

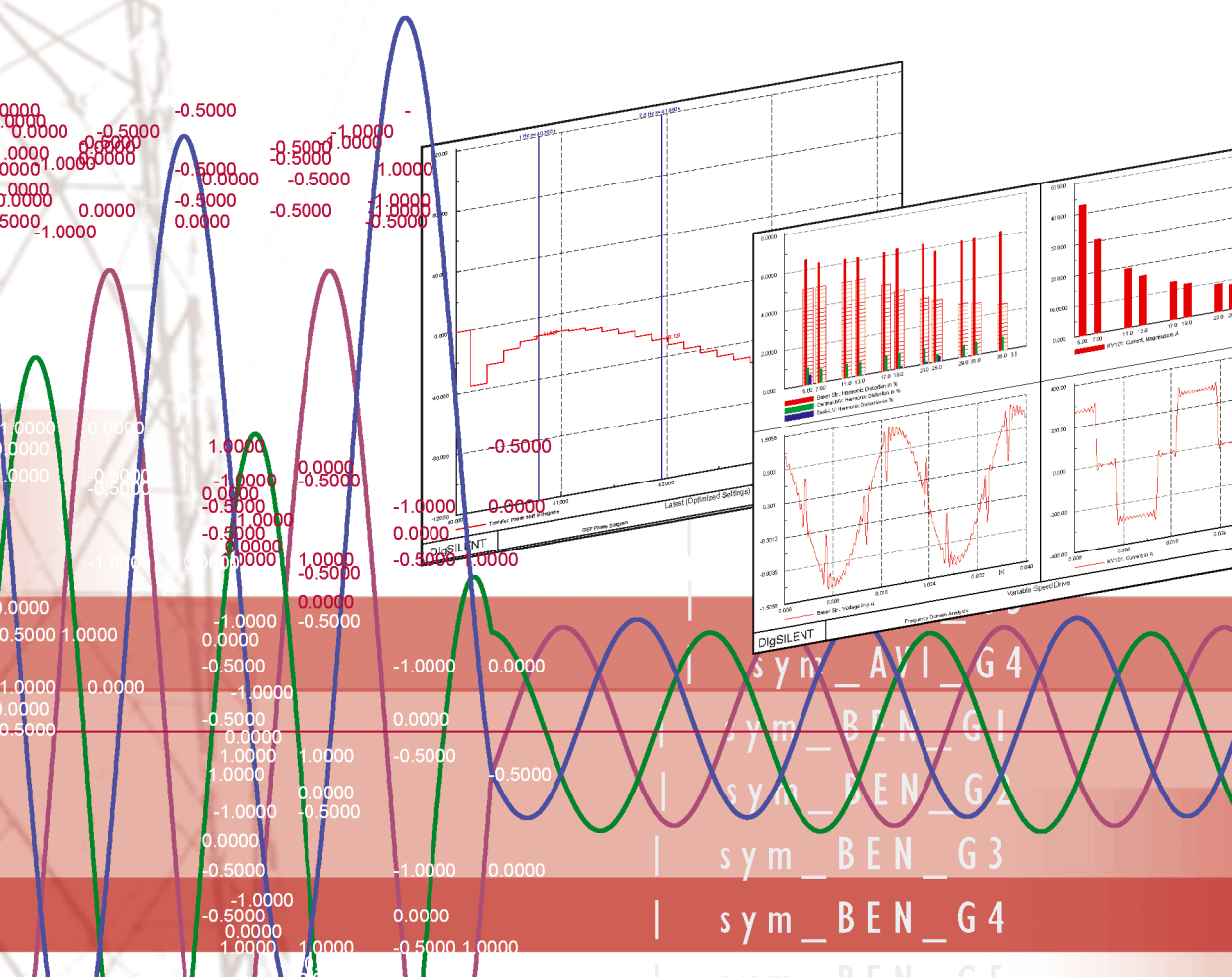
# DigSILENT *PowerFactory*

## Application Guide

PowerFactory Version 15.2

## Python Tutorial

DigSILENT Technical Documentation





**DlgSILENT GmbH**

Heinrich-Hertz-Str. 9  
72810 - Gomaringen  
Germany

T: +49 7072 9168 00  
F: +49 7072 9168 88

<http://www.digsilent.de>  
[info@digsilent.de](mailto:info@digsilent.de)

Revision r1530

Copyright ©2014, DlgSILENT GmbH. Copyright of this document belongs to DlgSILENT GmbH.  
No part of this document may be reproduced, copied, or transmitted in any form, by any means  
electronic or mechanical, without the prior written permission of DlgSILENT GmbH.

## Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Getting Started in Python-<i>PowerFactory</i>-A Quick Tutorial</b> | <b>5</b>  |
| 1.1      | Creating a small sample project . . . . .                             | 5         |
| 1.2      | Create a Python command object . . . . .                              | 6         |
| 1.3      | Write the Python Script . . . . .                                     | 7         |
| 1.3.1    | How to start a Python script . . . . .                                | 8         |
| 1.4      | Example . . . . .   | 8         |
| <b>2</b> | <b>Anatomy of a Python Object</b>                                     | <b>10</b> |
| <b>3</b> | <b>Basic Python Scripting</b>   | <b>13</b> |
| 3.1      | Accessing Network Objects . . . . .                                   | 13        |
| 3.2      | Identifying, Accessing and Modifying Object Parameters . . . . .      | 13        |
| 3.2.1    | Identifying Variable Names for a Parameter . . . . .                  | 13        |
| 3.2.2    | Accessing Parameter . . . . .   | 14        |
| 3.2.3    | Modifying Parameters . . . . .  | 14        |
| 3.3      | Creating new objects . . . . .  | 14        |
| 3.3.1    | Copy from an Internal Template . . . . .                              | 15        |
| 3.3.2    | Copy from an External Template . . . . .                              | 16        |
| 3.3.3    | Create a New Object by Code . . . . .                                 | 17        |
| 3.4      | Navigate Folders and Object Contents . . . . .                        | 18        |
| 3.4.1    | Project Folders . . . . .   | 18        |
| 3.4.2    | Object Contents . . . . .   | 18        |
| 3.4.3    | Objects in a Study Case . . . . .                                     | 19        |
| 3.5      | Accessing Study Cases . . . . .                                       | 19        |
| 3.6      | Executing Calculations . . . . .                                      | 20        |
| 3.7      | Accessing Results . . . . .   | 20        |
| 3.7.1    | Static Calculations (Load Flow, Short Circuit, etc) . . . . .         | 20        |
| 3.7.2    | Dynamic Simulations . . . . .   | 21        |
| 3.8      | Writing to the Output Window . . . . .                                | 21        |
| 3.9      | Plotting Results . . . . .  | 22        |

|          |  |           |
|----------|--|-----------|
| 3.9.1    | Creating a New Virtual instrument Page . . . . .     | 22        |
| 3.9.2    | Creating a Virtual Instrument . . . . .              | 22        |
| 3.9.3    | Adding Objects and Variables to Plots . . . . .      | 23        |
| 3.9.4    | Plotting Example . . . . .                           | 23        |
| <b>4</b> | <b>Advanced Python Scripting</b>                     | <b>25</b> |
| 4.1      | Topological Search . . . . .                         | 25        |
| 4.2      | Reading from and Writing to External Files . . . . . | 27        |
| 4.2.1    | Standard File I/O Methods . . . . .                  | 27        |
| 4.2.2    | Exporting WMF Graphic Files . . . . .                | 27        |
| 4.3      | Fast Fourier Transformation . . . . .                | 27        |
| 4.3.1    | Specifying Input and Output Results Files . . . . .  | 28        |
| 4.3.2    | Executing the FFT Calculation . . . . .              | 28        |
| 4.3.3    | Accessing the FFT Results . . . . .                  | 29        |
| <b>5</b> | <b>Working with Results Files</b>                    | <b>30</b> |
| 5.1      | Adding Results Files to the Python Script . . . . .  | 30        |
| 5.2      | Structure of Results Files . . . . .                 | 30        |
| 5.3      | Loading a Results File into Memory . . . . .         | 31        |
| 5.4      | Getting the Relevant Column Number . . . . .         | 32        |
| 5.5      | Getting Data from the Result File . . . . .          | 32        |
| 5.6      | Finding the Number of Time Intervals . . . . .       | 32        |
| 5.7      | A Simple Example . . . . .                           | 32        |
| <b>6</b> | <b>Working with Virtual Instrument Panels</b>        | <b>34</b> |
| 6.1      | Introduction . . . . .                               | 34        |
| 6.2      | Local Title Blocks . . . . .                         | 34        |
| 6.2.1    | Creating Local Title Blocks Manually . . . . .       | 34        |
| 6.2.2    | Creating Local Title Blocks in Python . . . . .      | 35        |
| 6.2.3    | Title Block Python Script Example . . . . .          | 36        |
| 6.3      | Exporting VI Panels to WMF . . . . .                 | 37        |
| <b>7</b> | <b>Working with Graphical Objects</b>                | <b>39</b> |

|  |    |
|--|----|
| 7.1 Graphical Objects in <i>PowerFactory</i> . . . . . | 39 |
| 7.2 Creating Graphical Objects in Python . . . . .     | 41 |

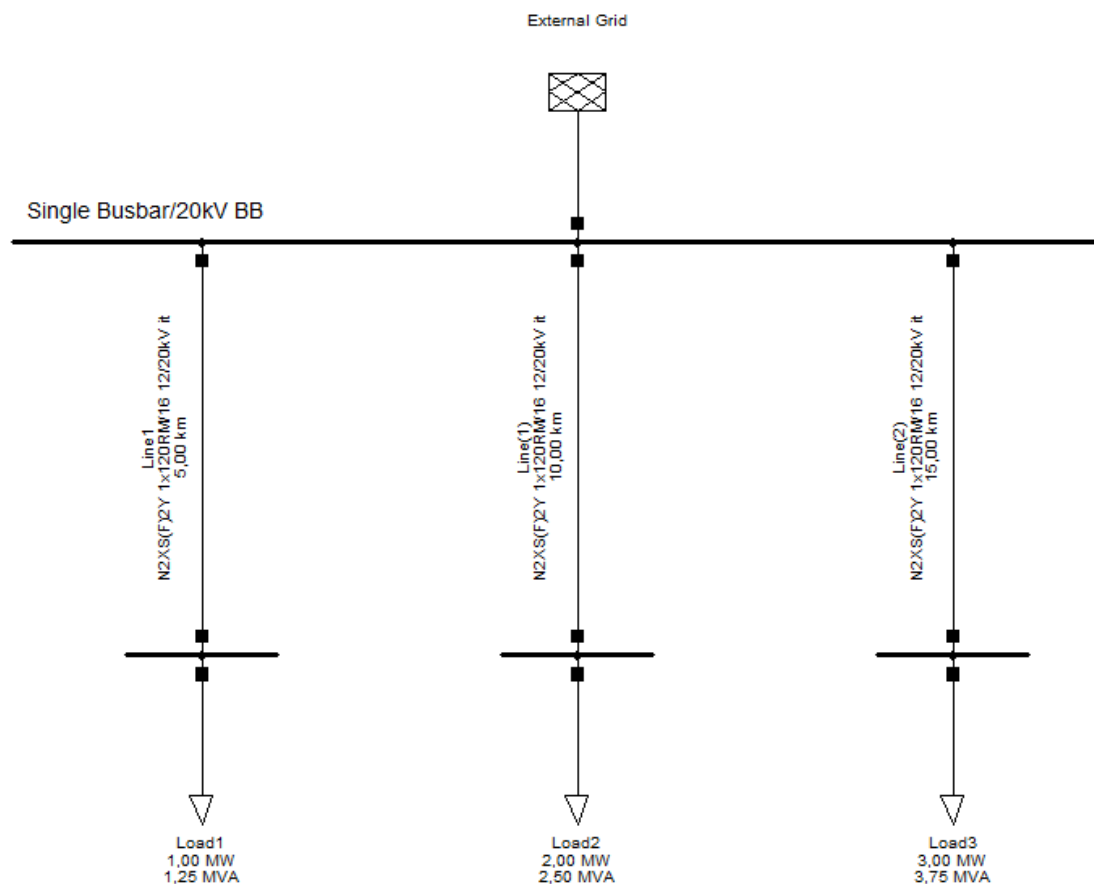
# 1 Getting Started in Python-*PowerFactory*-A Quick Tutorial

This is a short step-by-step tutorial that attempts to get the user started in writing and running Python scripts. Before going through this tutorial, the user should already have a basic grasp of *PowerFactory* handling and performing simple tasks such as load flows and short circuit simulations.

Python is not directly installed on host computer by installing *PowerFactory*. This means user have to install Python separately. Details on Python installation you may find in *PowerFactory* user Manual.

## 1.1 Creating a small sample project

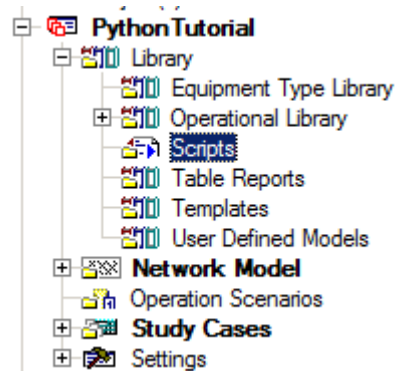
Firstly, we will create a small sample project. Create a new project and draw the following simple 20kV network using line types from the global library:



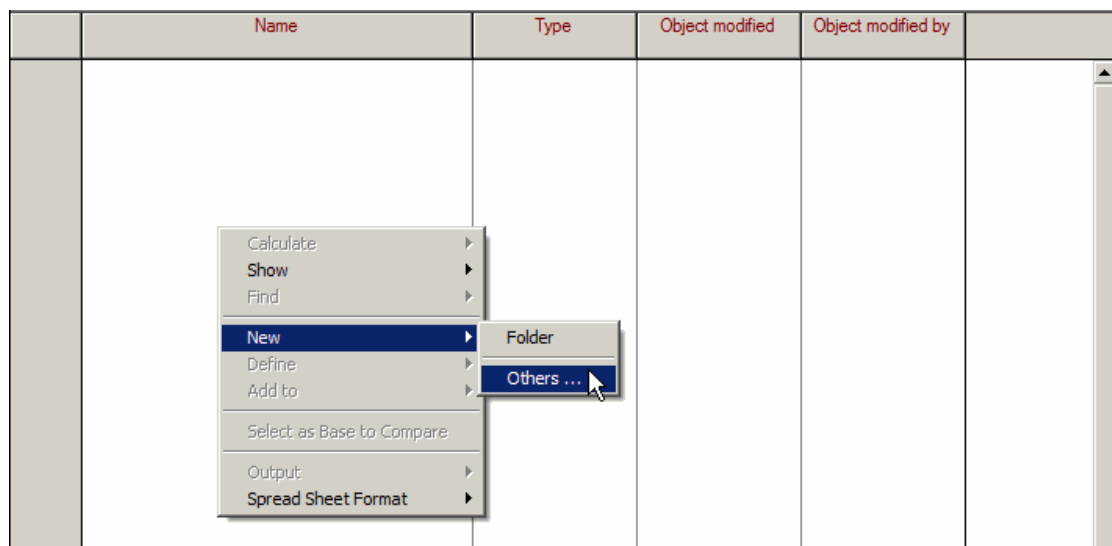
Run a load flow to make sure the model works.

## 1.2 Create a Python command object

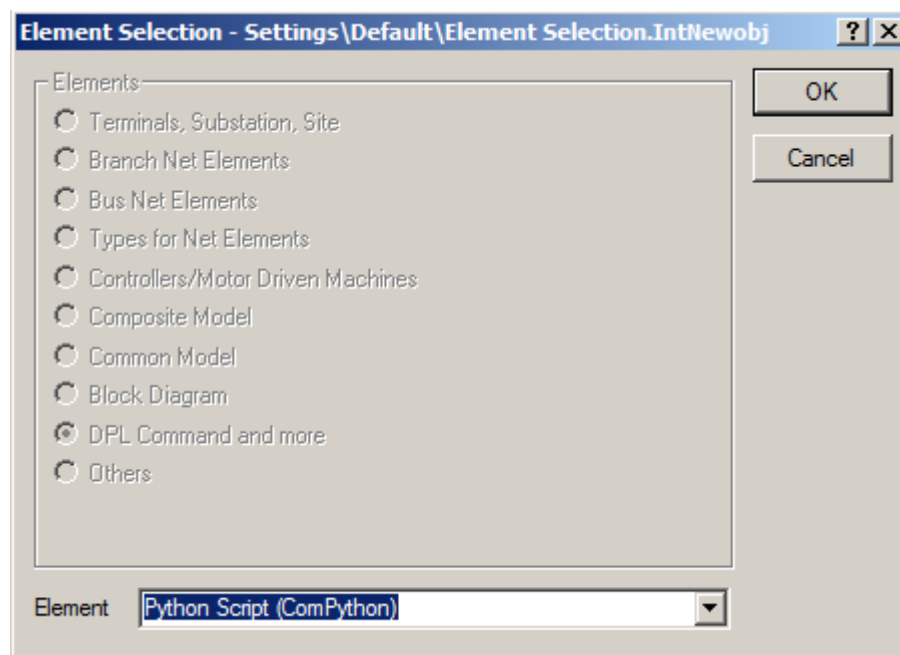
- Go to the scripts folder of the project library:



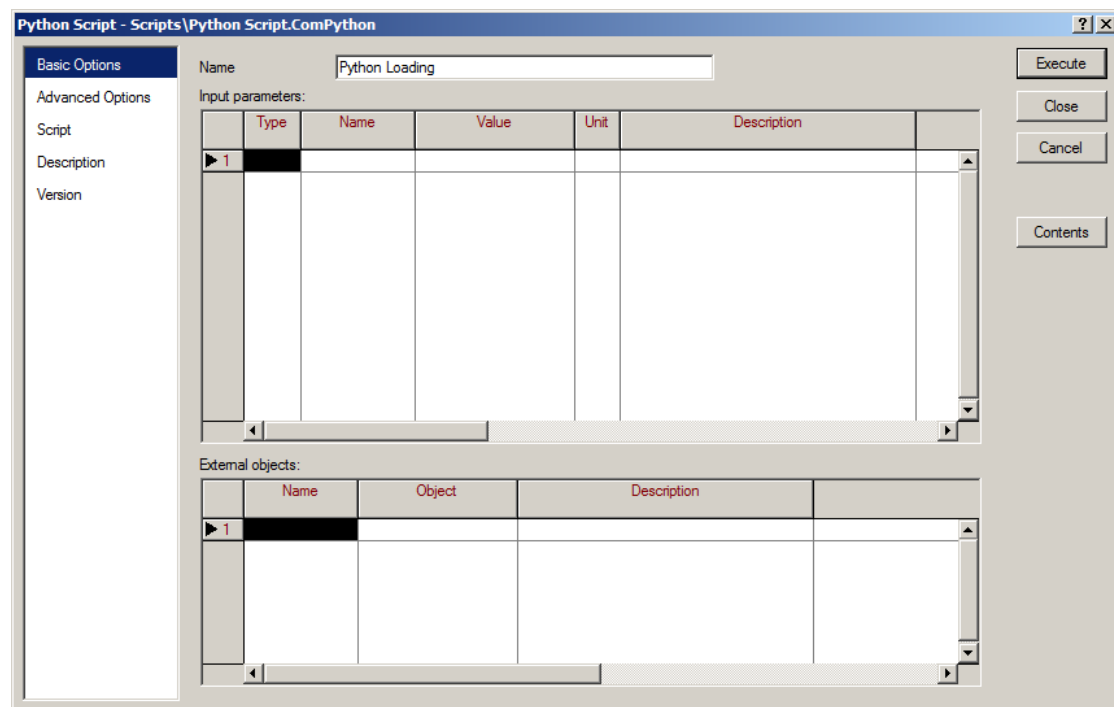
- Right-click anywhere in the data window and select *New* → *Others...*



- Select “Python Script (ComPython)” and press OK:



- Call the script “Python Loading”:

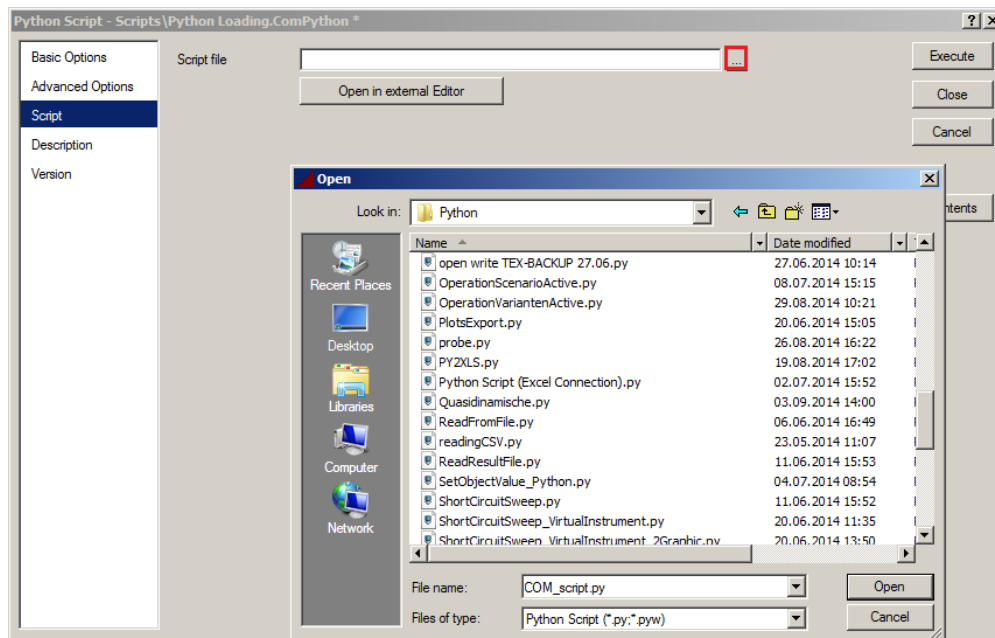


### 1.3 Write the Python Script

In contrast to DPL, Python command object *ComPython* does not contain the script's code, it only links the Python script file .py. It stores only the file path of the script and not the code itself.



- Click on the “Script” tab in the Python command object.
- By selecting the button “...”, a new pop-up window will appear. The user can select the .py file where his Python script is saved (if not already existing a new .py file may be created).



### 1.3.1 How to start a Python script

To allow Python to have access to *PowerFactory*, “powerfactory” module must be imported:

```
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)
- Most special built-in functions (DPL functions)

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

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

## 1.4 Example

We will now write a Python script that will execute a load flow and print out the name of each line and its corresponding loading in the output window. Type the following code into the script file:

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

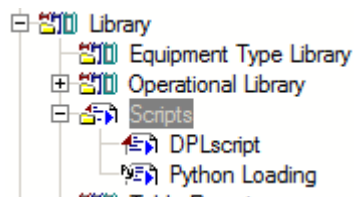
- Link the .py script file and the Python command object (see section 1.3)
- Click “Save” and then “Execute”. The following results should be seen in the output window:

```
DigSI/info - Python Script 'Python Loading' started
DigSI/info - Element 'External Grid' is local reference in separated area of '1'
DigSI/info - Calculating load flow...
DigSI/info - -----
DigSI/info - Start Newton-Raphson Algorithm...
DigSI/info - load flow iteration: 1
DigSI/info - load flow iteration: 2
DigSI/info - Newton-Raphson converged with 2 iterations.
DigSI/info - Load flow calculation successful.
DigSI/info - -----
DigSI/info - Report of Control Condition for Relevant Controllers
DigSI/info - -----
DigSI/info - Control conditions for all controllers of interest are fulfilled.
Loadiang of the line: Line1 = 9.86 %
Loadiang of the line: Line2 = 19.91 %
Loadiang of the line: Line3 = 30.35 %
DigSI/info - Python Script 'Python Loading' successfully executed
DigSI/info - DPL Program 'DPLscript' started
DigSI/info - DPL program 'DPLscript' successfully executed
```

## 2 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. 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 tabs : Basic Options, Advanced Options, Script, Description and Version. We will go through each tab and the functions that are available.

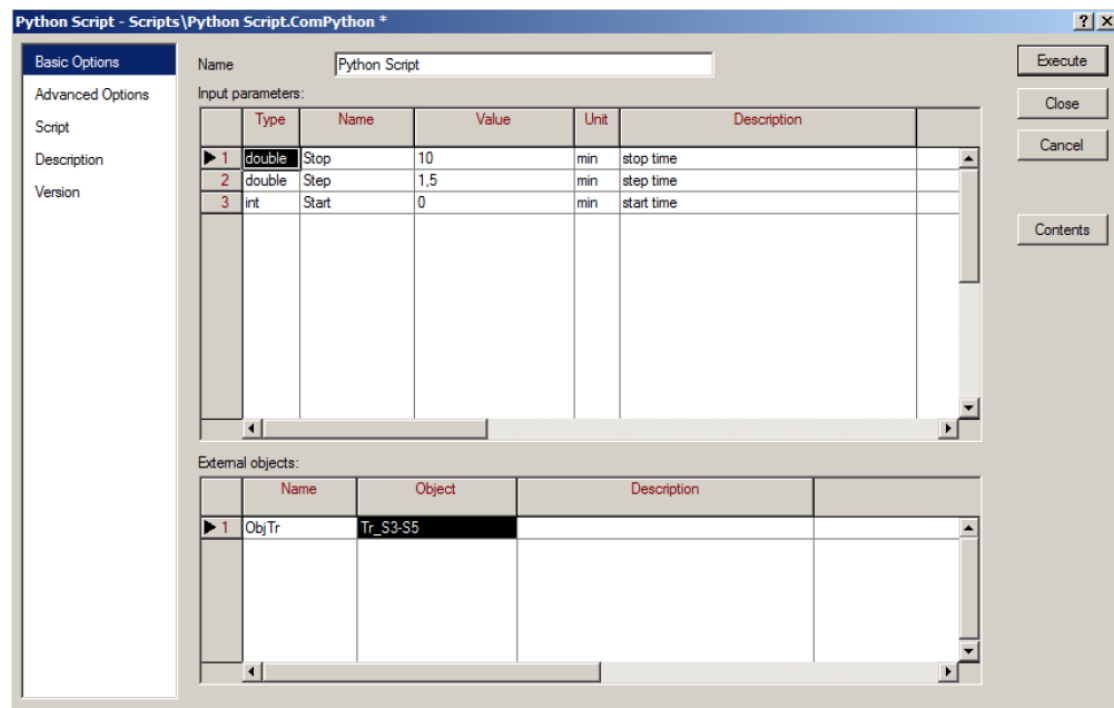


Figure 2.1: Python command object “Basic Option” page

The **Basic Options** tab (shown above) has the following functions:

- Name field - Changes the name of the Python command object
- Define internal input parameters - in the script, the names of the input parameters can be used like variables and do not need to be redefined

- Define external objects - these are objects external to the Python command object that you want to access via the script. This can be any object, e.g. an element in the network model, a study case, an equipment type, etc. Inside the script, the object can be used like a variable by calling the name defined here.

**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 to access the object/parameter through `ScriptObj.ExternObjName`:

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

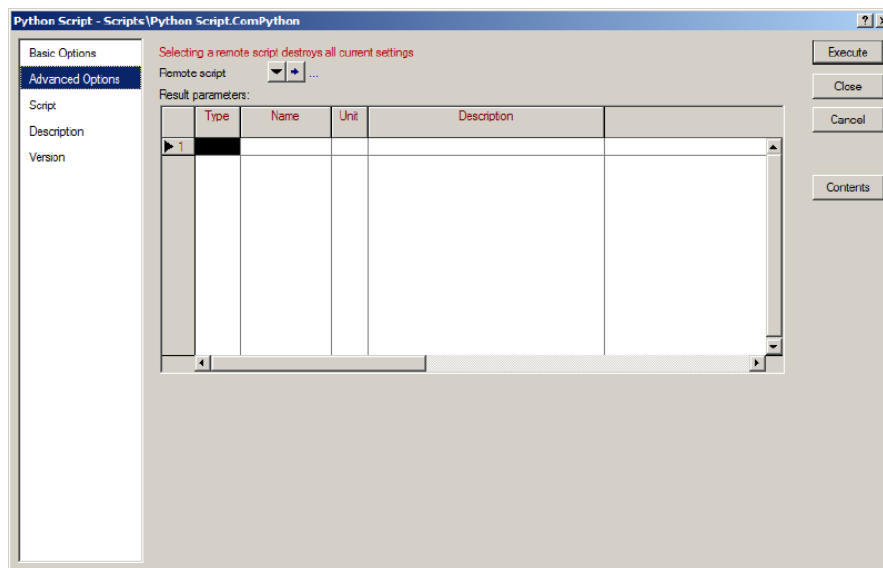



Figure 2.2: ComPython Object, Advanced Options tab

The **Advanced Options** tab (shown below) has the following functions:

- Select 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.
- 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 **Description** and **Version** tabs are informational tabs 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 dialogue 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. An example showing the possible contents of a Python command object is shown in Figure 2.3.

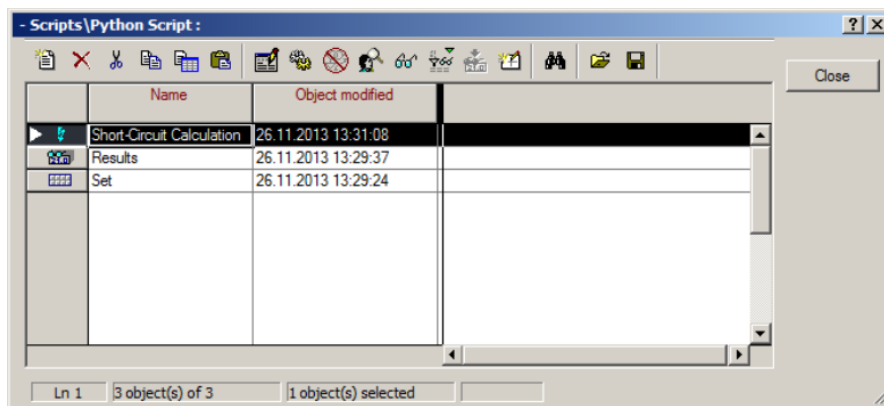


Figure 2.3: Contents of a Python command object

## 3 Basic Python Scripting

### 3.1 Accessing 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 full name:

```
import powerfactory
app=powerfactory.GetApplication()
Lines=app.GetCalcRelevantObjects('*.ElmLne') #get list of all lines
for line in Lines:
    app.PrintPlain(line.loc_name)
```

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

```
import powerfactory
app=powerfactory.GetApplication()
AllObj=app.GetCalcRelevantObjects() #get list of all objects
Line=app.GetCalcRelevantObjects('Line1.ElmLne') #get a line 'Line1'
app.PrintPlain(Line[0].loc_name)
```

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.

```
import powerfactory
app=powerfactory.GetApplication()
AllObj=app.GetCalcRelevantObjects() #get list of all objects
#Filter the set with all lines starting with 'Line'
Lines=app.GetCalcRelevantObjects('Line*.ElmLne')
for Line in Lines:
    app.PrintPlain(Lines.loc_name)
```

### 3.2 Identifying, Accessing and Modifying Object Parameters

Once a specific object has been selected, the way to access the object parameters or variables is by typing out the object name and the variable name separated by a point “.”, e.g.

Object\_name.Variable\_name  
for example  
NameOfALine=Line.loc\_name

#### 3.2.1 Identifying 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 dialogue 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”:

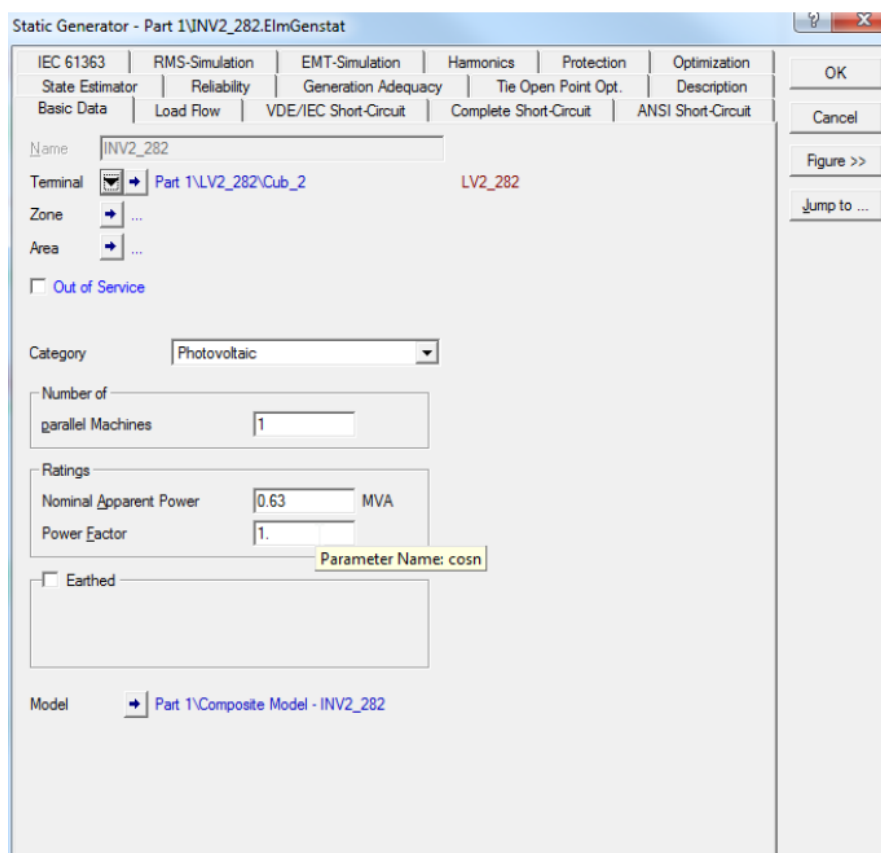


Figure 3.1: Static Generator “Basic Data” page

### 3.2.2 Accessing Parameter

Suppose we have a line object “oLine” 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=oLine.dline
```

### 3.2.3 Modifying Parameters

Suppose we have a line object oLine and we want to change the length of the line (variable name = dline) to 2km, the instruction below may be used:

```
oLine.dline=2
```

## 3.3 Creating new objects

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

- Copy template object contained inside the script object

- Copy external template object
- Create new object from scratch in code

### 3.3.1 Copy from an Internal Template

Copying an internal template object is potentially the easiest option for creating new objects. The Python command object can contain other objects within it. To access the contents of a Python object, click on the `smarksContents` button of the Python command dialogue:

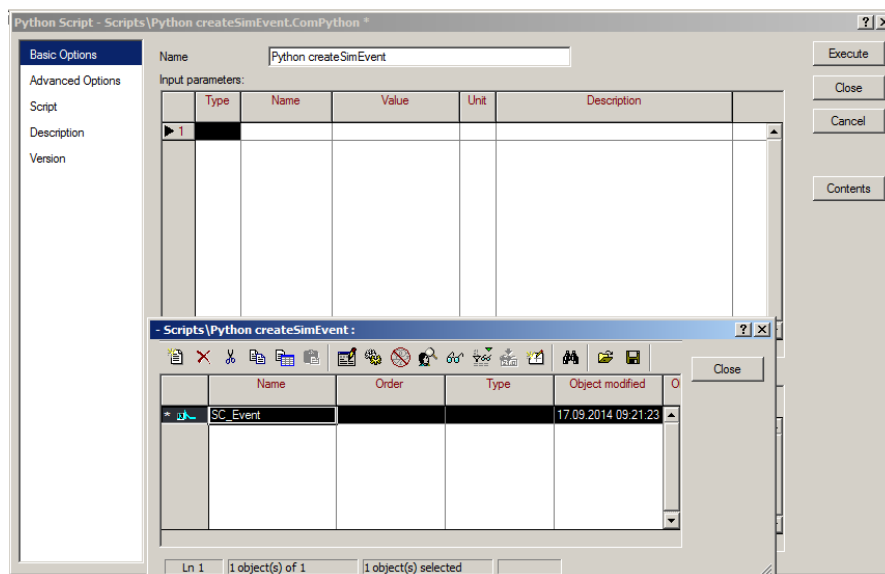


Figure 3.2: ComPython internal template object

You can use an internal object inside the Python command object as a template for the object type you want to create. In the figure above, an event (EvtShc) object named "SC\_Event" is placed inside the Python command object and used as template.

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 template into.

For example, suppose "oFold" is an object with the reference to the target folder and "SC\_Event" is the internal template object, a new object will be created in the target folder by the following command:

```
oFold.AddCopy(SCEvent)
```

The code snippet below copies an internal short circuit event template into the simulation events folder of the active study case and then changes the time of the copied 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)

#Get Internal template object
```



```
Script=app.GetCurrentScript()
Contents=Script.GetContents()
SC_Event=Contents[0]

#Copy the template short circuit event into the events folder
oFold.AddCopy(SC_Event)

#Get the copied event and set the time to 1
EventSet=oFold.GetContents('*.EvtShc')
app.PrintPlain(EventSet)
oEvent=EventSet[0][0]
oEvent.time=1
```

### 3.3.2 Copy from an External Template

This procedure is almost identical to copying an object from an internal template, except that an external object is referenced instead of using an object from inside the Python command. In the Python command dialogue (under “Basic Options”), there is an area for referencing external objects:

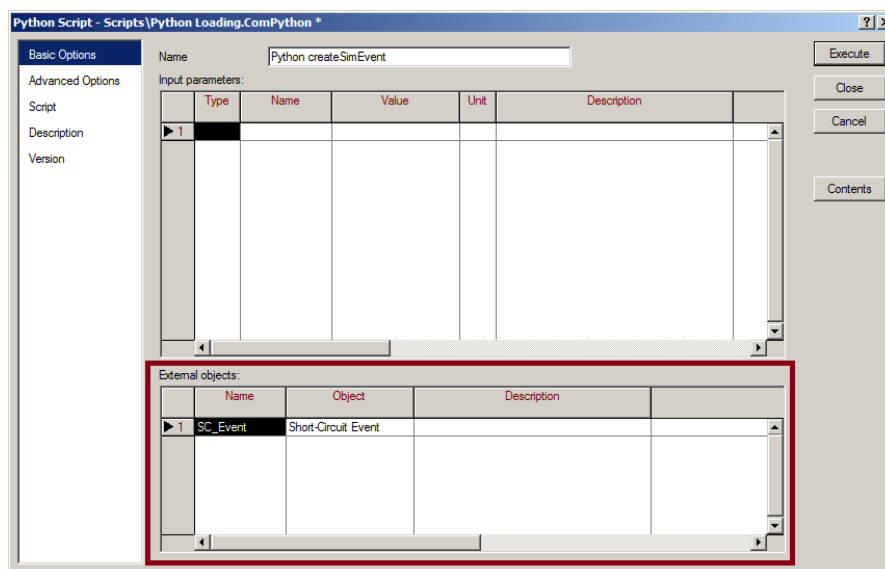


Figure 3.3: Python Command Object (ComPython) Dialogue - External Object

To use this area, simply double-click on the object field and select an object you would like to reference from anywhere in the project hierarchy. One distinction of this method is that you can give the object any name (using the “Name” field). This is the name that the object will be called inside the Python script.

Once the external template object has been selected, you can use the **AddCopy(object.name)** command to create a new object in the database. In order to use this command, you must first locate the folder (or object) that you want to copy the template into.

For example, suppose “oFold” is an object with the reference to the target folder and “SC.Event” is the external template object, a new object will be created in the target folder by the following:

```
oFold.AddCopy(SC_Event)
```

The code snippet below copies an external short circuit event template into the simulation events folder of the active study case and then changes the time of the copied 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)

#Get External template object
Script=app.GetCurrentScript()
SC_Event=Script.SC_Event

#Copy the template short circuit event into the events folder
oFold.AddCopy(SC_Event)

#Get the copied event and set the time to 1
EventSet=oFold.GetContents('*.EvtShc')
app.PrintPlain(EventSet)
oEvent=EventSet[0][0]
oEvent.time=1
```

### 3.3.3 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 need 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. Like the previous two methods, you must first locate the folder (or object) that you want to create the object in.

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

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

#Get the copied event and set the time to 1
EventSet=oFold.GetContents('*.EvtShc')
app.PrintPlain(EventSet)
```

```
oEvent=EventSet[0][0]
oEvent.time=1
```

## 3.4 Navigate Folders and Object Contents

### 3.4.1 Project Folders

As an entry or starting point into the project folder hierarchy, use the **GetProjectFolder(string)** function. 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 list of the project folders available and the corresponding string is shown below:

| String | Folder Description     |
|--------|------------------------|
| equip  | Equipment type library |
| netmod | Network model          |
| oplib  | Operational library    |
| 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                  |

### 3.4.2 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 that meet the criteria in the string.

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')
```

### 3.4.3 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 however that this function only works with the active study case.

The code snippet below gets the graphics board object from the active study case.

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

oDesk=app.GetFromStudyCase('SetDesktop')
app.PrintPlain(oDesk)
```

## 3.5 Accessing 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.

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

#Get the study case folder and its contents
aFolder=app.GetProjectFolder('study')
aCases=aFolder.GetContents('*.IntCase')
aCases=aCases[0]
#Counts the number of study cases
iCases=len(aCases)
if iCases==0:
    app.PrintError('There is 0 of study cases')
else:
```

```

app.PrintPlain('Number of cases: %d' %(iCases))
app.PrintPlain(aCases)
#Cycle through all study cases and activate each
for aCase in aCases:
    app.PrintPlain(aCase)
    app.PrintPlain(aCase.loc_name)
    aCase.Activate

```

Note that if there are subfolders within the study case folder, then further processing needs to be done to access the study case objects within these subfolders. For more information please refer to Navigating Project Folders in section 3.4.

### 3.6 Executing Calculations

The **GetFromStudyCase(string)** can be used to get an existing or create a new calculation command object. The **Execute()** object function can then be used to execute the calculation. The following list of calculation command objects are available:

|                 |                                     |
|-----------------|-------------------------------------|
| ComLdf          | Load flow                           |
| ComShc          | Short circuit                       |
| ComSim          | Time domain (RMS or EMT) simulation |
| ComInc          | Time domain initial conditions      |
| ComSimoutage    | Contingency analysis                |
| ComRel3         | Reliability assessment              |
| ComMod          | Modal analysis                      |
| ComHldf         | Harmonic load flow                  |
| ComGenrelinc    | Initialise generation adequacy      |
| ComGenrel       | Run generation adequacy             |
| ComCapo         | Optimal capacitor placement         |
| ComVstab        | Load flow sensiivities              |
| ComRed          | Network reduction                   |
| ComVsag         | Voltage sag table assessment        |
| ComCabsiz       | Cable reinforcement optimisation    |
| ComTieopt       | Tie open point optimisation         |
| ComSe           | State estimator                     |
| ComFlickermeter | Flickermeter                        |

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

```

import powerfactory
app=powerfactory.GetApplication()

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

```

### 3.7 Accessing Results

#### 3.7.1 Static Calculations (Load Flow, Short Circuit, etc)

The results of static calculations are stored as parameters in the objects themselves. To get the value of these results you can use **GetAttribute(string)** 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 “LoadLine”:

```
LoadLine=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('*.ElmLine')
for line in lines:
    name=line.loc_name
    value=line.GetAttribute('c:loading')
    app.PrintPlain('Loading of the : %s = %.2f %%%' % (name,value))
```

### 3.7.2 Dynamic Simulations

Unlike the static cases, the results of dynamic simulations are not stored as part of the object parameter, but in a separate results file “.ElmRes”. Refer to the section 5 for more information.

## 3.8 Writing to the Output Window

Python offers a few options for writing text to the output window:

- Print a formatted string to the output window  
**app.PrintPlain(string %(variables))**  
this function works on the same way as printf in DPL

```
app.PrintPlain('Hallo world')
```

Prints the string in output window

```
app.PrintPlain('Voltage = %f' %(dVoltage))
```

Prints combination out of string and value. Note that by default, *PowerFactory* prints 6 decimal places for floating point variables. To change the number of decimal places, use the decimal place specifier in the % tag. For example, for 3 decimal places

```
app.PrintPlain('Voltage = %.3f' %(dVoltage))
```

- Print a error, warning and information message  
**app.PrintError(string)**  
**app.PrintWarn(string)**  
**app.PrintInfo(string)**

## 3.9 Plotting Results

Plots from dynamic (time-domain) or static simulations can be easily created using Python.

### 3.9.1 Creating a New Virtual instrument Page

You can create a new virtual instrument (VI) page using 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 VI page and the create flag is activated (=1).

For example, the snippet below uses the graphics board object "Desktop" to create a VI page called "Plots":

```
Desktop=app.GetFromStudyCase('SetDesktop')
Vi=Desktop.GetPage('Plots',1)
```

### 3.9.2 Creating a Virtual Instrument

Similar to creating a new VI page, you can use the VI page object (\*.SetVipage) to create a new virtual instrument with the function **GetVI (string name, string class, int create)**. To create a new VI, the name should not be the same as another existing VI and the create flag should be activated (=1).

The class of the VI is the type of VI that you want to create. If it is not entered, the class is set as a subplot (\*.VisPlot) by default. The classes for virtual instruments are shown below:

- VisPlot - Subplot
- VisPlot2 - Subplot with two y-axis
- VisXyplot - X-Y plot
- VisFft - FFT plot
- VisOcplot - Time-overcurrent plot
- VisDraw - R-X plot
- VisPlotz - Time-distance plot
- VisEigen - Eigenvalue plot
- VisModbar - Mode bar plot
- VisModphasor - Mode phasor plot
- VisVec - Vector diagram
- VisHrm - Waveform plot
- VisBdia - Distortion plot
- VisPath - Voltage profile plot
- VisVsag - Voltage sag plot
- VisPdc - Probability density plot

- VisDefcrv - Curve-input plot
- VisMeas - Measurement (scales, displays, etc)
- VisBitmap - Picture box
- VisSwitch - Button
- VisButton - Command button

```
Desktop=app.GetFromStudyCase('SetDesktop')
Vi=Desktop.GetPage('Plot',1)
oPlot=Vi[0].GetVI('SubPlot','VisPlot',1)
```

### 3.9.3 Adding Objects and Variables to Plots

In order to show the actual plots on a VI, you need to add objects and variables to the VI. Using the VI object, use the **AddVars** function to add variables and objects to the plot. The function **AddVars** can be used in one of two ways:

- **AddVars(string V, object O1,...O8)**

Here we want to add the variable V to objects O1 to O8 (i.e. variables can be added to up to 8 objects simultaneously).

For example, the snippet below adds the variable “m:u1” for the objects “oBus1” and “oBus2” to the VI object “oPlot” (oBus1 and oBus2 could be substation buses and we want to plot the voltage “m:u1” on these buses):

```
oPlot.AddVars('m:u1', oBus1, oBus2)
```

- **AddVars(object O, string V1,..V8)**

Alternatively, we can add multiple variables V1 to V8 to a single object O (i.e. up to 8 variables can be added to an object simultaneously).

For example, the snippet below adds the variables “m:u1” and “m:phi” for the object “oBus1” to the VI object “oPlot”:

```
oPlot.AddVars(oBus1, 'm:u1', 'm:phi')
```

### 3.9.4 Plotting Example

The example below gets the current graphics board, creates a new VI page called “Plots”, creates a subplot, adds the variable “m:u1” for the object “oBus” (which is defined elsewhere) and then adjusts the y-axis maximum and minimum range:

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

#Create VI Page
oViPg=oGrb.GetPage('Plats',1)

#Create a new subplot
```



```
oPlot=oViPg[0].GetVI('Subplot','VisVec',1)

#Add variable 'm:u1' for the object 'oBus'
oPlot[0].AddVars(oBus[0],'m:u1')

#Adjust y-axis min and max settings
oPlot[0].y_max=2
oPlot[0].y_min=0
```

For further details on manipulating Virtual Instrument panels in Python, please refer to section 6.

## 4 Advanced Python Scripting

### 4.1 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:

- Object Functions
  - get a list of the elements connected to an object

#### **GetConnectedElms(Breakers,disconnectors,out\_of\_service)**

Example a): `setObj=oTerm.GetConnectedElms (1,1,0)`

Gets all of elements connected to “oTerm” and puts it into the list “setObj”, taking into account the state of breakers and switch-disconnector, but disregarding out of service flags.

Example b): `setObj=oTerm.GetConnectedElms ()`

By default, the options are set to (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.

**Important Node:** when using this function to find connected terminals, one must be careful about the use of the “internal node” option in terminals.

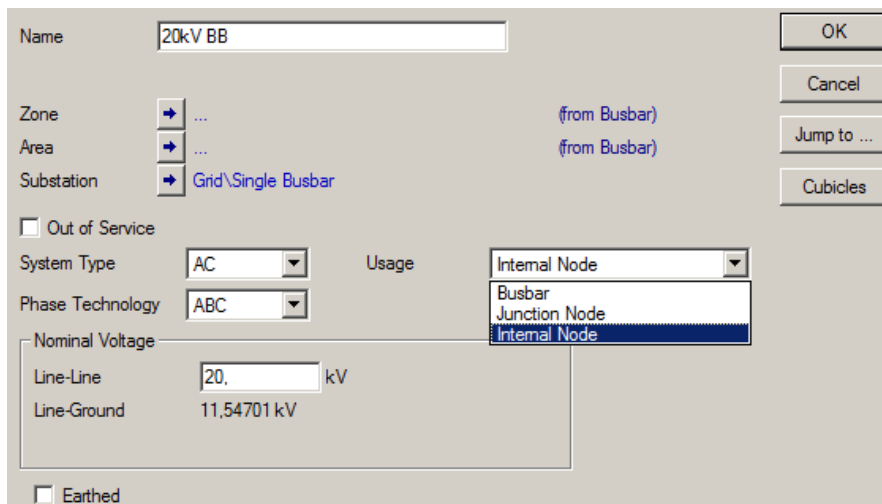


Figure 4.1: ElTerm “Basic data”

When this option is set, then `GetConnectedElms()` will ignore this terminal. This is to avoid returning the internal nodes of a subsection, and instead returning the main busbar.

- get the terminal/node connected to an object

#### **GetNode(bus\_no=0 or 1,switch\_state=0 or 1)**

Example a): `oTerm=oLine.GetNode(1,0)`

Gets the terminal connected to bus “0” of line object “oLine”, taking into account the switch state, and puts the terminal into object “oTerm”

Example b): `oTerm=oLine.GetNode(1)`

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.

- get the cubicle connected to an object

#### **GetCubicle(index)**

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

Gets the cubicle with index “1” ate line objcet “oLine” and puts it into the object “oCub”. Returns NULL if the cubicle does not exist.

- get the number of connections/ cubicles connected to an object

#### **GetConnectionCount()**

Example: `i=oLine.GetConnectionCount()`

Gets the number of connections or cubicles connected to line object “oLine” and puts it into the variable “i”

- gets name of the class (useful when filtering for certain types of objects)

#### **GetClassName()**

Example: `NameCl=obj.GetClassName()`

Gets the name of the class for the objcet “obj” and stores the result on variable “NameCl”. Returns string.

#### • Terminal Functions

- get the next busbar (at a higher nominal voltage level) connected to the terminal

#### **GetNextHVBus()**

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.

#### • Cubicle Functions

- get the list of all network elements downstream/upstream of a cubicle

#### **GetAll(direction=0(branch)or 1(terminal),switch\_state=0 or 1)**

Example a): `setObj=oCub,GetAll(1,1)`

Gets the list of all network elements starting from cubicle “oCub” in the direction

of the branch, and taking into account switch states (i.e. it will stop at an open switch).

Example b): `setObj=oCub.GetAll(0,0)`

Gets the list of all network elements starting from cubicle “oCub” in the direction of the terminal, ignoring switch states.

Example c): `setObj=oCub.GetAll()`

Without setting any of the options, the above snippet is equivalent to calling `oCub.GetAll(1,0)`, i.e. traversal in the direction of the branch ignoring switch states.

## 4.2 Reading from and Writing to External Files

### 4.2.1 Standard File I/O Methods

The standard file I/O methods for Python scripting in *PowerFactory* are the same as in Python. For more information please refer to standard Python literature e.g. <https://docs.python.org/3/>

### 4.2.2 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 VI page to the path / file “c:\temp\results1.wmf”:

```
import powerfactory
app=powerfactory.GetApplication()
oDesk=app.GetFromStudyCase('SetDesktop')
oDesk.WriteWMF('C:\Users\...\Desktop\result')
```

## 4.3 Fast Fourier Transformation

A Fast Fourier Transform (FFT) can be executed on the results of a time domain simulation using the FFT Calculation object (ComFft):

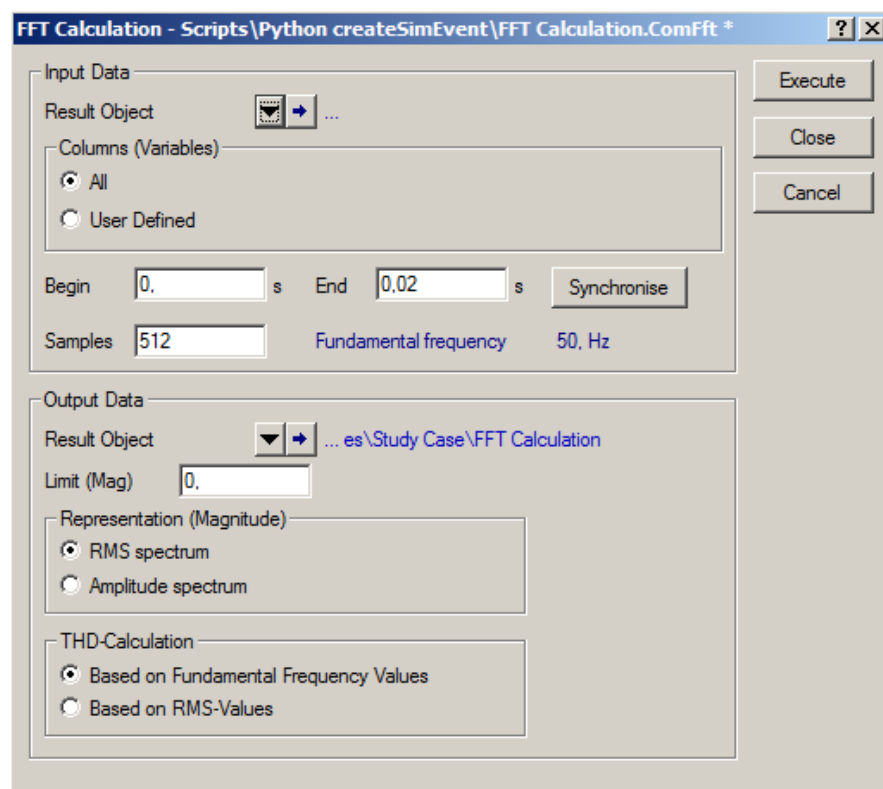


Figure 4.2: ComFft calculation object

To execute an FFT in Python, the easiest approach is to manually create the FFT Calculation object inside the Python command object with the relevant settings pre-defined. The following steps should then be followed to set up the results files, run the FFT and accessing the results.

### 4.3.1 Specifying Input and Output Results Files

The FFT calculation object requires a time domain simulation results file (ElmRes) and the FFT calculation output is stored in another results file. These input and output results files need to be specified. In the Python script, these can be specified by using the FFT calculation object variables “pRes.in” and “pRes.out”.

For example, given the FFT calculation object “FFT.Calculation”:

```
FFT.Calculation.pRes.in=EMT_Calc
FFT.Calculation.pRes.out=FFT_Results
```

### 4.3.2 Executing the FFT Calculation

To execute the FFT calculation, simply use the **Execute()** method of the FFT calculation object:

```
FFT.Calculation.Execute()
```

### 4.3.3 Accessing the FFT Results

The FFT results are stored in the output results file that was specified earlier in the FFT calculation object. How to access this result data please see Chapter 5.

## 5 Working with Results Files

Suppose you want to get a *PowerFactory* Results file (of type ElmRes) from a dynamic simulation, peer into it, pull out a set of relevant values, perform some calculations on the data and generate some outputs. How do you do that? This section describes how to use a Python script to manipulate Results files from dynamic simulations.

### 5.1 Adding Results Files to the Python Script

The easiest way to work with a results file is to save it into the Python script command object. For example, you can drag and drop an “All Calculations” results file into your script. Note that the name of the results file needs to be changed to eliminate spaces, e.g. in the screenshot below, “All Calculations” has been renamed to “All\_calcs”.

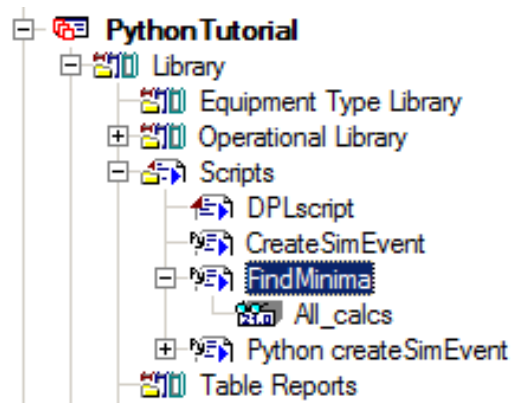
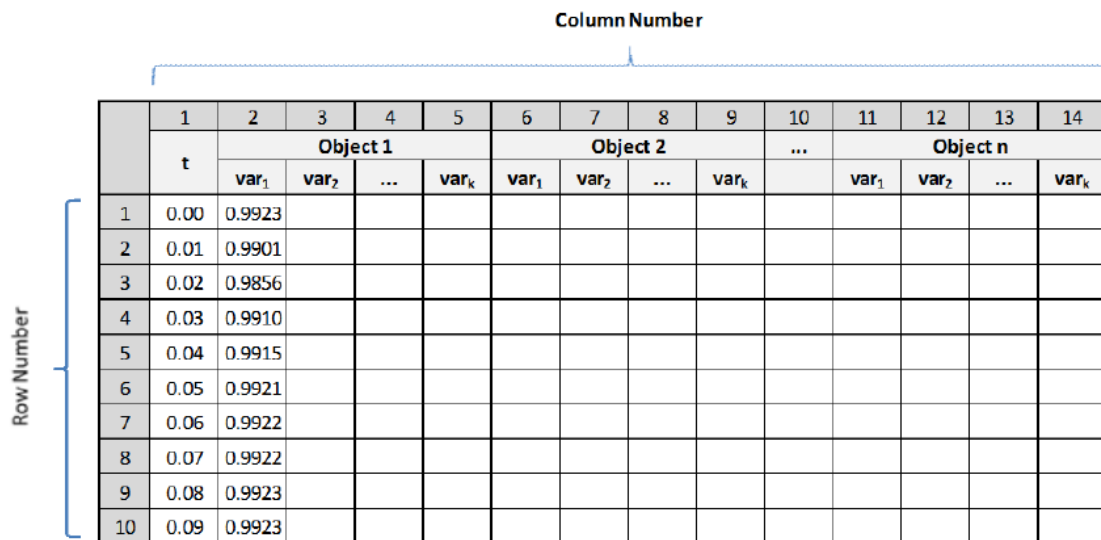


Figure 5.1: ElmRes object as contents of Python Script

Once it has been dragged or created into the script, the Results file can be called inside the Python script. All object methods and functions are exposed.

### 5.2 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.



|    | 1    | 2                | 3                | 4   | 5                | 6                | 7                | 8   | 9                | 10  | 11               | 12               | 13  | 14               |
|----|------|------------------|------------------|-----|------------------|------------------|------------------|-----|------------------|-----|------------------|------------------|-----|------------------|
|    | t    | Object 1         |                  |     |                  | Object 2         |                  |     |                  | ... | Object n         |                  |     |                  |
|    |      | var <sub>1</sub> | var <sub>2</sub> | ... | var <sub>k</sub> | var <sub>1</sub> | var <sub>2</sub> | ... | var <sub>k</sub> |     | var <sub>1</sub> | var <sub>2</sub> | ... | var <sub>k</sub> |
| 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 5.2: Structure of an ElmRes object

The number of rows represents the total number of time intervals for the simulation. For example, if there is 10s of simulation with a time step of 0.01s, then there would be 1,000 rows. Each row represents a time interval.

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 var1 to var<sub>k</sub>). 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 interval.

### 5.3 Loading a Results File into Memory

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

```
app.LoadResData(ResultsFile)
```

LoadResData loads the contents of a Results file into memory and allows you to begin accessing the data. You can only load one Results file into memory at any one time.



## 5.4 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:

```
app.ResGetIndex(Result_File, Object, Variable Name)
```

ResIndex 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)

## 5.5 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 only be accessed one at a time (i.e. you can not get a vector of data from the Results file). Use the following command to get a data point:

```
app.ResGetData(Result_File, row index, column index)
```

The row and column indices are integers.

## 5.6 Finding the Number of Time Intervals

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

```
app.ResGetValueCount(Result_File, column index)
```

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

Similar function ResGetVariableCount returns number of variables stored in Result\_File

```
app.ResGetVariableCount(Result_File)
```

## 5.7 A Simple Example

This example opens a Results file, prints number of rows and columns of it, and all values for "m:lkks:Z:bus1" variable of the an object. Note that in the example below, result file "Result" has been added into the Python script.

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

#Get Result file ElmRes
script=app.GetCurrentScript()
res=script.GetContents('Results.ElmRes')
res=res[0][0]

#Loading of Result file
app.ResLoadData(res)
```

```
#Calling of the object which results are from interest
line=app.GetCalcRelevantObjects('Line.ElmLne')

#Get number of rows and columns
NrRow=app.ResGetVariableCount(res)
NrCol=app.ResGetValueCount(res,0)

#print results
app.PrintPlain('ElmRes has %i rows and %i Columns' %(NrRow,NrCol))

#Get index of variable of interest
ColIndex=app.ResGetIndex(res,line[0],'m:Ikss:bus1')

#print results
for i in range(NrRow):
    value=app.ResGetData(res,i,ColIndex)
    app.PrintPlain(value)
```

## 6 Working with Virtual Instrument Panels

### 6.1 Introduction

Virtual instrument (VI) panels are contained in “SetVipage” objects within a graphics board, and are used to visualise the results of dynamic simulations for relevant variables of interest.

This document provides some detailed instructions on manipulating virtual instrument panel objects using the Python scripting language. Firstly some general definitions for the objects related to virtual instrument panels:

- **Graphics Bord** - is the container for all graphics, including the VI panels
- **VI Panel** - is the name of the whole virtual instrument page (\*.setVipage)
- **Visplots** - are the individual plots inside a VI panel (\*.VisPlot)

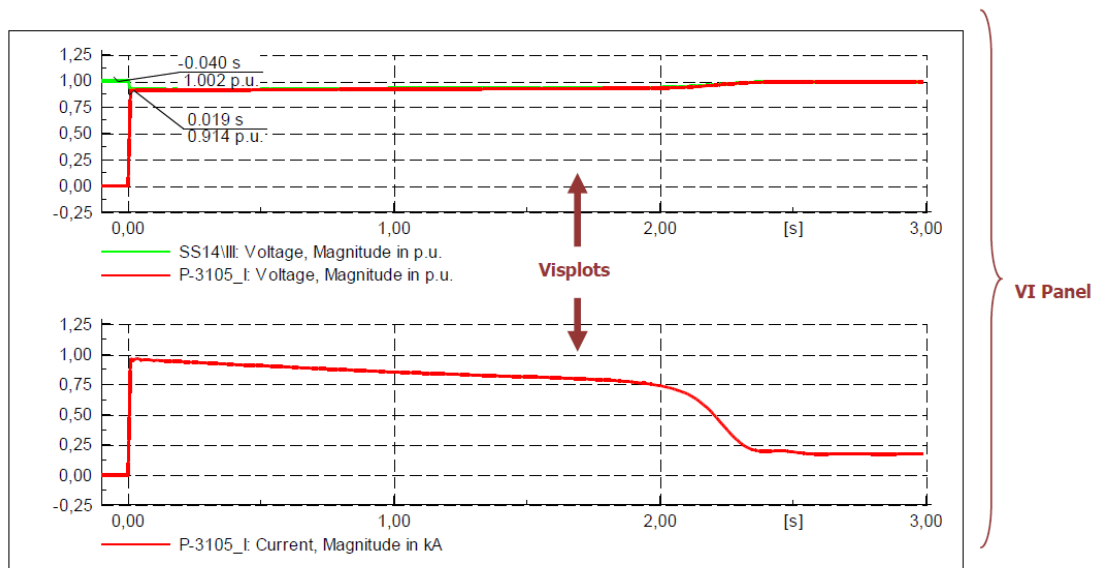


Figure 6.1: Contents of a VI panel

### 6.2 Local Title Blocks

Title blocks are “SetTitm” objects that can be attached to VI graphical objects. The title blocks provide drawing information, revision numbers, dates and logos and is therefore primarily used for printouts of VI graphics. One can manipulate the data in title blocks using Python, which makes automation possible for a large number of similarly themed drawings.

#### 6.2.1 Creating Local Title Blocks Manually

By default, a global title block is applied to each VI panel with common title block information (but with different page numbers). But sometimes you may want to have different title block

information for each VI page. Local title blocks can be created with information specific to the individual VI panel.

You can create local title blocks manually by right-clicking on the default title block and selected “Create local Title” from the context menu (see screenshot below).

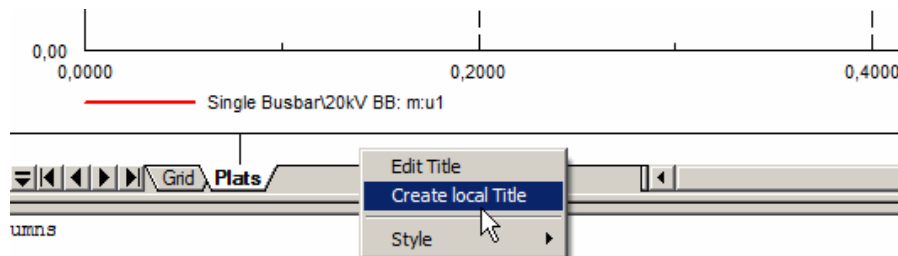


Figure 6.2: Manually creating local title

### 6.2.2 Creating Local Title Blocks in Python

Creating local title blocks manually can be tedious if you have to do it for many VI panels. A faster way is to use a Python script. However there is not straightforward way of creating local title block objects via a Python script. Instead, the simplest way is to copy a template title block into a VI panel object.

First, find the template title block and add it as an external object to the Python command object. The template title block can be any title block in the graphics board. In the example below, the default global title block for the graphics board is used as the template (which can be found in the main graphics board object).

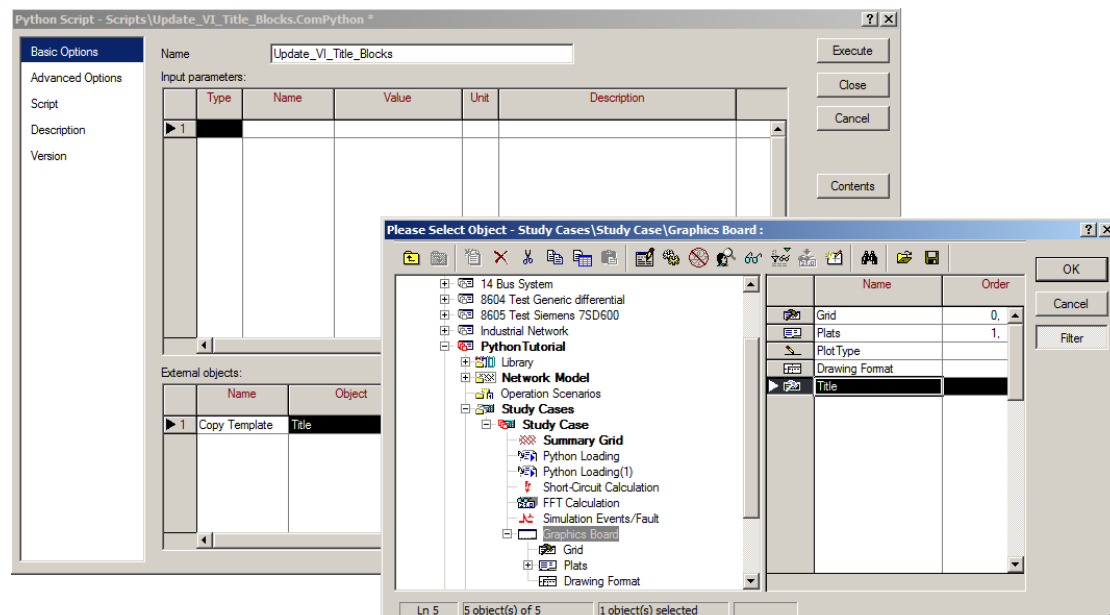


Figure 6.3: External Object inside of the Python script

Next, get the VI panel object (\*.SetVipage) and use the general object method **AddCopy** to copy the title block template into the object. This creates a local title block for the VI panel.

You can now fill in the title block information by getting the title block object (\*.SetTitm) and accessing its variables. For example, if the title block object is oTitle, then you can change the annex, title 1 and title 2 by accessing the variables oTitle:annex, oTitle:sub1z and oTitle:sub2z respectively.

The variable names for each title block field is shown below:

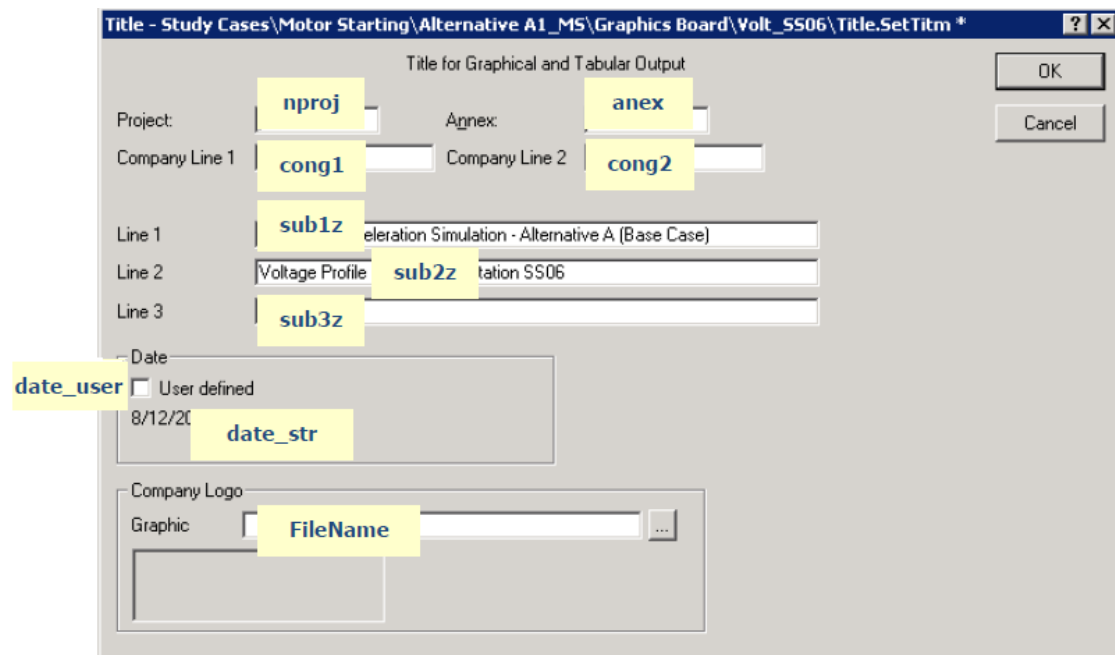


Figure 6.4: External Object inside of the Python script

### 6.2.3 Title Block Python Script Example

This Python script gets the first graphics board and cycles through each VI panel. The script then checks to see if it has a local title block and adds one if it does not. Lastly, the script fills in the title block information. In this example, the object of the class SetTitm (title block) will be created with method CreateObject().

```
import powerfactory
app=powerfactory.GetApplication()
#Get the graphic board
oCase=app.GetActiveStudyCase()
allDesks=oCase.GetContents('*.SetDesktop')
app.PrintPlain(allDesks)
#Get a list of all VI panels
pgs=allDesks[0][0].GetContents('*.SetVipage')
#go through each VI panel
for pg in pgs[0]:
    app.PrintPlain('Name of object:' + pg.loc_name)
    #get the first local title block
    allTitms=pg.GetContents('*.SetTitm')
    app.PrintPlain(allTitms)
    #If no local title block available create one
    if not allTitms[0] :
```

```

app.PrintPlain('No local title block existing.Creating one')
allTitms[0]=pg.CreateObject('SetTitm','name')
#Fill in title block
oTitle=allTitms[0][0]
app.PrintPlain(oTitle)
oTitle.anex = '1'
strVIpage = pg.loc_name
oTitle.sub2z = 'Voltage Profile Station' +strVIpage

```

**IMPORTANT NOTE:** Objects cannot be created into a VI panel object while the graphic is open. Therefore, close the graphics board prior to executing any Python scripts that will manipulate title block objects.

### 6.3 Exporting VI Panels to WMF

Often you will want to send VI panels as a picture file or insert it into a report. Obviously, you could do this manually using the export command (i.e. to a WMF or a BMP):

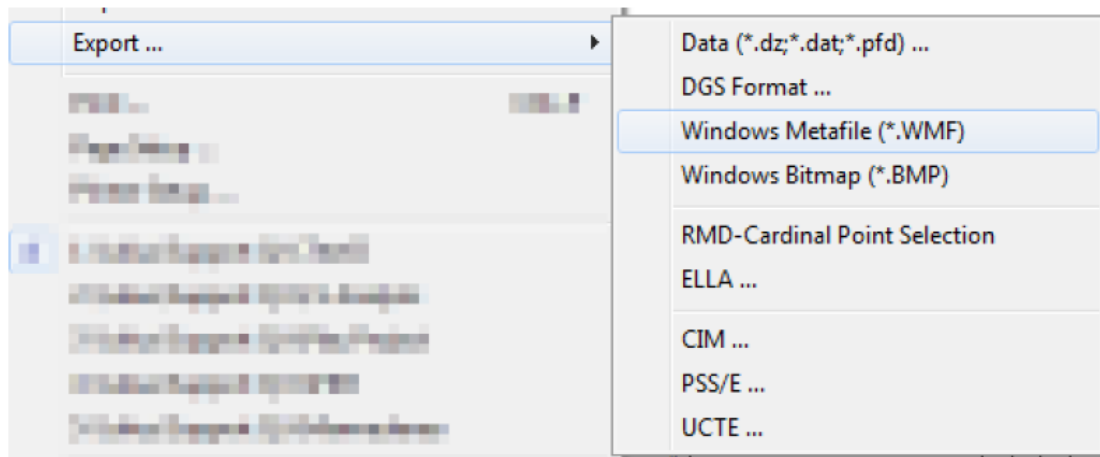


Figure 6.5: Manually exporting VI Panels as \*.wmf

But this can be a tedious task if there are many VI panels to export. A quicker alternative is to use a Python script to bulk export VI panels or any other type of graphic (this was covered briefly in Section 4.2.2). The general process is as follows:

- Locate the relevant graphics board
- Select the VI panels or graphic object that you want to export
- Export the VI panel with the **WriteWMF** function

The following example gets the first graphic board in the active study case and exports all of the VI panels in the graphics board as ViPanel.loc\_name.wmf in the directory File\_name:

```

import powerfactory
app=powerfactory.GetApplication()
#Get the graphic board
oCase=app.GetActiveStudyCase()
allDesks=oCase.GetContents('*.SetDesktop')
app.PrintPlain(allDesks)

```

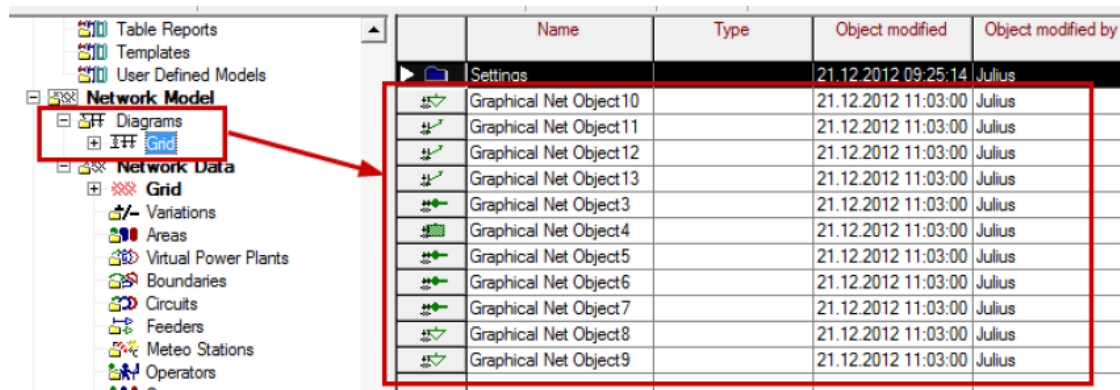
```
#Get a list of all VI panels
pgs=allDesks[0][0].GetContents('*.SetVipage')
#go through each VI panel
for pg in pgs[0]:
    allDesks[0][0].Show(pg)
    File_name=(r'C:\Users\sljiljak\Desktop' + '\\%s' %(pg.loc_name))
    allDesks[0][0].WriteWMF(File_name)
```

## 7 Working with Graphical Objects

### 7.1 Graphical Objects in *PowerFactory*

On single line diagrams, graphical representations of network elements, text objects and other graphics (e.g. rectangles, circles, etc) are stored **Graphical Net Objects** (\*.IntGrf) inside diagram graphics (\*.IntGrfnet), which are in turn located in the Network Model / Diagrams subfolder.

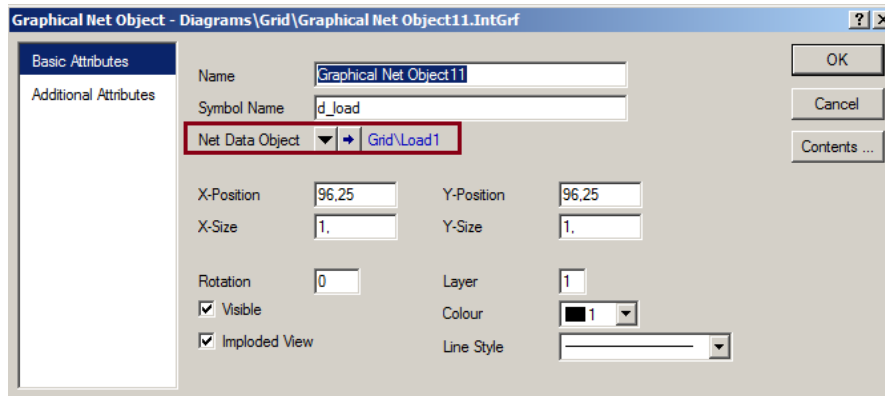
The figure below presents the contents of diagram graphic “Grid”, showing a number of graphical net objects stored inside the diagram:



|  | Name                   | Type | Object modified     | Object modified by |
|--|------------------------|------|---------------------|--------------------|
|  | Settings               |      | 21.12.2012 09:25:14 | Julius             |
|  | Graphical Net Object10 |      | 21.12.2012 11:03:00 | Julius             |
|  | Graphical Net Object11 |      | 21.12.2012 11:03:00 | Julius             |
|  | Graphical Net Object12 |      | 21.12.2012 11:03:00 | Julius             |
|  | Graphical Net Object13 |      | 21.12.2012 11:03:00 | Julius             |
|  | Graphical Net Object3  |      | 21.12.2012 11:03:00 | Julius             |
|  | Graphical Net Object4  |      | 21.12.2012 11:03:00 | Julius             |
|  | Graphical Net Object5  |      | 21.12.2012 11:03:00 | Julius             |
|  | Graphical Net Object6  |      | 21.12.2012 11:03:00 | Julius             |
|  | Graphical Net Object7  |      | 21.12.2012 11:03:00 | Julius             |
|  | Graphical Net Object8  |      | 21.12.2012 11:03:00 | Julius             |
|  | Graphical Net Object9  |      | 21.12.2012 11:03:00 | Julius             |

Figure 7.1: Contents of the IntGrfnet element ,Datamanager

If we were to edit a Graphical Net Object (GNO), we would see the following:



Graphical Net Object - Diagrams\Grid\Graphical Net Object11.IntGrf

Basic Attributes

Additional Attributes

Name: Graphical Net Object11

Symbol Name: d\_load

Net Data Object: Grid\Load1

X-Position: 96,25 Y-Position: 96,25

X-Size: 1 Y-Size: 1

Rotation: 0 Layer: 1

☒ Visible Colour: 1

☒ Imploded View Line Style:

OK Cancel Contents ...

Figure 7.2: IntGrf element “Basic Attributes” tab

The Net Data Object field (variable name “pDataObj”) highlighted in red indicates that this GNO is associated with grid element “Load1”, which simply means that this is a graphical representation of a network element. There can be more than one graphical representation of the same object across many diagrams, though not in a single diagram. When the GNO does not represent a grid element (e.g. it is just a rectangle, circle, etc), the Net Data Object field is blank.

The Symbol Name field (variable name “pSymNam”) defines the graphical symbol for this GNO, in this case a load symbol.



The X-Position and Y-Position fields(variable names “rCenterX” and “rCenterY” respectively) specifies the (x,y) coordinates in the diagram where the symbol should be drawn (centre points).

There is another type of graphical object that defines the connections between two objects (e.g. a load connected to a bus). This object is called a Graphical Connection Object (\*.IntGrf-con) and can typically be found inside a GNO:

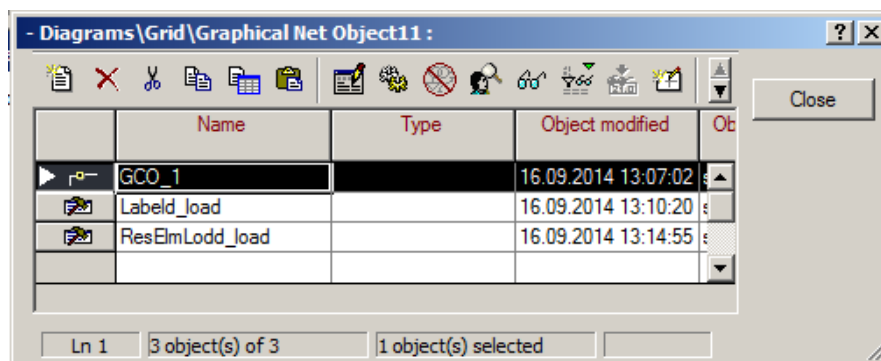


Figure 7.3: IntGrfcon element contents of an GNO

Editing the first Graphical Connection Object (GCO), we get the dialog box on the following page:

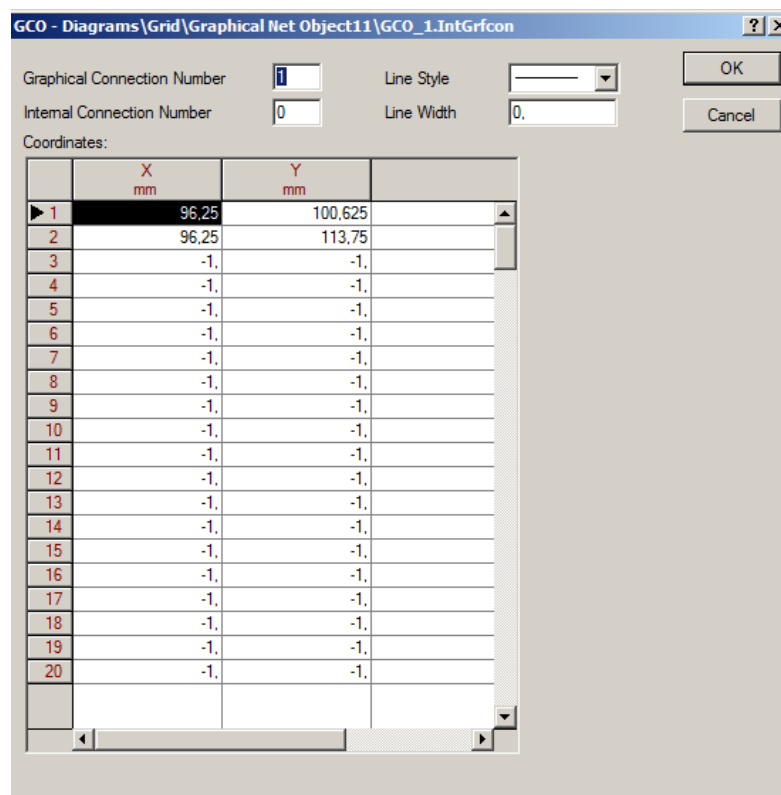


Figure 7.4: Contents of a Graphical Connection Object

The GCO specifies the (x, y) coordinates for straight-line connections between the GNO (e.g. a

load) and its connected element (e.g. a bus). In the example above, the GCO shows a simple vertical connection with endpoint (x, y) coordinates (96.25, 100.625) and (96.25, 113.75).

This brief background was intended to describe the nature of graphical objects and how they are used in *PowerFactory*. By understanding the concepts of GNOs and GCOs, we can start using Python to manipulate graphical objects.

## 7.2 Creating Graphical Objects in Python

As with other Python applications, the easiest way to create a graphical object is by copying a template object. As the configuration of the graphical objects will be largely dependent on the specific application, only a general process for creating graphical objects is presented here.

- Create template graphical net object (GNO), complete with graphical connection object (GCO) stored within. Make sure the template objects are accessible by the Python script either as internally or externally referenced object (see section 3.3)
- Find the diagram object (\*.IntGrfnet) where you want to create the graphical objects
- Find or create the network element with which you want to associate the graphical object
- Copy the template GNO into the selected diagram object (with the AddCopy function)
- Get the newly created GNO and modify the following (as required):
  - Associated network element (GNO:pDataObj)
  - Graphical symbol (GNO:pSymNam)
  - X- and Y- coordinates for object (GNO:rCenterX and GNO:rCenterY)
- Access the associated GCO and modify the X- and Y- coordinates for the connections (GCO:rX and GCO:rY)