page = GraphicsBoard.GetPage('Page',1)
oPlot = page.GetOrInsertCurvePlot('Plot',1)
```

3.2.3 Adding Objects and Variables to Plots

In order to show the actual plots of specific variables in a diagram, you need to add objects and variables to the diagram. It is necessary to first access the data series of the diagram using the `GetDataSeries` function. If there are already curves in the data series object, they are cleared using the function `ClearCurves`. The function `AddCurve` then adds variables of an object to the data series of the plot.

```
DataSeries = oPlot.GetDataSeries()
DataSeries.ClearCurves()
DataSeries.AddCurve(oBus1, 'm:u1')
```

3.2.4 Plotting Example

The example below executes several load flow calculations for various setpoints of a load. The results of the calculations are written to a result file (see 3.1). Then, it gets the current graphics board, creates a graphics page called “'Plots'”, creates a subplot, adds the curve for the object “oLine” and the attribute “c:loading”. In the end the axes scale is adjusted:

```
import powerfactory
app = powerfactory.GetApplication()

# Get line
oLine = app.GetCalcRelevantObjects('Line to Load.ElmLne')[0]
# Get result object (stored in Python object)
script = app.GetCurrentScript()
oRes = script.GetContents('All_Calcs')[0]
oRes.Clear()
# Add loading variable
oRes.AddVariable(oLine,'c:loading')
oRes.InitialiseWriting()

# Calculate load flow for various loadings by varying load active power
oLdf=app.GetFromStudyCase('ComLdf');
oLoad = app.GetCalcRelevantObjects('General Load 1.ElmLod')[0]
for load_active_power in range(50,100,10):
  # Set load active power
  oLoad.plini = load_active_power
  # Execute load flow command
  oLdf.Execute()
  # Write to result object
  oRes.Write()

# Get current graphic board
oGrb = app.GetGraphicsBoard()
# Create plot page
oPage = oGrb.GetPage('Plots',1)
# Create a new Plot
oPlot = oPage.GetOrInsertCurvePlot('Diagram',1)
DataSeries = oPlot.GetDataSeries()
DataSeries.ClearCurves()
# Add variable 'c:loading'
DataSeries.AddCurve(oLine,'c:loading')
# Set axis scale to automatic
oPlot.SetAutoScaleModeX(1)
oPlot.SetAutoScaleModeY(1)
```

3.2.5 Exporting WMF Graphic Files

The function `WriteWMF(filename)` can be used to export the active graphics page to a graphic file in Windows Metafile (WMF) format. The function can only be used with a graphics board object (*.SetDesktop), so a relevant graphics board object needs to be retrieved before exporting to a WMF.

The example below gets the first graphics board in the active study case and exports the active page to the desktop and names it "graphic_export". Note there is an "r" in front of the file path which means it is interpreted as a raw string and special characters have no meaning.

```
import powerfactory
app = powerfactory.GetApplication()

#Get current graphic board
oGrb = app.GetGraphicsBoard()

#Get all Plot Pages
pages = oGrb.GetContents("*.GrpPage")

exportPath = r"C:\\temp" #example path.
for page in pages:
  app.PrintPlain("Plotting %s"%page)
  oGrb.Show(page)
  oGrb.WriteWMF(exportPath + "\\" + page.loc_name)
```

If graphics should be exported to another file format, the command *ComWr* can be used.

3.3 Topological Search

It is sometimes useful to be able to navigate topologically through a network and search for elements using Python. For example, suppose you want to get the list of transformers connected to a bus, or you want to follow a branch circuit through lines and transformers to its end point. *PowerFactory* has a number of Python functions that can help in this task.

Some of the useful topological search functions:

3.3.1 Object Functions

- `GetConnectedElements(Breakers,disconnectors,out_of_service)`: get a list of the elements connected to an element.
  - Example a): to get all of elements connected to "oTerm" and puts it into the list "setObj", taking into account the state of breakers and switch-disconnecter, but disregarding out of service flags.

    ```
    setObj = oTerm.GetConnectedElements(1,1,0)
    ```

  - Example b): not setting a value, uses the default options (0,0,0). Therefore, this gets the set of all elements connected to "oTerm", irrespective of breaker/switch operating states and out of service flags.

    ```
    setObj = oTerm.GetConnectedElements()
    ```

    Important: when using this function to find the connected terminals, be careful about the use of the *internal node* option in the busbars/terminals. When this option is set, the `GetConnectedElements()` will ignore the terminal. This is to avoid returning the internal nodes of a subsection.

- `GetNode(bus_no=0 or 1,switch_state=0 or 1)`: get the terminal/node connected to an object.
  - Example a): to get the terminal connected to bus "0" of line object "oLine", taking into account the switch state, and put the terminal into object "oTerm":

    ```
    oTerm=oLine.GetNode(1,0)
    ```

  - Example b): By default, the switch state setting is "0", so the above snippet gets the terminal connected to bus "1" of line object "oLine", ignoring switch states.

    ```
    oTerm = oLine.GetNode(1)
    ```

- `GetCubicle(index)`: gets the cubicle connected to an object. For example to get the cubicle with index "1" at the line object "oLine" and put it into the object "oCub":

  ```
  oCub = oLine.GetCubicle(1)
  ```

  Returns NULL if the cubicle does not exist.

- `GetConnectionCount()`: get the number of connections/ cubicles connected to an object. For example, to get the number of connections or cubicles connected to line object "oLine" and assign it to the variable "i":

  ```
  i = oLine.GetConnectionCount()
  ```

- `GetClassName()`: gets the name of the class (useful when filtering for certain types of objects). For example, to get the class for the object "obj" and stores the result on variable "Namely":

  ```
  Namely = obj.GetClassName()
  ```

3.3.2 Terminal Functions

- `GetNextHVBus()`: get the next busbar (at a higher nominal voltage level) connected to the terminal. Example:

  ```
  oBus = oTerm.GetNextHVBus()
  ```

  Gets the next bus with a higher voltage relative to terminal “oTerm” and returns the result to object “oBus”. If no bus is found, then null is returned.

For more functions please refer to the scripting reference.

3.4 Create new objects

There are three general ways to create new objects in the database:

- Copy an existing object with the AddCopy() function
- Create new object from scratch in code with the CreateObject() function
- Use the get or create functionality of various function like the GetFromStudyCase(), GetOrInsertPlot(),...

3.4.1 Create a copy

Copying an object (network element) is potentially the easier option for creating new objects, since all the parameters and settings but the name are take from the original object. The original object can be found or given in various ways e.g. as an external object, by a (topological)-search, as content of the Python command object...

To create the object, you can use the **AddCopy(object_name)** command. In order to use this command, you must first locate the folder (or object) that you want to copy the original or template object into.

For example, suppose “oFold” is the target folder (for example a Grid) and “Line” is the original object to copy, a new line object will be created in the target folder “oFold” by the following command:

```
NewLine = oFold.AddCopy(Line)
```

The code snippet below copies line into a selected grid and then changes the length of the line to 10 km.

```
import powerfactory
app = powerfactory.GetApplication()

#Get grid as a target folder
TargetGrid = app.GetCalcRelevantObjects("Grid.ElmNet")[0]

#Get the line to copy (User selected as external object)
script = app.GetCurrentScript()
#External object name is "LineToCopy"
LineTemplate = script.LineToCopy

#check if a line was selected
if LineTemplate == None or LineTemplate.GetClassName() != "ElmLne":
  app.PrintError("Please select a line object (ElmLne)")
  exit() #exit terminates the script

#Copy the line and give it a new name
newLine = TargetGrid.AddCopy(LineTemplate, "NewLineName")
```

```
app.PrintPlain("%s created"%newLine)

#change the length of the new line
newLine.dline = 10
```

3.4.2 Create a New Object by Code

Creating new objects purely by code is the most intensive method for making new objects, because all of the object parameters will be initialised with their default value and each has to be set in code. With template objects, you can set default parameters and even add child objects inside the template.

The **CreateObject(class_name, object_name)** function is used to create new objects. You must first locate the folder (or object) that you want to create the object in. The function can be used to create any kind of object like network elements, folders, events, library contents...

For example, suppose “oFold” is the target folder and you want to create a short-circuit event object (*.EvtShc) called “SC_Event”. You would use the following command:

```
  oFold.CreateObject('EvtShc','SC_Event')
```

Note that if the target folder (or object) does not accept the object class you are trying to create (e.g. a plot page object in the simulation events folder) then an error will be raised.

The code snippet below creates a new short circuit event into the simulation events folder of the active study case and then sets the time of the event to t=1.

```
import powerfactory
app = powerfactory.GetApplication()

#Get the simulation events folder from the active study case
oFold = app.GetFromStudyCase('IntEvt')
app.PrintPlain(oFold)

#Copy the template short circuit event into the events folder
EventSet = oFold.CreateObject('EvtShc','SC_Event')

# set the time to 1
oEvent.time=1
```

4 Additional Exercise 1

Before starting with this exercises make sure that you have gone through the previous exercises. Click on the icon from the exercise “Additional Exercise 1” of the “Scripting with Python in *PowerFactory*” tutorial window, to import and activate the project.

The code examples from the individual subsections are available in the folder *Solutions* within the *Scripts* library.

The exercise is separated in several subtasks:

- Create time characteristics
- Execute a load flow calculation
- Execute a quasi dynamic simulation and plot the results
- Export plots and network diagrams

4.1 Time Characteristics

In *PowerFactory* any parameter can contain a range of values (known as a Characteristic). This values can represent for example:

- Load demand based on the minute, day, season, or year of the study case.
- Generator operating point based on the study being conducted.
- Line/transformer ratings, generator maximum power output, etc. vary with ambient temperature.

Values from the characteristic are selectable by date and time, or by a user-defined trigger. The range of values may be in the form of a scaling factor, a one-dimensional vector or a two-dimensional matrix.

In this exercise a characteristic on load consumption data for a 24h period will be created in time characteristic. This time characteristic will be assigned to one specific load.

4.1.1 Exercise 1: Import Time Characteristic

- Import powerfactory module and call the application object
- Select load to which the characteristic will be assigned as external object
- Check if there are any characteristic already assigned to this element and if there are, delete them.
  - Check the contents of the load for “ChaRef”-objects
  - Delete the existing “ChaRef”-objects, if there are any
- Create the *ChaTime* object inside of the *Operational Library Characteristics* and fill it with following parameters:
  - *Data source* to be table
  - *Recurrence* to be daily
  - *Resolution* to be hours
  - *Usage* to be absolute values
  - *Values* to be 24 values of choice between 30 and 55 (MW).
- Create ChaRef object inside of the selected load and assign the created characteristic to it.

The following code shows parts of the task and some modifications may be needed to solve the exercise as described above. A code solution to the exercise is provided in the pfd-file as embedded code.

The following parts are needed in the script:

Part 1:
Define new ChaTime:

The following Python code is just an example of how this could be done:

```python
import powerfactory #importing powerfactory module
app = powerfactory.GetApplication() # defining application object

#accessing Operational Library subfolder Characteristics
charFolder = app.GetProjectFolder('chars')
app.PrintPlain(charFolder)

#Creating an object of the type ChaTime with the name LoadChar
newChar = charFolder.CreateObject('ChaTime','LoadChar')
app.PrintPlain(newChar)
```

For more information on *CreateObject* and *GetProjectFolder* methods, refer to our Python technical references.

```python
newChar.source = 0 #source table
newChar.cunit = 1 # Unit set to hours

#List of active power set points (24h/day)
L_Pvalues = [30, 32, 31, 33, 34, 38, 41, 43, 44, 45, 50, 55,
             53, 52, 50, 50, 52, 55, 56, 54, 46, 40, 35, 33 ]
#Setting list as daily characteristic
newChar.vector = L_Pvalues

newChar.usage = 2 #values are absolute values
```

This part should create a new time characteristic inside the local library. Run the script and check for errors.

Part 2:
Access the load and assign the created ChaTime as an Active Power characteristic.

```python
#Accessing a load element
Load1 = app.GetCalcRelevantObjects('*.ElmLod')[0]
app.PrintPlain(Load1)
```

If a characteristic is assigned to an element it can be removed by deleting the reference object in the contents of the network element:

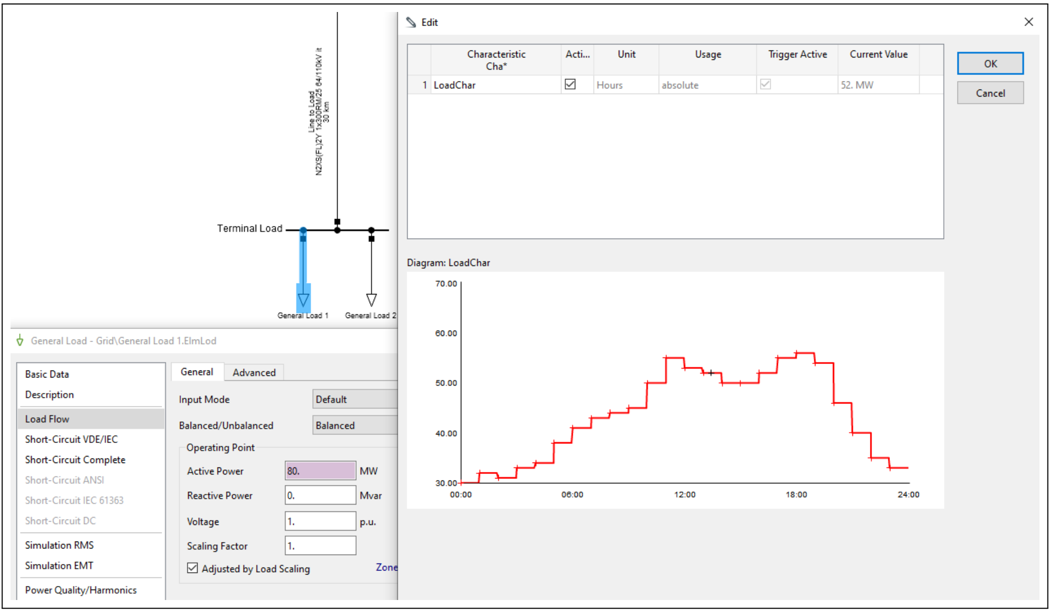
```python
#Look for all existing characteristics and delete it
sOld = Load1.GetContents('*.ChaRef')
for char in sOld:
    char.Delete()
```

The symbol * replaces the missing string part. In this case with **.Cha** we will get all objects that contain *Cha* as a part of their class.

```python
#Create ChaRef object and name it plini
refObj = Load1.CreateObject('ChaRef','plini')
```

```
app.PrintPlain(refObj)

#Assign created ChaTime to ChaRef
refObj.typ_id = newChar
```

After running the script look inside the load element. The parameter `plini` field should be coloured as shown below. Select the `plini` field with right mouse click and select *Edit Characteristic*. A new window will open with graphical interpretation of the characteristic (see Figure 4.1).



	Characteristic Cha*	Acti...	Unit	Usage	Trigger Active	Current Value
1	LoadChar	☑	Hours	absolute	☑	52. MW

General Load - Grid\General Load 1.ElmLod

Basic Data
Description
Load Flow
Short-Circuit VDE/IEC
Short-Circuit Complete
Short-Circuit ANSI
Short-Circuit IEC 61363
Short-Circuit DC
Simulation RMS
Simulation EMT
Power Quality/Harmonics

General | Advanced

Input Mode	Default	
Balanced/Unbalanced	Balanced	
Operating Point		
Active Power	80.	MW
Reactive Power	0.	Mvar
Voltage	1.	p.u.
Scaling Factor	1.	

☑ Adjusted by Load Scaling   Zone

Figure 4.1: Created characteristic by the script.

4.2 Load flow calculation

After importing the time characteristic in first step we will now execute a load flow calculation for just one point in time, in a separate script.

- Create a new Python script object and name it “Load flow sweep”.
- Get the “SetTime” object from the study case and set the time to 10 am (The SetTime functions are described in the scripting reference).
- Execute the load flow calculation from Python script and report the results in output window.
- Change the time of the study case to 11 am and execute a load flow again to see the changed operation point of the network.
- Write the difference of active power infeed for all loads between the two hours into the output window.

Part of the possible solution code:

```python
import powerfactory
app = powerfactory.GetApplication()

studyTimeObject = app.GetFromStudyCase('SetTime')
#set the time to  10am
studyTimeObject.SetTime(10,00,00)#HH,MM,SS
```

4.3 Quasi-dynamic simulation

In this section it will be shown how to set and run Quasi-dynamic simulation and how to graphically represent the results inside *PowerFactory*. For this exercise basic knowledge of plots in *PowerFactory* is needed. In order to execute a Quasi-dynamic simulation, the corresponding licence is required. So the execution of the first part of the exercise is only possible if this licence is available. For the second part, the plotting of results, the required result file is provided in “PythonTutorial_AdditionalExercise1.pdf”.

4.3.1 Executing a Calculation to write a Result File

Task:

- Import powerfactory and call the application object
- Get the quasi-dynamic simulation command object (ComStatsim) from the study case and run it with the following settings:
  - Time Period: Complete Day
  - Step Size: Step = 1
  - Step Size: Unit = Hours

The following simple script is just an example of how this can be done.

As in the previous exercises, we will access the calculation command using the GetFromStudyCase() method and use the Execute() method to execute it.

```python
import powerfactory
app = powerfactory.GetApplication()
dynLdf = app.GetFromStudyCase('ComStatsim')
dynLdf.calcPeriod = 0  #Time period = Complete Day
dynLdf.stepSize = 1    #Step Size Step = 1
dynLdf.stepUnit = 2    #Step Size Unit = Hours
dynLdf.Execute()
```

4.3.2 Displaying Simulation Results in *PowerFactory*

This subsection focusses on accessing the result file from the previously executed Quasi-dynamic simulation and creating plots to display the simulation results. The code can be added to the script from the previous subsection or written in a separate script.

- Switch to the study case “Plotting results”. Here a valid result file is provided.

- Create a Python script object and name it “PlottingResults”
- Import powerfactory and call the application object
- Get the result file from the study case.
- Get all calculation relevant loads from the result file and make sure the variables for the active power infeed are recorded in the result file.
- Create a new plot page with a new diagram.
- Get the data series object from the diagram and clear all curves.
- Add the variable for the active power infeed (m:P:bus1) for all loads to the data series.
- Autoscale the plot to make sure the simulation curves are visible.

Partial Code Solution:

```python
#access to file that contains results of calculation
dynLdf = app.GetFromStudyCase('ComStatsim')
res = dynLdf.results
# plot results
oGraph = app.GetGraphicsBoard()
oPage = oGraph.GetPage('QuasyDynSim',1)
oPlot = oPage.GetOrInsertCurvePlot('Loads',1)
oDataSeries = oPlot.GetDataSeries()
oDataSeries.ClearCurves()
for load in app.GetCalcRelevantObjects("ElmLod"):
  oDataSeries.AddCurve(load,'m:P:bus1')
```

There can be several plots on one page or several pages in the study case:

- Extend your code to plot the generator infeed (Synchronous machine, Static generator and external grid) on a second plot on the same page (parameter m:P:bus1).
- Create a second plot page with one plot displaying the loadings of all lines (parameter c:loading)

4.4 Exporting plots from Graphic Board

In many cases a project contains many diagrams that we have to export from *PowerFactory* in order to use them in different reports. One way would be to do this manually by going *File* → *Export*→ *Graphic* and selecting where this plot should be exported. This can be done easily, but if the number of plots that has to be exported is large this will be faster via a script.

Thus, our task here is to create a script that will find all available plots in a study case and export each plot to the desktop. The single line diagram of the network should also be exported.

Note: This script could be extended to loop through different study cases or projects if needed.

Create a Python script object called “ExportPlots” and write the code to export the graphics. The following steps should be followed:

- Import the powerfactory module and get the application object
- Get the graphics board (GetGraphicsBoard()) from the active study case

- Look for plot pages (GrpPage) and the pages with network diagrams (SetDeskpage) in the active study case(GetContents()).
- Print how many pages of each category are found.
- Export the pages (Functions: GraphicsBoard.Show() and WriteWMF()) by using an input parameter in the Python script object

Partial code solution:

```
gb = app.GetGraphicsBoard()
activeStudyCase = app.GetActiveStudyCase()

# Get Plot Pages
plotPages = gb.GetContents("*.GrpPage")
# Get network diagrams
networkDiagrams = gb.GetContents("*.SetDeskpage")
```

It is also possible to export diagrams to other file formats. To this end, a *ComWr* object needs to be added to the contents of the script.

```
# Save first plot page to pdf format
gb.Show(plotPages[0])
# A ComWr object must be stored in script oject
# 'pdf' format must be set in the ComWr object
SaveFile = script.GetContents("*.ComWr")[0]
SaveFile.f = path+"\\"+page.loc_name +".pdf"
SaveFile.Execute()
```

5 Additional *PowerFactory* Documentation

There is additional information available, depending on what you are looking for; the following documents can be accessed either directly from *PowerFactory* or from the *DIgSILENT* download area (`https://www.digsilent.de/en/downloads.html`).

- **Tutorials:** step by step description of several tasks in *PowerFactory*. *Help → Tutorial...*
- **Examples:** the window *PowerFactory* Examples provides a list of application examples of *PowerFactory* calculation functions. Every example comes with an explanatory document. Additionally, videos demostrate the software handling and its functionalities. *File → Examples...*
- **User Manual:** all the functions, objects and settings of *PowerFactory* are described in the User Manual. Pressing the key **F1** while working with *PowerFactory* will lead directly to the related topic inside the User Manual. *Help → User Manual*
- **Technical References:** description of the models implemented in *PowerFactory* for the different power systems components. *Help → Technical References*
- **What's New:** document and video provided with every annual release. *Help → What's New*
- **Release Notes:** for all new versions and updates of the program *Release Notes* are provided, which document the implemented changes. *Help → Release Notes*
- **Knowledge base:** a database of information, based on an FAQ format, available for any users (whether registered or not) in `https://www.digsilent.de/en/faq-powerfactory.html`
- **Scripting References:** description and examples of DPL and Python commands. *Help → Scripting References*
- **Additional Packages:** documents with additional description and /or examples of the *PowerFactory* Interfaces. *Help → Additional Packages*

Apart from the mentioned documentation, *DIgSILENT* provides **Direct Technical Support**, where *PowerFactory* experts offer direct assistance to registered users with valid guarantee/maintenance. The Support Centre is located on the website `https://www.digsilent.de/en/support.html`.

For more information, visit
www.digsilent.de

DIgSILENT GmbH
Heinrich-Hertz-Straße 9
72810 Gomaringen (Germany)
T: +49 7072 9168-0
mail@digsilent.de

DIgSILENT GmbH is certified
to the ISO 9001:2015 standard.
More information is available at
www.tuv-sud.com/ms-cert.

© DIgSILENT GmbH