PowerFactory

TRAINING MATERIAL

Scripting in PowerFactory with Python

March 2021

Online

POWER SYSTEM SOLUTIONS

MADE IN GERMANY

PowerFactory

Training Material

Scripting in PowerFactory with Python

Publisher:
DIgSILENT GmbH
Heinrich-Hertz-Straße 9
72810 Gomaringen / Germany
Tel.: +49 (0) 7072-9168-0
Fax: +49 (0) 7072-9168-88

Please visit our homepage at:
https://www.digsilent.de

Copyright DIgSILENT GmbH
All rights reserved. No part of this publication may be reproduced or distributed in any form without written permission of the publisher.

February 2021
r7575

Contents

Introduction

The purpose of these exercises is to introduce the basic principles of the Python Programming Language in *PowerFactory*. There are different ways to solve the exercises in the scripts; the sample solution shows only one possible solution.

During the exercises the supervisor will support and help you with your tasks in the exercise. Additionally he can provide answers to general questions regarding the training topic or different problems from your own experience. Please do not hesitate to address the supervisor at any time to any topic!

1 Basic Python Scripting in *PowerFactory*

Purpose: Familiarisation with the general handling of the Python Programming Language in *PowerFactory*, e.g. the creation Python script in *PowerFactory* and the access of *PowerFactory* data within Python.

Contents:
Creating a Python scripts in *PowerFactory*.
Python *PowerFactory* module.
ComPython Object.
Simple Python Script examples.

1.1 Presentation: Basic Python Syntax

Notes:

Agenda

- Basic Python Scripting in *PowerFactory*
- *PowerFactory* Objects access
- Execution of *PowerFactory* Commands
- Navigation through the *PowerFactory* Project
- Report Results with Functions, Subroutines and Modules
- Result Files
- Graphical Representation
- Additional Exercises:
  - AddOn-Module and User Handling
  - Unattended/Engine Mode
  - External File Access
  - Graphical User Interface in Python

Introduction and Basics 2

Introduction

- Python
  - General purpose programming language
- Enhance *PowerFactory* functionality with Python by:
  - Automating tasks
  - Creating user defined calculation commands
  - Integrating *PowerFactory* with other applications

Introduction and Basics 3

Python - Modules and Interpreter

- Basic modules
  - math, tkinter, time, sys…
- Additional modules
  - numpy (Numeric Python; Data structures, matrices, arrays…)
  - matplotlib (Plotting of diagrams)
  - powerfactory
  - …
- Anaconda distribution (Python installation with pre-installed additional modules)
- Interpreter
  - ipython
  - IDLE
- Editor
  - Notepad, Notepad++, PSPad, Spyder

Introduction and Basics 4

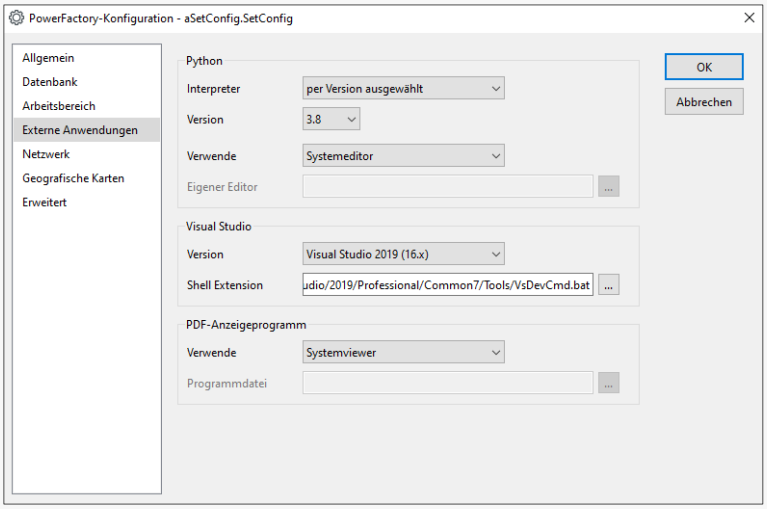
Python Scripting vs DPL

- Self-containment of scripts within *PowerFactory*:
  - ComPython links a .py-file with *PowerFactory* or executes embedded Python code with an external Python application.
  - DPL (ComDpl) contains the code inside. No external application needed.
- Scripting Editor can be used for DPL and embedded Python scripts.
- Variable declaration
  - DPL declares variables at the beginning of the Code
  - Python needs no explicit declaration (initialisation at runtime)
- Open source software: (Python only)
- Global scripts (available in DPL, not available in Python)
- A Python script (ComPython) can not execute other ComPython commands.
- Additional information in the *PowerFactory* User Manual (Chapter Scripting)

Introduction and Basics 5

Preparing Python for use in *PowerFactory*

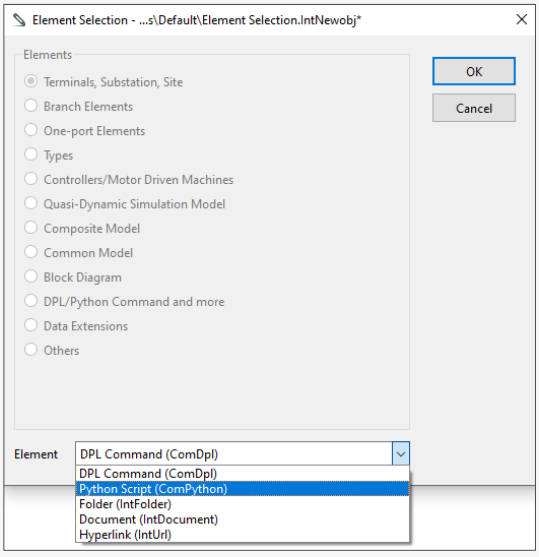
- Installation of a Python Interpreter
  - Py 3.3 to 3.8
  - 32/64 bit *PowerFactory* requires 32/64 bit Python
  - installation of corresponding Python add-on/package
- Tools -> Configuration



Introduction and Basics 6

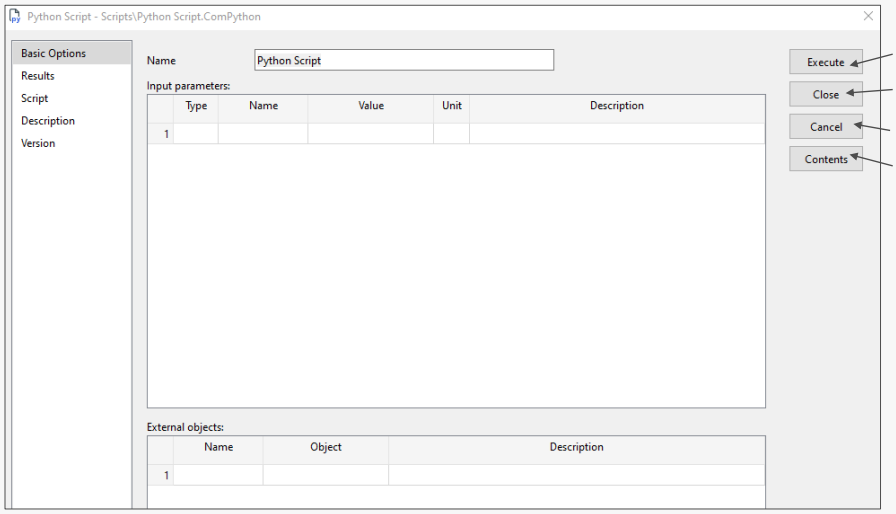
Creating a Python Script Object

- Navigate to intended parent folder
  - Dedicated script folder in Project Library
    - Data Manager → …\Library\Scripts
    - Data → Scripts…
  - Other locations are also possible
    - Study Case
    - …
- Create new script object
  - New Object → Python Script (ComPython)

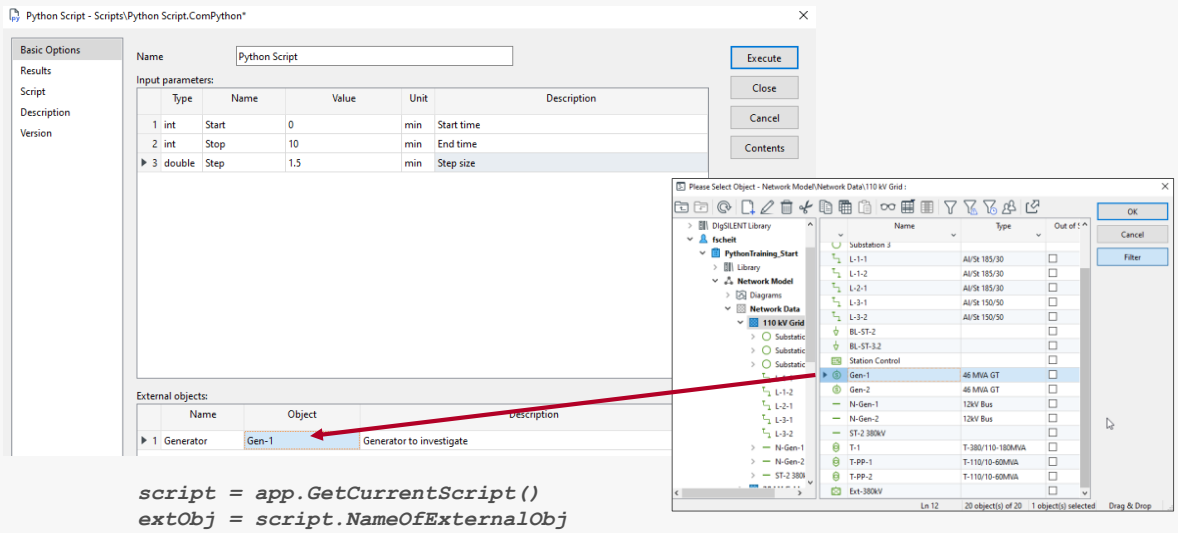


Introduction and Basics 7

Handling Python in *PowerFactory*

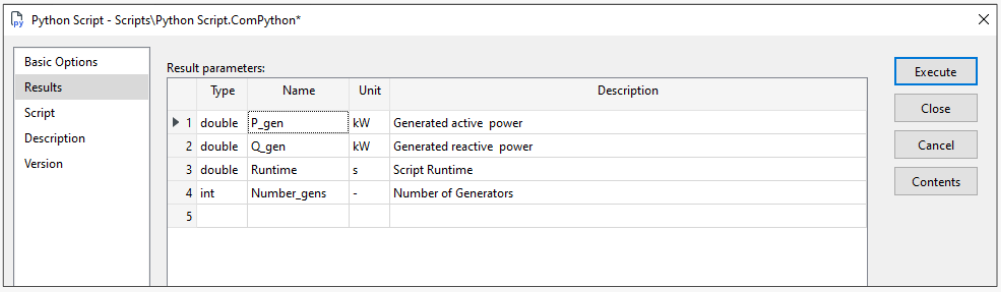


Input Parameters and External Objects

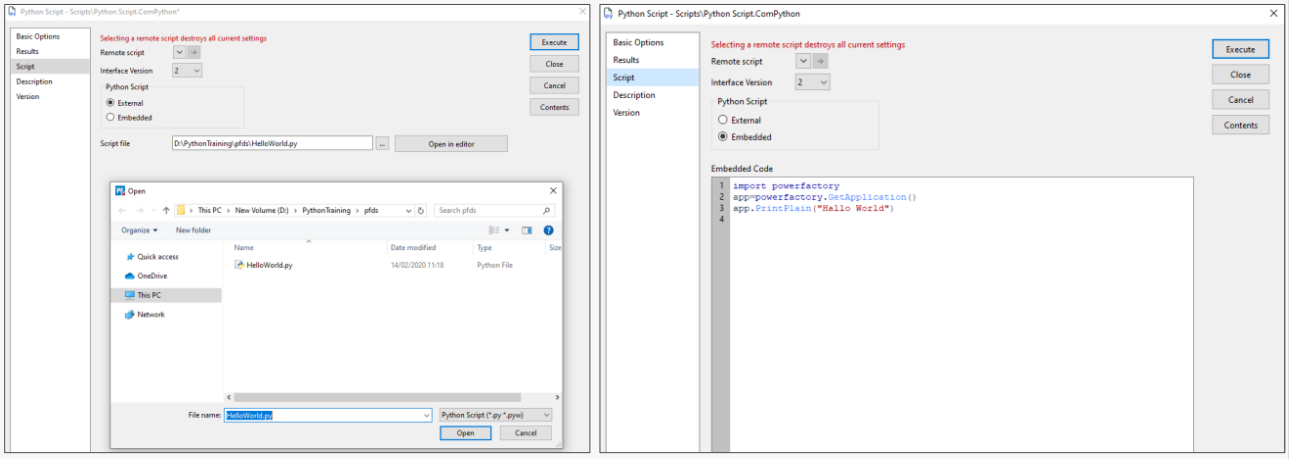



```
script = app.GetCurrentScript()
extObj = script.NameOfExternalObj
inpPar = script.NameOfInpParm
```

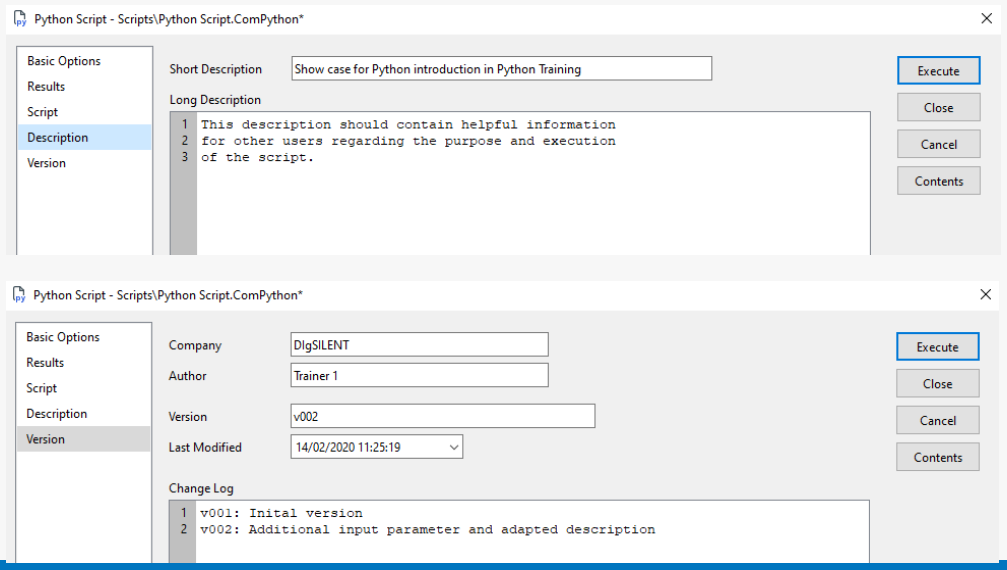
Output Variables



Python Code – External vs Embedded



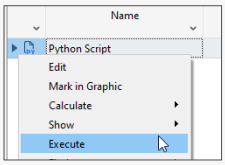
Description and Versions



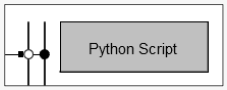
Execution of Python Scripts

- Main Icon Bar
  - List of all available global and local DPL and Python scripts

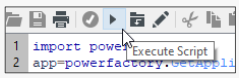
- Execute ComPython directly
  - Press Button Execute in an open ComPython object
- Execution from Data Manager
  - Right click on script object and selection of “Execute” from the context menu



- Execution from Network Diagram
  - Through Command Button (VisButton)
  - Right click on script object and selection of “Execute” from the context menu

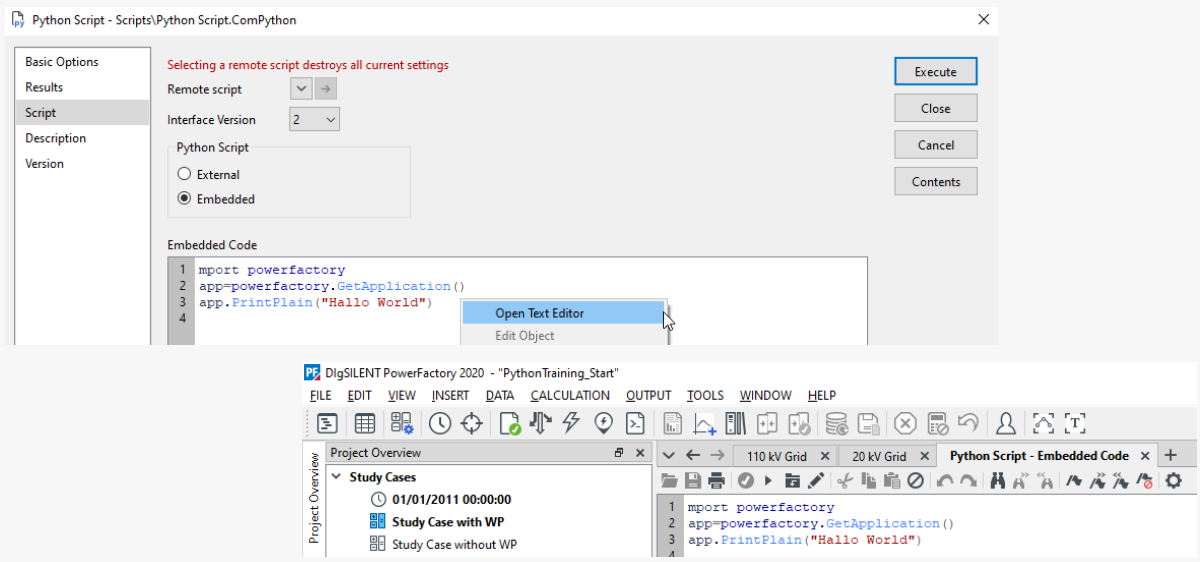


- Execution from Scripting/Text Editor (Only Embedded Scripts)
  - Click the “Execute Script” button in the editor



- Execution Interruption
  - At any time from Main Icon Bar

Scripting/Text Editor for Embedded Scripts



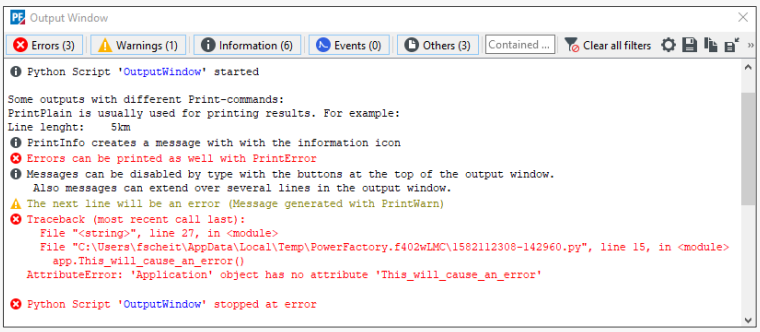
How to begin a Python Script

- *PowerFactory* module import in Python
  - **import** *powerfactory*
- To access the *PowerFactory* environment
  - app = powerfactory.GetApplication()
- "app" - Application object that provides access to global *PowerFactory* functionality
- Example:

```
import powerfactory as pf
app = pf.GetApplication()
app.ClearOutputWindow()
ldf = app.GetFromStudyCase("ComLdf")
ldf.Execute()
```

Python Script - Outputs

- Output Window:
  - Messages
  - Errors
  - Warnings
  - Information
  - Protocols
- Functions:
  - Application.PrintPlain(str)
  - Application.PrintError(str)
  - Application.PrintWarn(str)
  - Application.PrintInfo(str)
  - Application.ClearOutputWindow()



Introduction and Basics 16

1.2 Creating Python Scripts in *PowerFactory*

There is a difference between a Python Script-file and a Python Command Object in *PowerFactory*. The Python Script-file is created outside of *PowerFactory*, contains Python code and is independent from *PowerFactory*. It is easy to recognise by the *.py* extension. The Python Command Object is an internal *PowerFactory* object (*ComPython*) that is linked to an external Python Script-file.

Python Script-file (*.py*)

There are many ways to write a **.py* file to be linked with a *ComPython* object in *PowerFactory*. It can be done, for example, using **.txt* files to write the code and then changing the extension **.txt* → **.py*. Important is that the script file is saved using the UTF-8 character encoding format. Another way is using the "IDLE(Python(GUI))" program and selecting *File* → *New Window...*

After being written, the Python Script-file *.py* can be now linked with a *ComPython* object.

Python command Object (*ComPython*)

The Python command object can be created using the menu *Data* → *Scripts...*. In the new dialog, click on the *NewObject* button and select *Python Script (*ComPython*)* as Element (see figure 1.1).

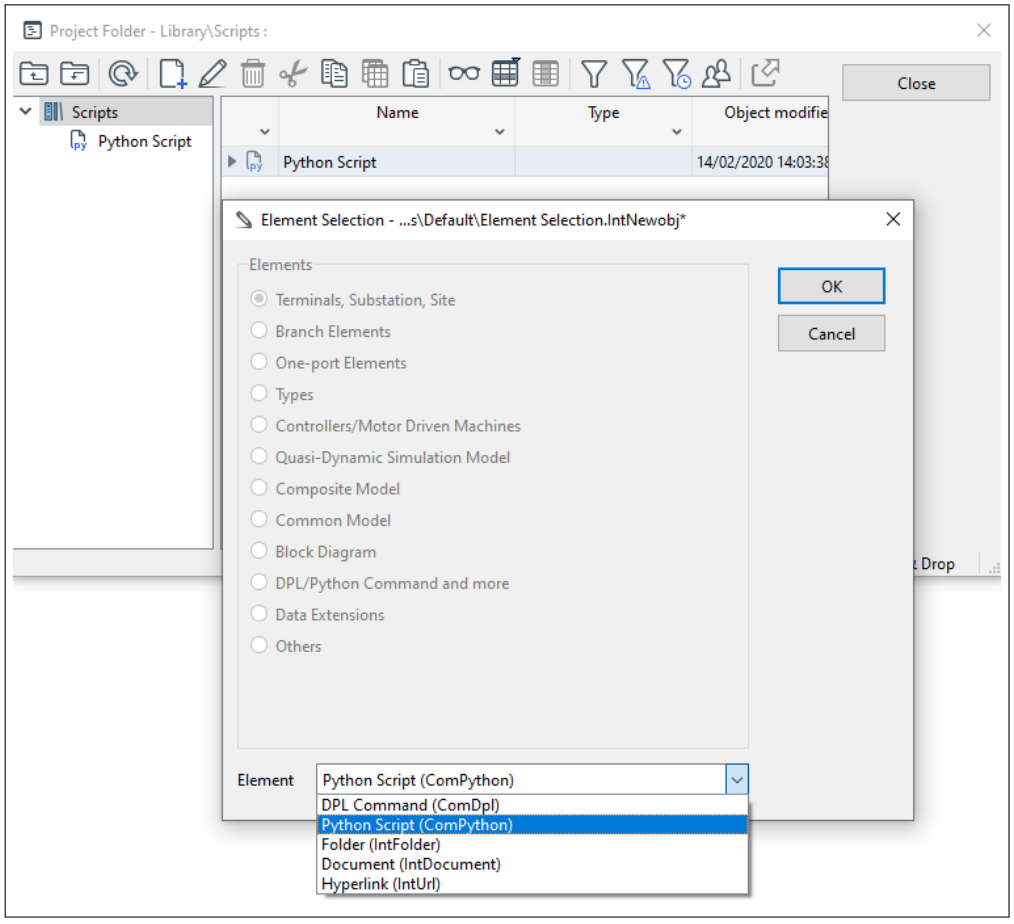


Figure 1.1: Creating a Python Script in *PowerFactory*

Each *ComPython* object dialog contains five pages:

- **Basic Options:** Input Parameters and External Objects.
- **Results:** Result variables.
- **Script:** link to the Python Script .py and *PowerFactory* Python objects see figure 1.2. Also, a remote script link can be defined. From *PowerFactory* 2019 on embedded Python scripts are supported.
- **Description:** description of the script.
- **Version:** additional data e.g. Company, Author, Version.

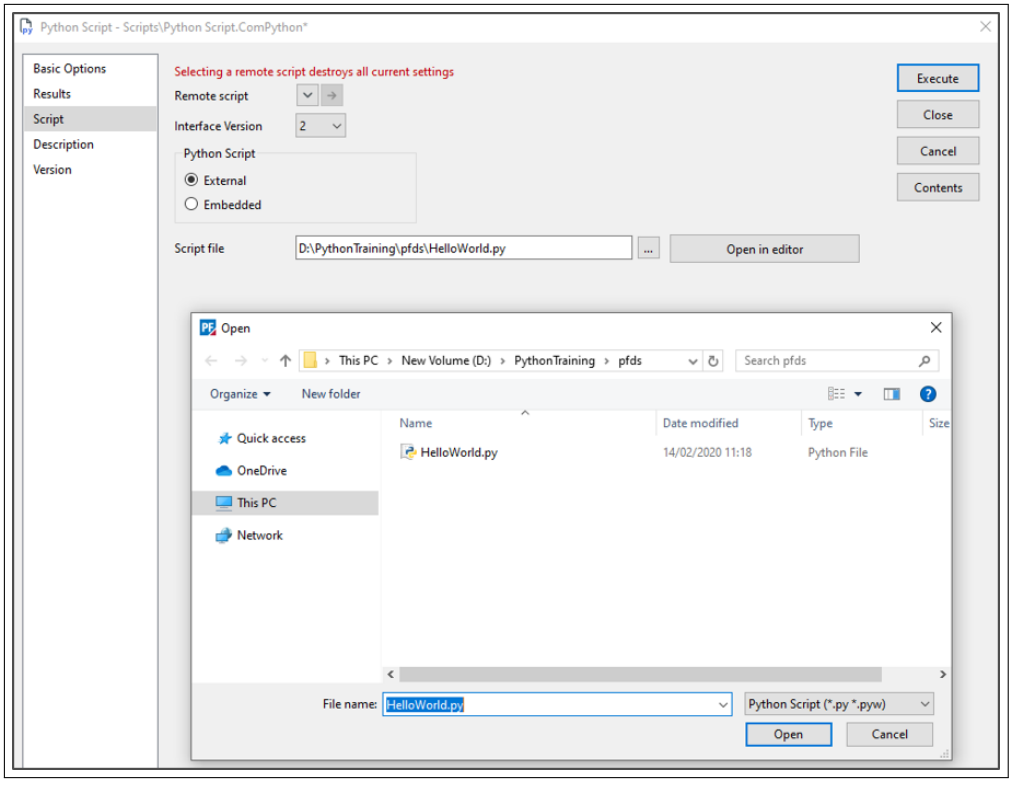


Figure 1.2: Selecting of the *.py-files for a Python Script in *PowerFactory*

Additional information regarding the Python Script options is available in the User Manual.

1.3 Python *PowerFactory* Module

The Python *PowerFactory* Module enables a Python script to have access to data already existing in *PowerFactory*:

- All objects
- All attributes (element data, type data, results)
- All commands (Load Flow Calculation, Short-Circuit Calculation,. . .)
- Most special built-in functions (DPL functions)

To be able to use data available in *PowerFactory*, this module has to be imported in the Python environment by using a standard Python import statement:

```
import powerfactory
```

To access the *PowerFactory* environment the method GetApplication has to be called:

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

"app" is in the above an "application object", containing global *PowerFactory* functionality.

1.4 Getting Started

Write a python script that is printing "Hello PowerFactory" into the output window.

- Create a ComPython object in the Script folder in the project library and name it "HelloPF".
- Write the code to print the string "Hello PowerFactory" into the output window of *PowerFactory*. The code should contain three steps:
  - Import the *PowerFactory* module
  - Get the application object and call the variable "app"
  - Use the PrintPlain function to print text into the output window.
- Safe the code as "HelloPowerFactory.py" and select the file as external Python script in the ComPython object
- Execute the script (HelloPF.ComPython) in *PowerFactory*.
- If the script is successfully executed go back to the "HelloPF.ComPython" and change the Python Script to embedded on the Script page.
- Copy the code from the .py file into the Embedded Code field and execute the script again. You should see exactly the same output.
- Extend the code with the PrintInfo and PrintWarn functions to print "Hello PowerFactory" as an information message and a warning and notice the difference when executing the script.

Code for printing a plain text into the output window:

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

Hint: Make sure you have the same Python version selected in *PowerFactory* as you have installed on your computer. This can be done under *Tools* → *Configuration*→ *External Applications*→ *Python Version*

---

2 *PowerFactory* Objects access with Python

Purpose: Learn to access objects of different classes and their attributes.

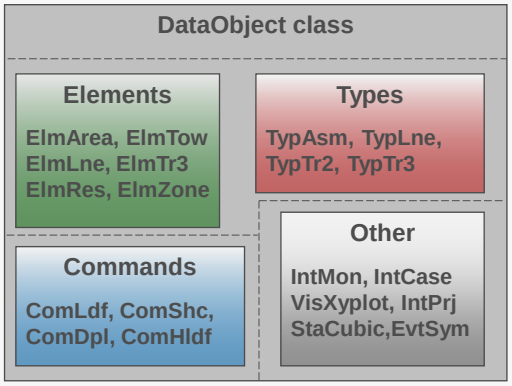
Contents: Creating Python scripts in *PowerFactory*.
Basics of Object classes in *PowerFactory*.
Object access.
Read/Write of Objects attribute.

2.1 Presentation: *PowerFactory* Objects

Notes:

Object classes

- Objects are grouped according to the kind of element that they represent. These groups are known as “Classes” within the *PowerFactory* environment
- Everything in *PowerFactory* is an object. Every object belongs to an object class.
- Objects are stored according to a hierarchical arrangement in the database tree.



Object Access 2

Object classes in *PowerFactory*

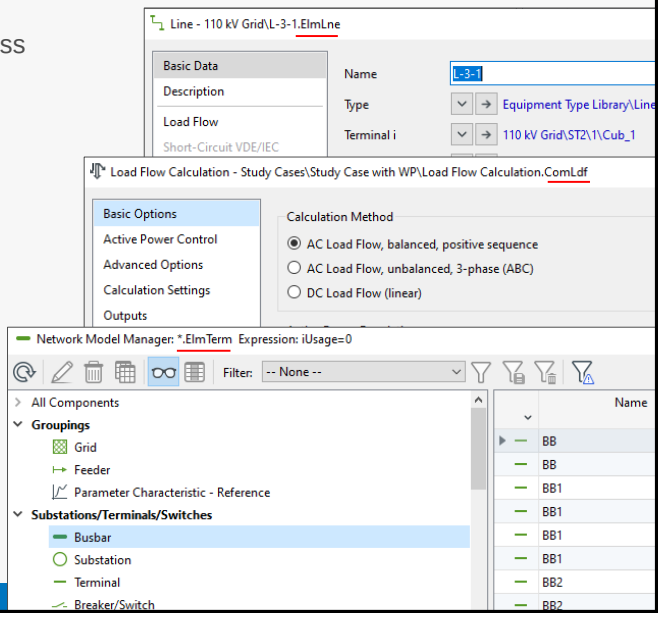
- Each *PowerFactory* object belongs to a Class and each Class has a specific set of parameters that defines the objects it represents.
- ObjectName.**ClassName**

  2-Winding Transformer.ElmTr2

  LoadFlow Calculation.ComLdf

  *.ElmTerm

  terminal*.ElmTerm
- ObjectName:
  - No empty names
  - No invalid characters *?=“,\~|
  - Unique (in the same data-tree level)



Object Access

PowerFactory object access

- Accessing external objects:
  - variable = script.extObjectName
  - easy to select
  - faster and shorter code

External objects:

	Name	Object	Description
1	Generator	Gen-1	Generator to investigate

- With Method:
  - **GetCalcRelevantObjects()**
    list Application.GetCalcRelevantObjects(
    [str Name],[int includeOutOfService],
    [int elementsOnly])

```
terminals = app.GetCalcRelevantObjects('*.ElmTerm')
terminal = app.GetCalcRelevantObjects('Bus1.ElmTerm')[0]
```

  - **GetFromStudyCase()**
    DataObject Application.GetFromStudyCase(str className)

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

  - **GetProjectFolder()**
    DataObject Application.GetProjectFolder(string)

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

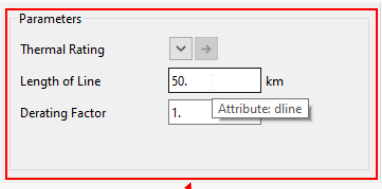
  - ...

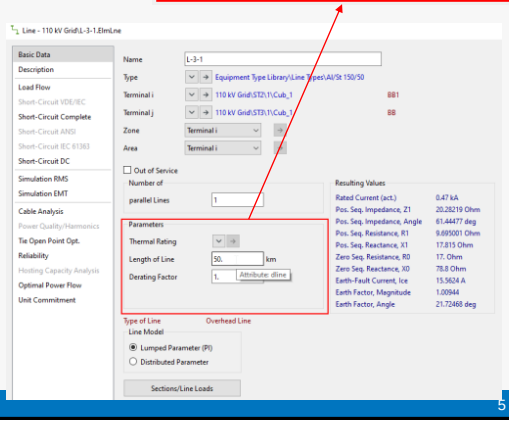
Object Attributes

- Syntax:

  Object.attrname

  Object.GetAttribute("attrname")



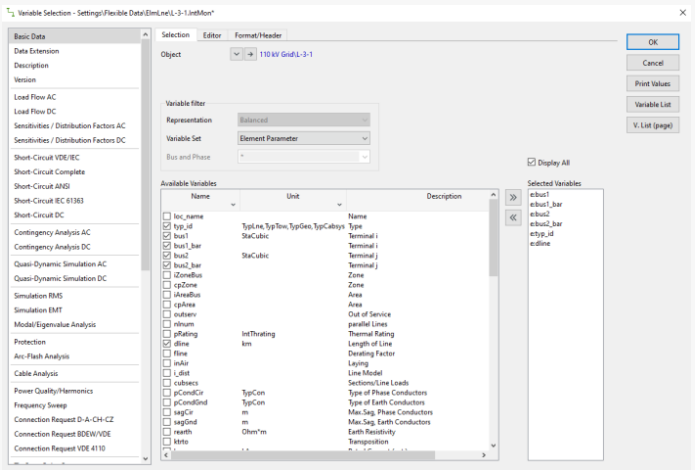


- Examples:

```
line.dline = 5.4
length = line.dline
if length > 10:
    code
Ldf.iopt_at = 1
value = Line.GetAttribute('m:u:bus1')
```

Object Attributes – Variable Selection

- List of available attributes of an object class:
  - Element Parameter (*e:AttrName*)
  - Reference Parameter (*r:AttrName*)
  - Type Parameter (*t:AttrName*)
  - Currents, Voltages and Powers / Measurement Parameter (*m:AttrName*)
  - Calculation Parameter (*c:AttrName*)
  - Signals (*s:AttrName*)
  - Data Extensions (*p:AttrName*)
  - AddOn (*x:AttrName*)



Object Access 6

2.2 Object Access

In this first exercise you will learn how to access elements in *PowerFactory* and their parameters, how to change them and present the values in the output window.

2.2.1 Total Line Length

Create a script that calls all active lines in the test system, sums the line lengths and reports the results in the output window.

- Import the project 'Python_Start' into your user account and activate it.
- Create a Python Script in the project folder *Library* → *Scripts* and name it 'Line Length'
- Create a Python file (*.py*) in the external editor and name it 'LineLength'
- Import *PowerFactory* module and call *GetApplication* command
- Assign all active Lines to a list parameter
- Use a loop to go through all elements of the list (all Lines)
- Call the length parameter of the Line element (*ElmLne*) and calculate the sum
- Report how many lines are in the system and the total length of lines

```
Python Script 'LineLenght' started
Total length of the 22 lines is: 200.80 km
Python Script 'LineLenght' successfully executed
```

Figure 2.1: Output of the script 'Line Length'

keywords:
GetCalcRelevantObjects(), ElmLne, PrintPlain()

Hint: It is possible to access the element parameter directly. The "Flexible data" page helps to find the parameter name and class (m:monitor, e:element, t:typ . . .).
Example: 'dline' is the parameter for the line length of a line.

The following steps show how to find the corresponding parameter for line length:

- Press the icon *Network Model Manager* and select the lines.
- Select the page 'Flexible Data' and use the icon *Define Flexible Data* to get the 'Variable Selection' menu.
- Search the parameter 'Length of Line' (*Basic Data* → *Element Parameter*).
- Use the found parameter in the Python script to sum the line lengths.
- Get the unit of the line length.
- Add a formatted output of the line length including the unit at the end of the script.
- Execute the script and compare the output with the sum determined using the 'Flexible Data' page.

- Add the number of lines to the output string.

Note: List is a container that holds a number of elements, in a given order. For example output of the method GetCalcRelevantObjects('*.ClassName') is a list containing all objects of the given class that are relevant for calculation. To access one element of the list you may use indexing:

```
firstElement = List[0]
secondElement = List[1]
xElement = List[x-1]
lastElement = List[-1]
secondLastElement = List[-2]
```

Other option of accessing the elements is to loop over the list. **for in** statement makes it easy to access each element of the list, one by one:

```
for element in List:
    app.PrintPlain(element.loc_name)
```

If both are needed, i.e. the element and its index, then instead of using:

```
index = 0
for element in List:
    app.PrintPlain('Nr %i %s' %(index,element.loc_name))
    index += 1
```

one may use the built in function **enumerate()**:

```
for index,element in enumerate(List):
    app.PrintPlain('Nr %i %s' %(index,element.loc_name))
```

Python offers a lot of built in functions (such as **len(),min(),max(),...**) and methods (such as **sort(),..**) that make working with lists easier. For more information on this please refer to the official Python documentation.

2.2.2 Expanded LineLength Script

- Expand your Python file script and name it 'LineLengthStep1', so that it also calculates the average line length.
- Print the resulting average length to the output window.

```
Python Script 'LineLenght' started
Total length of the line is: 200.80 km
Average length of the lines is 9.13 km
Active grid includes 22 lines
Python Script 'LineLenght' successfully executed
```

Figure 2.2: Output of the expanded script 'Line Length'

- Edit the Python file script and name it 'LineLengthStep2', so that it calculates separate values for the total length of overhead lines and cables.

Hint: The specification whether a line object is a cable or an overhead line can be found in the type.

- Adapt the output of the script accordingly.

```
Python Script 'LineLenght' started
Total length of :
- Overhead lines : 170.00 km
- Cables : 30.80 km
Python Script 'LineLenght' successfully executed
```

Figure 2.3: Output of the finished script 'Line Length'

2.2.3 Optional Task: Total Load

Create a script that calls all loads, calculate some values regarding the power and report the results to the output window.

- Get all the loads and loop through the received list.
- Calculate the total active power of all loads as well as the average active power.
- Count how many loads have a power factor bigger than 0.9. Therefore, use an if-condition to filter the loads.
- Find the load with the highest active power and print the value to the output window.

PowerFactory files

File Name	Description
PythonTraining_Start.pfd	Network used for this exercise
LineLength.py	Python script-file, solution of exercise 2.2.1
LineLengthStep1.py	Python script-file, solution of exercise 2.2.2
LineLengthStep2.py	Python script-file, solution of exercise 2.2.2

3 Execution of the *PowerFactory* Commands with Python

Purpose: Execute calculation commands with Python.

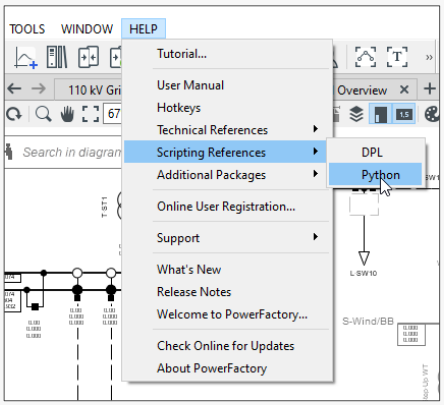
Contents: Basics of Python methods in *PowerFactory*.
Scripted execution of commands.
Modify command attributes.

3.1 Presentation: Objects Methods and Execute Function

Notes:

PowerFactory - Scripting Reference

- All available Python methods for the communication with *PowerFactory* are described in the Python Scripting-Reference.
  - *PowerFactory* Module
  - Application methods
  - Output Window
  - Object methods
- General information can be found in the *PowerFactory* user manual.



Object Methods and Execute Functions 2

Classification of Python Methods for *PowerFactory*

- Application methods

  val **app**.Method(arg1, arg2, ..)

  app – the application method (retrieved with powerfactory.GetApplication())
  val – returned variable/object (if any)
  arg1, arg2 – specific arguments of the function (if any)
  Example: retrieving the currently active project with the method **GetActiveProject**()

- Object methods:
  - General methods

    val **obj**.Method(arg1, arg2, ..)

    obj – any object within PowerFactory
    Example: Creating a *PowerFactory* object using the method **CreateObject()**
  - Specific methods

    val **obj**.Method(arg1, arg2, ..)

    obj – an object of specific PowerFactory class (e.g. ElmLne, TypTr2, etc)
    Example: Executing the load flow command with the method **Execute()**

Object Methods and Execute Functions 3

Application Class Methods

- Examples of application class methods
  - **ActivateProject**(str project)
  - **CreateProject**(str projectName, str gridName,[object parent])
  - **GetActiveProject**()
  - **GetActiveScenario**()
  - **GetProjectFolder**('str type')
  - **GetFromStudyCase**(str className)
  - **GetCalcRelevantObjects**([str className],[int includeOutOfService])

- Usage:

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

Object Methods and Execute Functions 4

Object Class Methods

- Syntax:

  *iret = Object.**Method** (arg1,arg2,....,argn)*

- Some Methods are reserved for Objects of special Class

  Execute() → only for *.Com* objects

  Activate(), Deactivate() → for some Int* and Elm* objects (IntCase, IntPrj, ElmNet)

- Some Methods are General (they apply for any class of objects)

  Delete(), GetAttribute()

- Examples:

```
ldf.Execute()      ->     class specific method (ComLdf)
Results.Write()    ->     class specific method (ElmRes)
Folder.Delete()    ->     general method
```

Object Methods and Execute Functions 5

Object Class Methods

- Methods without arguments:
  - Object.**Delete**(), Object.**Activate**()
    → Deletes Object, Activates Object
- Methods with arguments:
  - NewObject = TargetObject.**CreateObject**(string1, string 2, string 3)
    → Creates and returns a “NewObject” of class “string1” and the name “string2”+”string3” within the *PowerFactory* path of the “TargetObject”.
- Method calls cannot be used in expressions:
  - not allowed :

```
if ldf.Execute() == 0:
    Code
```

  - instead :

```
ierr = ldf.Execute()
if not ierr:
    Code
else:
    app.PrintError("Error in Ldf")
    exit(1)
```

Object Methods and Execute Functions 6

3.2 Command Execution

In this chapter we will learn how to access and execute *PowerFactory* Commands. Some examples of *PowerFactory* commands are: Load Flow Calculation (*ComLdf*), Short-circuit Calculation (*ComShc*), DPL Command (*ComDpl*), Result Export (*ComRes*) and Contingency Analysis (*ComSimoutage*).

3.2.1 Load Flow Calculation

Through the script created in this exercise you will learn how to retrieve the Load Flow Calculation command out of the active Study Case, how to execute it and how to access the calculation values.

- Import the project “PythonTraining_Start.pfd” and activate it.
- Create a new Python script in the project folder *Library* → *Scripts* and name it ’Loadflow’.
- Create a Python file (*.py*) in the external editor and name it ’Loadflow.py’
- Use the command *GetFromStudyCase()* to get the Load Flow Calculation command of the active Study Case.
- Execute this load flow command.
- Access and report the active power consumption of the load elements installed in the network.
- Test your script.

```
Python Script 'Loadflow' started
Load BL-ST-2     P= 35.00 MW
Load BL-ST-3.2   P= 50.00 MW
Load L-ST1       P= 9.00 MW
Load L-ST3       P= 11.00 MW
Load L-SW1       P= 3.80 MW
Load L-SW10      P= 6.00 MW
Load L-SW11      P= 7.00 MW
Load L-SW12      P= 8.00 MW
Load L-SW13      P= 10.00 MW
Load L-SW14      P= 3.00 MW
Load L-SW2       P= 3.00 MW
Load L-SW4       P= 4.50 MW
Load L-SW5       P= 4.20 MW
Load L-SW6       P= 2.00 MW
Load L-SW7       P= 1.00 MW
Python Script 'Loadflow' successfully executed
```

Figure 3.1: Output of the script ’Loadflow’

keywords:
GetFromStudyCase(), Execute(), ComLdf, ElmLod, GetCalcRelevantObjects(), GetAttribute()

Hint: With “\n” or “\t” it is possible to insert a tabulator or a new line in a python string. This may help to improve the clarity of the plotted text in the output window.

3.2.2 Extended Load Flow Calculation

Now the script will be extended. In the first step the Load Flow Calculation will be executed without the option 'Consider Voltage Dependency of Loads'. The script compares the sum of the calculated active power of all loads with the sum of the set active power of the loads. In the next step the Load Flow Calculation will be executed again with the option 'Consider Voltage Dependency of Loads'. The quotient (calculated active power/set active power) should be calculated again. The output of both calculations should be displayed in the output window at the end of the script.

- Open the Load Flow Calculation command and find out the parameter name for the option 'Consider Voltage Dependency of Loads'.

Hint: The parameter name will be displayed, if the mouse cursor rests over the input box (this is also valid for check boxes)

- Use the flexible data page for the loads to get the parameter name for the active power load flow result and the set active power.
- The solution for the quotient without voltage dependency is 1 and with voltage dependency is 0.93.

keywords:
GetFromStudyCase(), GetCalcRelevantObjects(), Execute(), GetAttribute(), PrintPlain()

3.2.3 Optional Task: Voltage Comparison

Write a script to compare the voltage at a 20 kV busbar with and without voltage dependency of loads. For this find a 20 kV busbar. Get the set target voltage and compare it to the resulting voltage after the execution of a load flow calculation with and without the consideration of the voltage dependency of loads.

PowerFactory files

File Name	Description
PythonTraining_Start.pfd	Network used for this exercise
Loadflow.py	Python script-file, solution of exercise 3.2.1
LoadflowExtended.py	Python script-file, solution of exercise 3.2.2

4 Navigation through the *PowerFactory* Project

Purpose: Learn how to access different objects placed inside the *PowerFactory* project.

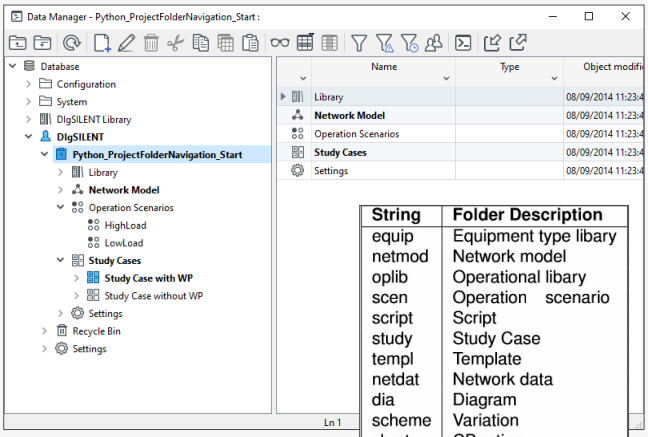
Contents: Accessing different study cases, activate them and execute calculations.
Introducing EchoOff and EchoOn functions.
Using operation scenarios.
Using external objects.
Reporting calculation results.

4.1 Presentation: Python Folder Navigation

Notes:

Navigate the Project Contents

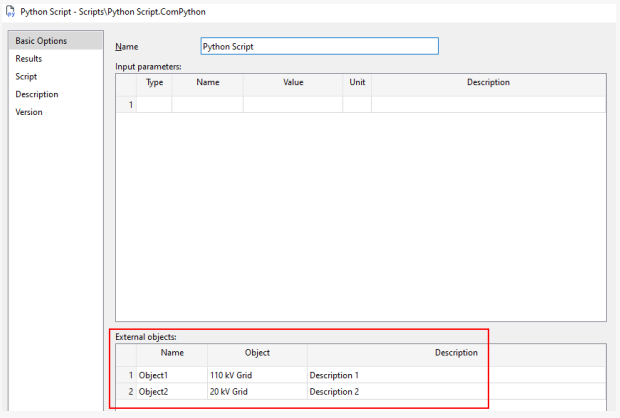
- All data of the Project are sorted under corresponding folders/subfolders and can be directly accessed (GetProjectFolder() or via external objects) or indirectly accessed (Get Contents(), GetParent(), GetChildren()).
- Via GetProjectFolder() only some folders of a project can be accessed.
- Using GetContents()/GetChildren() or GetParent(), the user can navigate through the content of the project.



String	Folder Description
equip	Equipment type libary
netmod	Network model
oplib	Operational libary
scen	Operation 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

Accessing External Objects

- The External Objects option offers quick access to any object that is contained in the *PowerFactory* Database.
- To access an external object the user first has to access the script currently used (GetCurrentScript()).
- The access to input parameters has to be made in the same way.




```
script = app.GetCurrentScript()
extObj = script.NameOfExternalObj
inpPar = script.NameOfInpParam
```

4.2 Study Case Sweep

Often it is useful to get the results of several study cases one after another. This way the user gets an opportunity to compare the results of different cases.

- Import the project file “Python_ProjectFolderNavigation_Start.pfd” file.
- Create a new Python script with the name “StudyCaseSweep”:
  - Get all the Study Cases that are available in the project.
  - Run the load flow command for each study case located in the project folder Study Cases.
  - Report the loading of all lines which are loaded more than 80 %.

The following piece of code shows just one way of accessing the Study Case folder and looping through it to get the Study Cases that are available in the project:

```
studyCaseFolder = app.GetProjectFolder("study")
for studyCase in studyCaseFolder.GetContents():
    studyCase.Activate()
    ldf = app.GetFromStudyCase("ComLdf")
    ldf.Execute()
```

Note: The EchoOff() and EchoOn() Functions can be used improve the readability of the output window.
The EchoOff() Function turns off the displaying of information messages from executed *PowerFactory* commands in the output window. EchoOn() enables it again. This way, the user can select if messages should be shown or not. This also improves the performance of the script.

```
app = powerfactory.GetApplication()
app.EchoOff()
.
.
Some Code to execute
.
.
app.EchoOn()
```

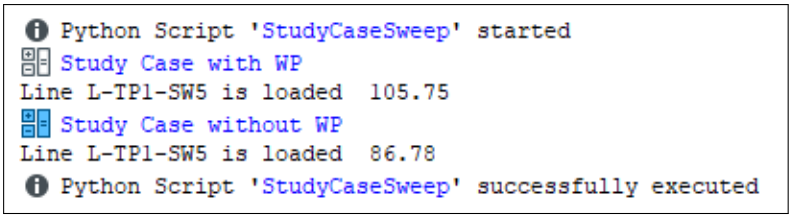


Figure 4.1: Output of the script ’Study Case Sweep’ with EchoOff() and EchoOn() function

4.3 Operation Scenarios Sweep

In the project two operation scenarios are available: LowLoad and HighLoad. The task is to expand the Study Case sweep script that each of these two operation scenarios is accessed, activated and investigated for both study cases. This means we have to execute four load flow calculations.

- Expand the sweep script to gain access to the operation scenario.
- Instead of using GetProjectFolder("scen") use the external object approach by defining the folder “Operation Scenarios” as a external variable in the ComPython-Script.
- Plot the results for each study case and operation scenario combination in the output window.

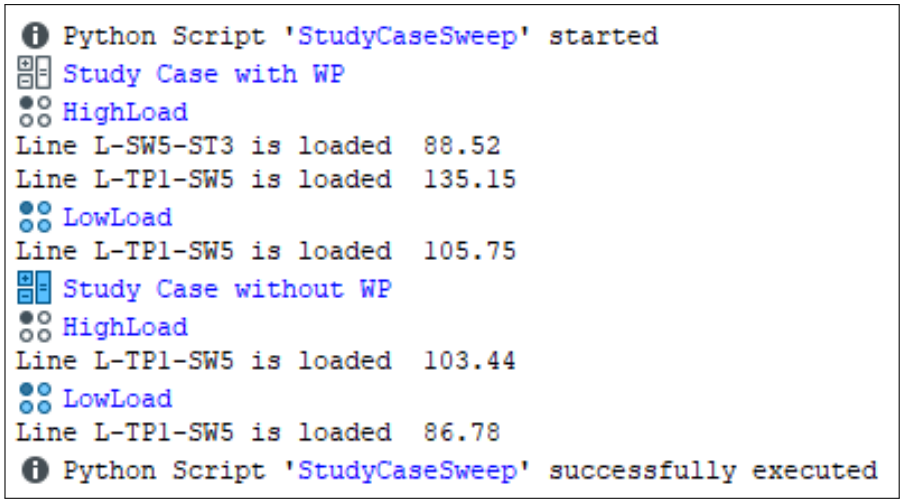


Figure 4.2: Output of the script ’Study Case Sweep’ with different Operation Scenarios

PowerFactory files

File Name	Description
Python_ProjectFolderNavigation_Start.pfd	Network used for this exercise
StudyCaseSweep.py	Python script-file, solution of exercise 4.2
StudyCaseSweepOperationScenario.py	Python script-file, solution of exercise 4.3
Python_ProjectFolderNavigation_Finish_.pfd	Solution project file

5 Report Results, Subroutines, Functions and Modules

Purpose: Learn how to output customised results in the output window, use subroutines and functions, keep the script readable by avoiding long code.

Contents: Report Results in *PowerFactory* output window.
Create functions.
Import modules.
Reloading modules.
Study case sweep.

5.1 Presentation: Python Modules and Subroutines

Notes:

Functions

- A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

  - In Python functions are defined with “def function():”
  - Functions take in arguments and optional arguments and can have a return value

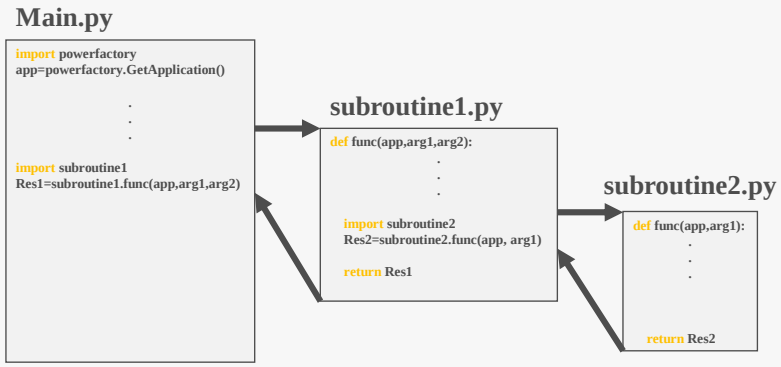
  - def function(arg1, arg2,…, optArg1=a, optArg2=b):
    …
    code
    …
    return x, y

```
In [1]: def example_function(list_of_values, factor = 1):
   ...:     #function calculates the sum of a list with values
   ...:     #and returns the sum multipied with a factor
   ...:     #factor is an optinal argument (Default = 1)
   ...:     sum_of_values = sum(list_of_values)
   ...:     return_value = sum_of_values * factor
   ...:     return return_value
   ...:

In [2]: example_function([1,2,3],1)
Out[2]: 6

In [3]: example_function([1,2,3])
Out[3]: 6

In [4]: example_function([1,2,3],2)
Out[4]: 12
```


Subroutines

- Using subscripts allows to tackle bite size pieces of a problem individually.
- Once each piece is working correctly the whole solution can be built by putting the pieces together.

- Only one Python Script can run at one given moment.
- A ComPython cannot contain/ execute other ComPython objects.



Importing subroutines

- **import** subroutineName

  Res= subroutineName**.function(**app,arg1,arg2,…**)**

- **import sys**

  sys.path.append(r"C:\Users\...:")

  "This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available." (https://docs.python.org/2/library/sys.html)

- **import imp**

  imp.reload(subroutineName)

  "This module provides an interface to the mechanisms used to implement the import statement." (https://docs.python.org/3.1/library/imp.html)

Subroutines 4

5.2 Customised Reports

In the following exercises you will create some customised reports with Python and print them in the output window. You will learn how to keep your script readable by using functions, avoiding long code and import subscripts.

5.2.1 Project Data Reporting

The first step of the task is to create a script that calls the active project, prints its name, executes a load flow and prints a report in the output window.

- Import the project file “PythonTraining_Start.pfd”.
- Create a new script in the project folder *Library* → *Scripts* and name it ’Load Flow Report’.
- Create a Python file (*.py*) in the external editor and name it ’LDF_Report.py’
- Import the *PowerFactory* module and call the GetApplication() method. The Script should carry out the following operations:
  - Get the active project and study case. Print an error if they don’t exist.
  - Execute the Load Flow Calculation command.
  - Print an error if the execution failed.
- Print the current Project and Study Case in the output window.
- Test your script with the set-up of the already imported project.

keywords:
GetActiveProject(), GetFromStudyCase(), GetActiveStudyCase(), PrintError, PrintPlain, Execute()

Hint: If an error occurs during the execution of a script it can be handy to stop the script at this point. The Python command **exit()** stops the execution of the script.

Note: Python offers an easy way to check if a variable is None. This is very useful and can help in debugging process. For example:

```
project = app.GetActiveProject()
if project is None:
    app.PrintError("There is no active Project")
else:
    app.PrintPlain("Project: " + project.loc_name)
```

It is also possible to check if a variable is not None:

```
project = app.GetActiveProject()
if project is not None:
    app.PrintPlain("Project: " + project.loc_name)
else:
    app.PrintError("There is no Active Project")
```

Possible partial code of the LDF_Report.py file. How to check if no error is returned and stopping the script.

```
ierr = ldf.Execute()
if ierr == 0:
    app.PrintInfo("Load Flow command returns no error")
else:
    app.PrintError("Load Flow command returns an error")
    exit()
```

keywords:
HasResults, GetCalcRelevantObjects, .path.append, GetAttribute, PrintPlain

5.2.2 Loading Violations

In many cases it is important to have an overview of the highest loaded elements in the network in order to identify the weakest points in the network. So the task in this exercise is to plot a report in the output window that includes all overloaded lines. The purpose is to use functions and subscripts for this task.

The goal of the second step is to add a new function called **report()** to the LDF_Report.py file:

- First the functionality of getting the loading violations should be implemented without a function:
  - Get a list of lines relevant for calculation.
  - Loop over the contents of the list.
  - Check, if there are results available for the lines, using the command 'HasResults()'.
  - Print each line loaded above 65% to the output window.
  - The output should at least contain the name and the loading of the line.
- Do a similar check for the loading violations for the 2-winding transformers in the grid.
- The script contains now two nearly identical parts of code for lines and transformers. The only difference is the Class of the element. Create a new function called report(app, ClassName) and adapt the code that it can be reused and works the argument ClassName for the selection of the elements.
- Optional: With ljust you can align your outputs by making it independent from the name length. This only works for strings and not for DataObjects.
- Optional: Additional arguments like the variable to investigate, the critical threshold and whether elements smaller or larger then the threshold shall be reported might be very handy and can be implemented as well.
  - Add a copy of your function report and rename it to report_advanced()
  - Add optional arguments like the threshold, the variable name and checking for larger or smaller values to the function and extend the function by these functionalities.

```
Python Script 'LoadFlowReport' started
Project: PythonTraining_Start(1)
Study Case: Study Case with WP
Element 'Ext-380kV' is local reference in separated area of '— ST-2 380kV'
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.
Load Flow command returns no error

Name            c:loading
L-SW13-ST3      70.51
L-SW5-ST3       78.76
L-SW6-TP1       69.21
L-TP1-SW5       120.06
L-TP2-TP1       66.24


Name            c:loading
T-PP-1          69.16
T-PP-2          69.16
T-ST1           80.21
T-ST3           71.16

Python Script 'LoadFlowReport' successfully executed
```

Figure 5.1: Output of the script 'Load Flow Report'

Possible partial code of the function def report()

```
def report(app):
  lines = app.GetCalcRelevantObjects("*.ElmLne")
  for line in lines:
    ...

#call the function report(app)
report(app)
```

Note: Python methods **ljust(x), rjust(x)** return string of the length x justified on the left/right side by adding blank spaces.

For example:

```
    app.PrintPlain(Elm.loc_name.ljust(10) + str(Elm.dline) +
      Elm.GetAttributeUnit('dline')
```

This could be very useful to get a better optical presentation of results for example:

```
Python Script 'Load Flow Report' started
Project: PythonTraining_Start
Study Case: Study Case with WP
Load Flow command returns no error
Name     Loading
T-PP-1     69.16
T-PP-2     69.16          without .ljust()
T-ST1     80.21
T-ST3     71.16
Python Script 'Load Flow Report' successfully executed
Python Script 'Load Flow Report' started
Project: PythonTraining_Start
Study Case: Study Case with WP
Load Flow command returns no error
Name           Loading
T-PP-1         69.16
T-PP-2         69.16          with .ljust()
T-ST1          80.21
T-ST3          71.16
Python Script 'Load Flow Report' successfully executed
```

The third step is to move the function to a subscript to gain more structure in the main script. This is especially important for larger scripts.

- Create a new Python script file in the External Editor and name it 'ViolationCheck', this script will be later called by the LDF_Report.py file.
- The created functions report() / report_advanced() shall be moved to the subscript ViolationCheck.py and be called from the LDF_Report.py script.

Possible partial code to link the sub-file ViolationCheck.py to the LDF_Report.py file.

```
import powerfactory
import imp
import sys
sys.path.append(r"E:\PYT_Report\Step3")

import ViolationCheck
imp.reload(ViolationCheck)

LoadingViolation.report(app)
```

5.2.3 Optional Task: Sorting

Until now the violations are printed unsorted. The list of violations will now be sorted from the highest to the lowest loadings and therefore show the worst violation at the begin.

- Modify the 'ViolationCheck.py' script or respectively the functions in the script so that the worst violation are displayed first.
- Optional: Functions can also call other functions. So the sorting routine can be implemented in a sub function.

keywords:
GetCalcRelevantObjects, iHasResults, GetAttribute, append, sort, reverse, PrintPlain

Note: Methods **append(), sort() and reverse()** are to be applied on the variables of the type list. Method **append** appends a new element in the list for example:

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53)
[MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more infor
mation.
>>> list1=[] #defining an empty list
>>> list1
[]
>>> list1.append(1)
>>> list1
[1]
>>> list1.append([3,2,5,4,7,6])
>>> list1
[1, [3, 2, 5, 4, 7, 6]]
>>> list1.remove([3,2,5,4,7,6])
>>> list1
[1]
>>> list2=[[1,'one']] # defining 2 dimensional list
>>> list2
[[1, 'one']]
>>> list2.append([3,'three'])
>>> list2
[[1, 'one'], [3, 'three']]
>>>
```

Method **sort** sorts a one dimensional list from the smallest value to the greatest and the two dimensional list by the same principle just according to the first dimension.

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53)
[MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more infor
mation.
>>> list1=[1,3,2,5,4,7,6]
>>> list1
[1, 3, 2, 5, 4, 7, 6]
>>> list1.sort()
>>> list1
[1, 2, 3, 4, 5, 6, 7]
>>> list2=[[1,"one"],[3,"three"],[2,"two"]]
>>> list2
[[1, 'one'], [3, 'three'], [2, 'two']]
>>> list2.sort()
>>> list2
[[1, 'one'], [2, 'two'], [3, 'three']]
```

Advanced Note: There are many ways how you can sort lists in Python. Most handy way of sorting a list of Objects is by defining a key as sorting criteria. Key provides a function that returns information how the list should be sorted.
For example:

```
1 def funcKey(elm):
2     return elm.dline
3
4 lines.sort(key = funcKey)
```

With the lambda-function, we can write the same code in a faster way:

```
1 funcKey = lambda elm:elm.dline
2
3 lines.sort(key  = funcKey)
```

Or even faster by assigning the lambda function directly to the key argument. The lambda helps us to define an anonymous function (function that is not bound to a name). This is a very powerful concept that is well integrated in Python. For more information on this please refer to official python literature.

```
listOfObjects.sort(key = lambda x:x.count)
```

Example for sorting a list of lines by the length in km from shortest to longest line:

```
listLineObjects.sort(key  = lambda x:x.dline)
```

With reverse the sorting can be reversed and the list is sorted by the ordering criteria from largest to smallest:

```
listLineObjects.sort(key = lambda x:x.dline, reverse = True)
```

5.2.4 Optional Task: Voltage Violations

Now we will report the busbars whose voltages are outside the established limits.

- Value of the high voltage limit should be 1.05 p.u and low voltage limit 0.95 p.u.
- The output should be sorted from worst to least violation.
- Plot the voltage violations for different operation scenarios and study cases

Hint: For this task you can modify your script/functions to make it flexible regarding the element class, the variable and the limits instead of writing new functions.

```
Python Script 'Load Flow Report' started
Project: PythonTraining_Start

Voltage Violations :

Study case: Study Case with WP
Operation scenario: HighLoad

Name            m:u
BB1             0.93
BB2             0.93
BB              0.91


Name            m:u
N-Gen-2         1.06
N-Gen-1         1.06

Operation scenario: LowLoad

Name            m:u
BB1             0.93
BB2             0.93
BB              0.91


Name            m:u
N-Gen-2         1.06
N-Gen-1         1.06
```

Figure 5.2: Output of the script including sorted loading and voltage violations

Advanced Note: In previous Exercise 5.2.3 the following code was used to filter all lines with the loading bigger than 65 %.

```
SetElm = [] #creating a empty list

for line in lines:
    iHasResults = line.HasResults()
    if iHasResults == 1:
        value = line.GetAttribute("c:loading")
        #reading the value of c:loading parameter
        if value > 65: #filtering for all values bigger than 65\%
            SetElm.append([value,line.loc_name])
            #appending values into the empty list SetElm
```

This could also be done in a much more casual way by using just the standard for-loop and if-condition and the so called list comprehension:

```
lines_loaded = [x for x in lines if x.GetAttribute("c:loading")>65]
```

PowerFactory files

File Name	Description
PythonTraining_Start.pfd.pfd	Network used for this exercise
LDF_Report_Step1.py	Python script-file, solution of exercise 5.2.1
LDF_Report_Step2.py	Python script-file, solution of exercise 5.2.2
ViolationCheck_Step3.py	Python script-file, solution of exercise 5.2.2
LDF_Report_Step3.py	Python script-file, solution of exercise 5.2.2
ViolationCheck_Step4.py	Python script-file, solution of exercise 5.2.3
LDF_Report_Step4.py	Python script-file, solution of exercise 5.2.3
ViolationCheck_Step5.py	Python script-file, solution of exercise 5.2.4
LDF_Report_Step5.py	Python script-file, solution of exercise 5.2.4

6 Results File

Purpose: Get familiar with the Result File element (*ElmRes*) and learn how to write to and read from it.

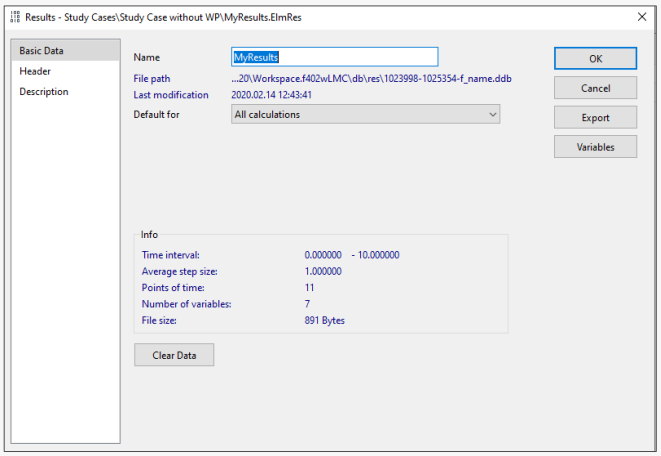
Contents: Result file Handling.
Writing/reading a Result File.

6.1 Presentation: Results File

Notes:

Result Element

- The Results object (ElmRes) is used to store tables with the results obtained after the execution of a command in *PowerFactory*.
- The obtained results can later be used to generate plots or in scripts.



How to write to a Result File (ElmRes)

1. Get access to ElmRes and eventually delete old contents:

```
result = app.GetFromStudyCase(„ResultObject.ElmRes")
result.Clear()                          # Removes previous stored result data.
```

2. Initialise ElmRes:

```
result.AddVariable(Obj1,"m:variable")   # Adds one variable to the list of monitored variables for
                                        # the Result object.
result.AddVariable(Obj2,"m:variable")
result.InitialiseWriting()              # Initialises the result file
```

3. Write values to ElmRes and at the end close it:

```
result.Write()          # Writes the current results to the result object.
result.FinishWriting()  # Closes the result file
```

How to read from a Result File (ElmRes)

1. Get access to ElmRes:

```
result = app.GetFromStudyCase(„ResultObject.ElmRes"),…
```

2. Load the data from ElmRes:

```
result.Load()                    # Loads the data of a result file in memory.
```

3. Work with ElmRes data:

```
NumVar = result.GetNumberOfColumns()    # Retrieves the number of stored variables.
NumVal = result.GetNumberOfRows()       # Retrieves the number of stored data points per variable.
ColIndex = result.FindColumn(Obj,'m:variable,)    # Identifies a specific result variable in the data.
```

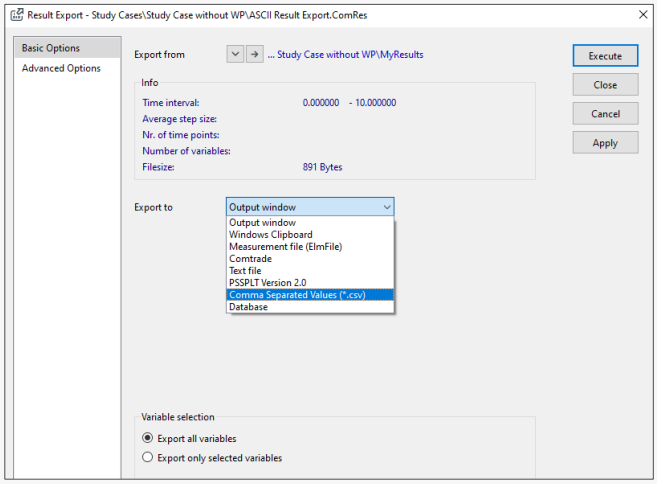
4. Reading specific value from the ElmRes:

```
for row in range(NumVal):
  value = result.GetValue(row,ColIndex)[1]    # Gets the value ([int error, float d]) of variable at "ColIndex" at data point
                                              # "row".
```

Result Elements 4

Exporting data from a Results object

- An ASCII Result Export object (ComRes) can be used:



Result Elements 5

6.2 Result File

In this exercise calculation results will be written in a result file. Afterwards the result file will be read and the results will be presented in the output window.

6.2.1 Short Circuit Sweep

The script in this exercise sweeps the short circuit location along a line. The development of the script starts with a short circuit calculation at 50% of a line.

- Import and activate the project 'PythonTraining_Start.pfd'.
- Activate the study case 'Study Case with WP'.
- Get the short circuit command of the active study case.
- Open the short circuit command to find out the parameter names for the options 'Method' and 'Fault Type'.

Remember, the parameter name will be displayed, if the mouse cursor rests over the input box (this is also valid for check boxes, radio buttons or even assignments/references to other objects).

- Configure the short circuit command manually so that it executes the short circuit *Fault Location* → *At*: User Selection.
- Create a new Python script called 'Short Circuit Sweep' in the project folder *Library* → *Scripts* of the project 'Python_Start'.
- Create a Python file (*.py*) in the external editor and name it 'SHC_Sweep.py'
- Set following parameters in the Python script:
  - *Fault Type*: 3-phase short circuit.
  - *Method* : according to IEC60909.
  - *Fault Location* → *User Selection*: Line object: L-3-1
  - *Short-Circuit at Branch/Line* → *Fault Distance*→ *Relative: 50%*
- Execute this short circuit command.
- Test your script.

keywords:
ClearOutputWindow, GetFromStudyCase, GetCalcRelevantObjects, Execute, PrintPlain

Following code presents how to fill the ComShc command.

```
#assigen a object of the class ComShc to SHC
SHC = app.GetFromStudyCase("ComShc")
#select three phase short circuit
SHC.iopt_shc = "3psc"
#select the first IEC60909 (1-second method, 0-first-one )
SHC.iopt_mde = 1
line = app.GetCalcRelevantObjects("L-3-1.ElmLne")[0]
#choosing the line where the short circuit should occur
SHC.shcobj = line
#position of the short circuit
SHC.ppro = 50
#execution of the short circuit command
SHC.Execute()
```

Now expand your script, so that the short circuit location sweeps along the line by steps of 10 %.

- Results:
  - Currents and impedances at both ends of the line
  - Short circuit current
- The script should report the results as shown below in figure 6.1.

```
---------------------------------------
Short circuit sweep at line L-3-1
---------------------------------------
Position: 0%% I1=11.41kA I2=0.21kA Z1=0.00Ohm Z2=20.28Ohm Ishc=11.61kA
Position: 10%% I1=8.28kA I2=0.81kA Z1=2.03Ohm Z2=18.25Ohm Ishc=9.10kA
Position: 20%% I1=6.43kA I2=1.23kA Z1=4.06Ohm Z2=16.23Ohm Ishc=7.66kA
Position: 30%% I1=5.20kA I2=1.57kA Z1=6.08Ohm Z2=14.20Ohm Ishc=6.78kA
Position: 40%% I1=4.33kA I2=1.89kA Z1=8.11Ohm Z2=12.17Ohm Ishc=6.22kA
Position: 50%% I1=3.68kA I2=2.21kA Z1=10.14Ohm Z2=10.14Ohm Ishc=5.89kA
Position: 60%% I1=3.16kA I2=2.55kA Z1=12.17Ohm Z2=8.11Ohm Ishc=5.71kA
Position: 70%% I1=2.74kA I2=2.94kA Z1=14.20Ohm Z2=6.08Ohm Ishc=5.68kA
Position: 80%% I1=2.37kA I2=3.41kA Z1=16.23Ohm Z2=4.06Ohm Ishc=5.77kA
Position: 90%% I1=2.04kA I2=3.98kA Z1=18.25Ohm Z2=2.03Ohm Ishc=6.01kA
Position: 100%% I1=1.72kA I2=4.71kA Z1=20.28Ohm Z2=0.00Ohm Ishc=6.43kA
---------------------------------------
```

Figure 6.1: Output of the script 'Short Circuit Sweep'

6.2.2 Writing the Result File

The next step is to write the results into a result file.

- Create a result file object (*ElmRes*) inside the active Study Case and name it "MyResults".
- Create a copy of the Python file 'SHC_Sweep.py' and name it 'WriteRes.py'
- Add the variable for the current short circuit location to the result file.
- Add the result parameters to the list of monitored parameters for the result object.
- Write the result parameters in the result file.
- Execute the Python script and check, if there are results stored in the result file.

You could check, if results are stored in the result file by opening the file and checking the file size or exporting the results to the output window.

keywords:
GetFromStudyCase, ClearOutputWindow, GetCalcRelevantObjects, Clear, Init, AddVariable, Execute, PrintPlain, Write, Flush

Part of the code for the WriteRes.py file

```
1 #creating object of the class ElmRes
2 ElmRes = app.GetFromStudyCase("MyResults.ElmRes")
3 ElmRes.Clear()  #clearing of all old date if there are some
```

```
4
5
6  #adding variables and initialising the result file
7  ElmRes.AddVariable(SC,"ppro")
8  ElmRes.AddVariable(line,"m:Ikss:bus1")
9  ElmRes.AddVariable(line,"m:Ikss:bus2")
10 ElmRes.AddVariable(line,"m:Z:bus1")
11 ElmRes.AddVariable(line,"m:Z:bus2")
12 ElmRes.AddVariable(line,"m:Ikss:busshc")
13 ElmRes.InitialiseWriting()
14
15 SHC.Execute() #executing some command
16
17 #writing the values that are initialised after calculation is being completed
18 ElmRes.Write()
19
20 ElmRes.FinishWriting() #close the result file
```

6.2.3 Reading the Results File

The next Python script reads the values from the result file created in the script 'Short Circuit Sweep' (MyResults) and writes the values in the output window.

- Create a new Python script called 'Read Result File'
- Create a Python file (*.py*) in the external editor and name it 'ReadRes.py'
- Use the result file 'MyResults' from the Study Case.
- Load the data from the result file in the memory.
- Print the number of the columns and rows in the output window.
- Print the values for the short circuit current in the output window.
- Test your script.

part of the code of the ReadRes.py file

```
1  #reloading the values from the result file
2  ElmRes.Load()
3  #getting the number of variables and values
4  NumVar = ElmRes.GetNumberOfColumns()
5  NumVal = ElmRes.GetNumberOfRows()
6
7  #get the index of some variable saved inside of result file
8  ColIndex = ElmRes.FindColumn(line,'m:Ikss:busshc')
9
10 # getting all values from column ColIndex from all rows on the
11 for row in range(NumVal):
12     value = ElmRes.GetValue(row, ColIndex)[1]
```

6.2.4 Optional Task: Working with dictionaries

Dictionaries are similar to lists. But instead of using an index to call a list element in a dictionary the values are called by a key. Keys have to be strings but the values can be of any type like string, list, float...

```
>>> Dict = {"key1" : "value 1", "key2": 2 }
```

Dictionary operations:

```
>>> Dict_transl = {"line" : "Leitung", "load":"Last", "load flow":"Lastfluss"}
>>> #Get value of the key "line"
>>> Dict_transl["line"]
'Leitung'
>>> #Get value of the key "load flow"
>>> Dict_transl["load flow"]
'Lastfluss'
>>> #Add a key to a dictionary
>>> Dict_transl.update({"calculation": "Brechnung"})
>>> Dict_transl
{'line':'Leitung','load':'Last','load flow':'Lastfluss',
  'calculation':'Brechnung'}
>>> #Delete something from the dictionary
>>> del Dict_transl["load flow"]
>>> Dict_transl.keys()
['line', 'load','calculation']
>>> #Changing a value
>>> Dict_transl["line"] = "Freileitung"
>>> Dict_transl["line"]
'Freileitung'
```

A dictionary can be used to work with the data from the result file in the Python script. Instead of reading the results for just one variable we will read the data for all variables from the result file.

- Define a dictionary.
- Loop trough the result variables and get the variable name (GetVariable()).
- For each variable loop through the points in time and append them to a list.
- Update your dictionary with the variable and the results.
- Now you can print some or all of the results to the output window or do some further post processing.

part of the code of the ReadRes.py with dictionary

```
D_data = {}
for var_idx in range(NumVar):
  variable_name = ElmRes.GetVariable(var_idx)
  D_data.update({variable_name:[]})
  for row in range(NumVal):
    D_data[variable_name].append(ElmRes.GetValue(row, var_idx)[1])
```

6.2.5 Optional Task: Exporting Result Files

Export all data out of *ElmRes* in output window and later on in *.csv file by using an ASCII Result Export.ComRes object.

MyResults Index	L-3-1	L-3-1	L-3-1	L-3-1	L-3-1	Short-Circuit Calculation Relative: in %
0.0000	11.3574	0.1859	0.0000	20.2822	11.5427	0.0000
1.0000	8.2532	0.7905	2.0282	18.2540	9.0437	10.0000
2.0000	6.4047	1.2087	4.0564	16.2258	7.6132	20.0000
3.0000	5.1861	1.5501	6.0847	14.1975	6.7359	30.0000
4.0000	4.3202	1.8645	8.1129	12.1693	6.1842	40.0000
5.0000	3.6684	2.1807	10.1411	10.1411	5.8483	50.0000
6.0000	3.1538	2.5201	12.1693	8.1129	5.6728	60.0000
7.0000	2.7296	2.9039	14.1975	6.0847	5.6320	70.0000
8.0000	2.3645	3.3573	16.2258	4.0564	5.7198	80.0000
9.0000	2.0351	3.9157	18.2540	2.0282	5.9479	90.0000
10.0000	1.7206	4.6331	20.2822	0.0000	6.3493	100.0000

11 points in time of 7 variables exported.

Figure 6.2: Output of the script 'Read Result File'

PowerFactory files

File Name	Description
PythonTraining_Start.pfd	Network used for this exercise
SHC_Sweep.py	Python script-file, solution of exercise 6.2.1
SHC_Sweep_Expand.py	Python script-file, solution of exercise 6.2.1
WriteRes.py	Python script-file, solution of exercise 6.2.2
ReadRes.py	Python script-file, solution of exercise 6.2.3
PYT_05_ElmRes_Finished.pfd	Solution of exercise 6.2

7 Graphical Representation

Purpose: Get familiar with graphical representations in *PowerFactory* and learn how to create a plot.

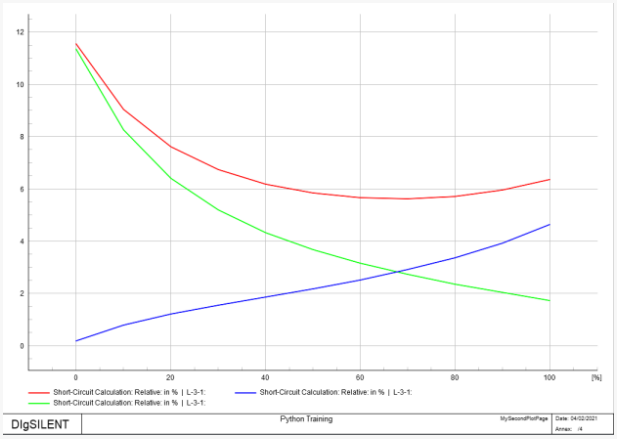
Contents: Plots.
Graphic Board.
Create a plot and change its variables.

7.1 Presentation: Graphical Representation

Notes:

Plots and Diagrams in PowerFactory

- Focus on new plot structure since PowerFactory 2021
  - New relevant class names are GrpPage, Plt…
  - Old structure had class names SetVipage, VisPlot… The old style is still available it is enabled in the project settings.
- There are several plot types available
  - The most common use of a plot is to look at the results of a time-domain simulation like an EMT or RMS simulation, by defining one or more plotted curves showing the variables changing with time.



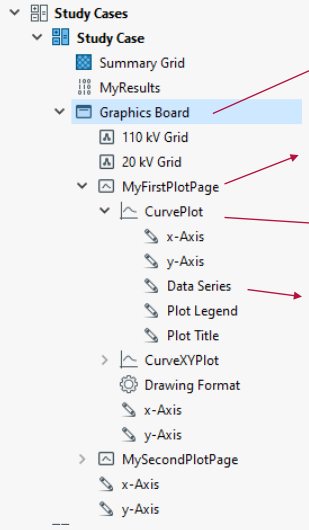
Graphical Representation 2

Anatomy of Plots in *PowerFactory*

- PowerFactory displayed graphics are dependent on the **active study case**
  - Enables customisation of displayed graphics per Study Case. Each study case contains a **graphic board** with the information about which graphics and plots are displayed.
- **Employed Concepts/Objects**:
  - "**Graphics Board**" object (SetDesktop)
    - Unique object per study case holding all graphic information
  - "**Plot Page**" objects (GrpPage)
    - A customisable page within the Graphics Board
    - Multiple Pages within a Graphics Board
  - "**Plots**" (PltLinebarplot)
    - The specific object containing the actual graphic/plot.
    - Multiple plots within a Page
    - Parent object of several plot objects like legends, data series, axis and title
  - „**Data series**" (PltDataseries)
    - Contains the actual link to the data to be displayed

Graphical Representation 3

Anatomy of Plots in *PowerFactory*



Graphics Board (SetDesktop): *graphicsBoard = app.**GetGraphicsBoard**()*

Plot Page (GrpPage): *plotPage = graphicsBoard.**GetPage**("MyFirstPlotPage", 1, "GrpPage")*

Plot (PltLinebarplot): *curvePlot = plotPage.**GetOrInsertCurvePlot**("CurvePlot",1)*

Data series (PltDataseries): *dataSeries = curvePlot.**GetDataSeries**()*

*dataSeries.**AddCurve**(obj1, „variable1")*
*dataSeries.**AddCurve**(obj1, „variable2")*
*dataSeries.**AddCurve**(obj2, „variable1")*
*dataSeries.**AddCurve**(obj3, „variable1")*

Scale and Export of Plots

- Scaling works on different levels
  - Graphics board: **DoAutoScaleX()**
  - Page: **DoAutoScale(), DoAutoScaleX(), DoAutoScaleY()**
  - Plot: **DoAutoScale()**
- Exporting graphics:
  - Export page from the graphics boards with **WriteWMF()**, exports the currently open graphic in the graphics board to a WMF figure.

    SetDesktop.WriteWMF(…)

  - **ComWr** is a command that can be used to export graphics to different data formats

    ComWr = app.GetFromStudyCase(“ComWr”)
    ComWr.SetAttribute(…)
    ComWr.Execute()
- More Methods for the different plot types are listed in the Python scripting reference under the according class names

7.2 Plots

In this exercise the variables of a results file should be displayed in a diagram.

- Import and activate the project *PYT_Results_Start.pfd*
- Verify that the Result File (*ElmRes*) 'MyResults' inside the Study Case has variables and results.
- If 'MyResults' is empty, execute the *Short Circuit Sweep.ComPython* script available in folder *Library* → *Scripts* (Update the link to the .py File WriteRes.py if necessary)

7.2.1 Creating a Curve Plot

- Create a new Python script called 'CreatePlots' in the project folder *Library* → *Scripts*
- Create a Python file (*.py*) in the external editor and name it 'Create_Plot.py'
- Create a new plot page called "MyFirstPotPage" in the Graphic Board with your script.
- Add a curve plot to the page (PltLinebarplot)
- Get the data series object from the plot and display a curve for Short-Circuit Current of the line (m:Ikss:busshc)
- Scale the x- and y-axis of the plot automatically to display the graph.
- After the script execution the diagram should by displayed like shown in the first diagram of figure 7.1.

Hint: When executing the script several times, it is helpful to use the PltDataseries.ClearCurves() function in order to delete old data in the plots

---

Part of the code of the Create_Plot.py file

```
graphicsBoard = app.GetGraphicsBoard()
plotPage = graphicsBoard.GetPage("MyFirstPlotPage", 1, "GrpPage")
curvePlot = plotPage.GetOrInsertCurvePlot("CurvePlot",1)
dataSeries = curvePlot.GetDataSeries()
```

Note: The command “Draw()” updates all plots that display values from the result object. Instead of using “Write()” for writing current results and “Draw()” for updating plots separately when doing a short circuit sweep, the command “WriteDraw()” can be used.

7.2.2 Creating a Curve Plot with dedicated x-Axis

Now, extend your script in order to display an additional plot on the same plot page.

- Add a curve XY plot to the page.
- Get the data series object from the plot and display a curve for Short-Circuit Current of the line (m:Ikss:busshc) over the short circuit position (ppro of the short circuit command)
- Scale the x- and y-axis of the plot automatically to display the graph.

The plot page should look like shown below.

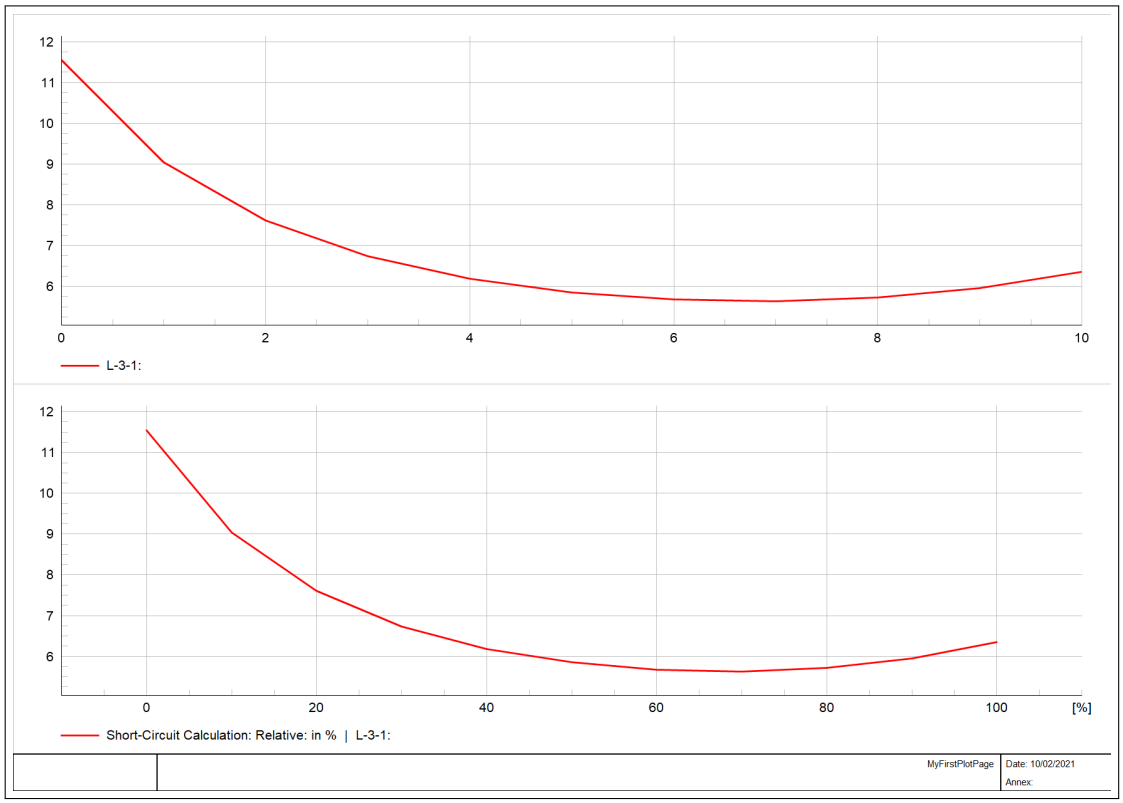


Figure 7.1: Plot page with two curve plots

Hint: You can use the SetLayoutMode() on the plot page in order to arrange the plots differently on the page.

7.2.3 Adding plots on additional page

- Add a second page to the graphics board and name it “MySecondPlotPage”.
- Add a new plot curveXYplot and display the curve of several variables. In this case, compare Short circuit current from bus1, bus2 and busshc of the line in the same plot over short circuit position (see Figure 7.2)

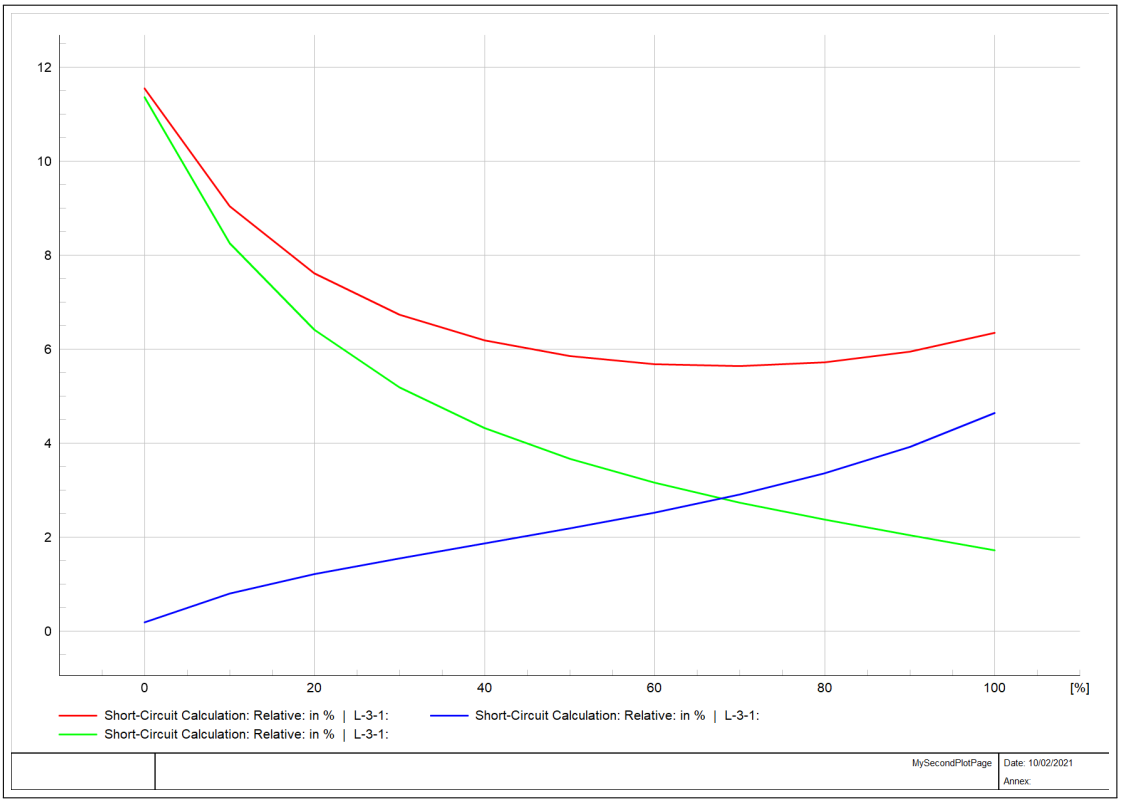


Figure 7.2: Plot with several curves

7.2.4 Exporting the plots from *PowerFactory*

Plots are commonly used for reporting results. Plots can be manually exported by selecting one of the exporting formats under *File* → *Export Graphic...*. This task can be automatised for the wmf-file using the Python function WriteWMF().

- First create an empty folder on your Desktop and name it “PlotsFolder”
- Create a new script, import the *PowerFactory* module and define an application object.
- Access all plot pages (*.GrpPage) from the Graphics Board and place them into a list.

```
gb = app.GetGraphicsBoard()
pages = gb.GetContents('*.GrpPage')
```

- Loop through the list of plot pages and export each page into a wmf file, by using the WriteWMF() function.

```
for page in pages:
    gb.Show(page)
```

```
3     file_name = (r'C:\Users\Desktop\PlotsFolder' + '\\' + page.loc_name)
4     gb.WriteWMF(file_name)
```

The method WriteWMF exports any graphical representation, that is active (.Show()) on the graphical board, into a wmf file.

Hint: You can use the ComWr-Command to export to other data formats

---

PowerFactory files

File Name	Description
PYT_Results_Start.pfd	Initial network with result file to be used in the exercise
Create_Plot.py	Python script-file, solution of exercise 7.2.2
GraphExport.py	Python script-file, solution of exercise 7.2.4
PYT_Plot_Finished.pfd	Solution of exercise 7.2

Additional Exercises

8 AddOn Module, Creating Objects and User Communication

Purpose: Using the Addon module and different dialogue boxes to communicate with *PowerFactory*. Creating a new object in *PowerFactory* via script

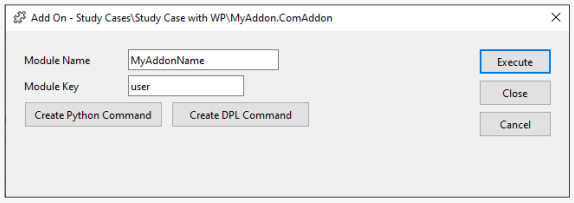
Contents:
Addon functionality in *PowerFactory*
Selection browser
Dialog boxes
Modify command attributes.
Creating Objects

8.1 Presentation: AddOn Module, Create Object and Communication

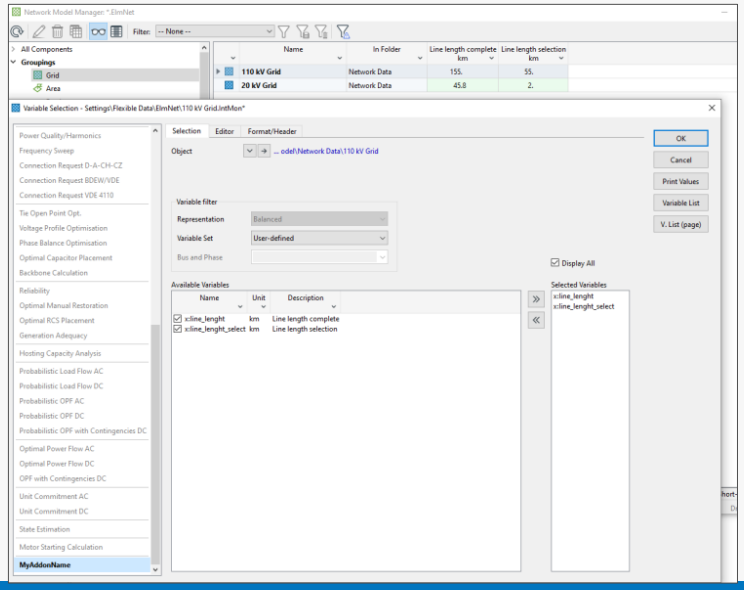
Notes:

AddOn Module

- Purpose: More flexibility in the processing of calculations and the presentation of results
- Allows to define and use User-defined variables (Alternative to data extensions)
- A script is mandatory to execute a ComAddon
  - Manual definition in the study case and placing a script inside the ComAddon
    - Variable definitions can be done manually by Creating a IntAddonvars inside the ComAddon
  - Create and set up the ComAddon directly via script
  - Script needs a „CreateModule“
  - The “FinaliseModule” command completes the Add On module and makes the results write protected



AddOn – User defined variables



AddOn – CreateModule and Adding Variables

Adding variables via script

 # Create AddOn

```
ComAddon = app.GetFromStudyCase("MyAddon.ComAddon")
ComAddon.DeleteModule()
ComAddon.name = "MyAddonName"
ComAddon.CreateModule()
```

 # Define Variables

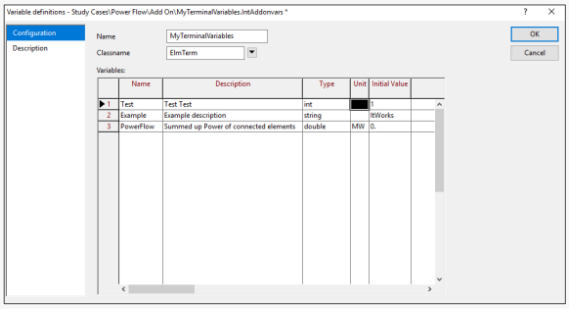
```
ComAddon.DefineDouble(ClassName, VariableName,
Description, Unit, IntialValue)
```

Also: Integer, matrix, vector, string… (See Python Reference)

 # FinaliseModule

```
ComAddon.FinaliseModule()
```

Adding variables via IntAddonvars



 # Create AddOn

```
ComAddon.CreateModule()
```

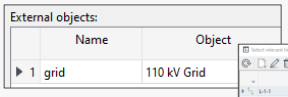
 # FinaliseModule

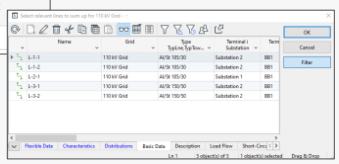
```
ComAddon.FinaliseModule()
```


User Dialog

- Input parameter / External objects (Script Object)

```
script = app.GetCurrentScript()
externalObject = script.GetAttribute("MyExternalObject")
```



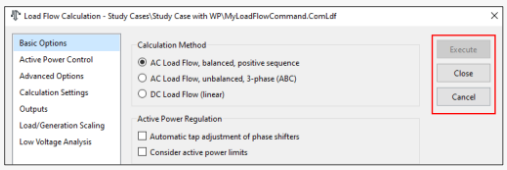


- Modal Browser / Modal Selection Browser (List of objects)

```
L_selected_obj = app.ShowModalSelectionBrowser("L_lines", "Please select relevant lines")
```

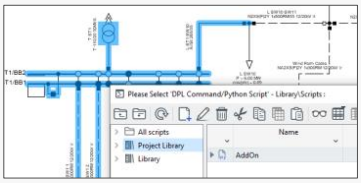
- Edit Dialogs (Commands)

```
ComLdf = app.GetFromStudyCase("MyLfd.ComLdf")
ComLdf.ShowEditDialog()
ComLdf.Execute()
```



- Diagram Selection (Diaragms)

```
L_objects = app.GetDiagramSelection()
```



- Browser Selection (Data Manager)

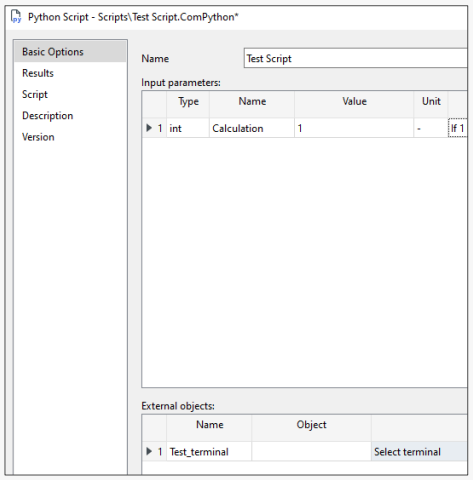
```
L_objects = app.GetBrowserSelection()
```

User Dialog – Implementing checks for correct/expected entries

```
script = app.GetCurrentScript()
error, value = Script.GetInputParameterInt("calculation)
if error:
   value = 1 #default is Ldf
```

```
script.GetAttribute(„terminal")
-> Error, No Attribute terminal
```

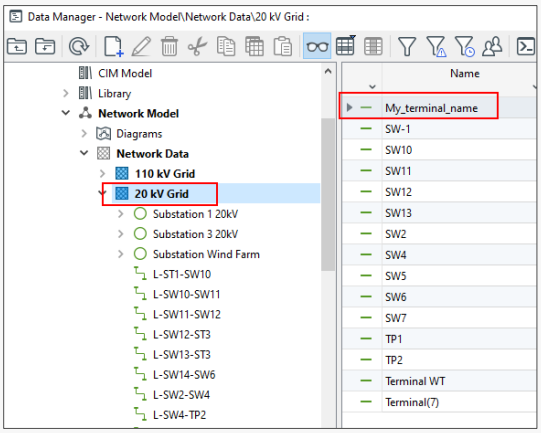
```
if script.HasAttribute („Test_terminal"):
   term = script.GetAttribute(„Test_terminal")
else:
   app.PrintError(„External Object ...")
   exit()
Unom = term.uknom
-> Error, None object has no attribute …   (if script.HasAttribute („Test_terminal") and script. Test_terminal != None: …)
```



Creating Objects via Script

- Objects are created in a parent object, like a study case, a project folder, a grid, substation,…

```
Grid = app.GetCalcRelevantObjects(„20kv Grid.ElmNet")[0]
MyTerminal =  Grid.CreateObject( "ElmTerm", "My terminal name")
```



- Commands:
  - CreateObject()
  - AddCopy()
  - PasteCopy()
  - …

8.2 AddOn-Module

In this chapter the Addon module is introduced and we will learn how to work with user defined variables. Also the modal selection browser is used, as well as the edit dialogue of commands, to allow a user interaction during the execution of a script. Additionally, we will use an external object and handle all input possibilities with if conditions to grantee a smooth execution of the script without unhanded exceptions.

8.2.1 ComAddon

In the first part of the exercise we will set up a AddOn module and set a user defined variable via script.

- Import the project “PythonTraining_Start.pfd” and activate it.
- Create a new ComPython object and name it “AddOn”.
- Create a Addon-Modul via script and define a the variable for the total line length for grids.
- Define the external object “grid” in the “ComPython” to allow the user an selection of the grids to investigate. The external object should allow the entry of grids (ElmNet) and general sets (SetSelect) containing references to grids. If nothing is selected all grids should be considered.
- Loop through the lines of each grid and sum up the lengths of the lines in service.
- Write the resulting length for each grid into the defined parameter.

Hint: When using GetCalcRelevantObjects() on grids, keep in mind that the summary grid will also be returned. The summery grid object is placed in the study case and sums up results of the single grids. It does not contain any network elements like lines and also does not possess some attributes like the nominal frequency

---

Hint: The FinaliseModule() is optional. If FinaliseModule() is called the entered values become write protected. Also it is recommended to delete the old module (ComAddon.DeleteModule()), when executing the script several time to avoid potential confusion of variables.

8.2.2 Modal Selection Browser

In addition it shall be possible for the user to select certain lines for each grid to be summed up. Modify your code to allow an interactive user selection with the “Modal Selection Browser”.

- Use an “Input parameter” to check whether a selection browser shall be shown. Make sure only the desired inputs are allowed. Otherwise rise an warning or error message.
- Define a new variable for the summed up length of the selected lines in the ComAddon.
- Show the “Modal Selection Browser” for each grid to allow the user to select certain lines.
- Write the resulting length of the selected lines to the defined variable for each grid.

8.2.3 Create a new Object

In this part of the exercise we will create a new area and use a diagram selection to select the elements to be grouped into the created area. Then we will execute a load flow and calculate the average voltage of the area. Before executing the load flow via script we will show the command dialog to allow the user to modify the load flow settings.

- Get a selection of network elements from the single line diagram.
- Filter the list of objects for terminals and assign them to the created area.
- Implement a check to make sure that terminals are selected. If this is the case, create a new area, name it “Area from diagram selection” and add the terminals to the new area
- Define a new variable for the average voltage in pu for areas
- Execute a load flow and show the command dialog to allow the user to modify the load flow settings
- Calculate the average voltage of the main busbars of the created area and write the result to the defined variable.

Hint: When creating a new object you should navigate to the appropriate parent object. It can be helpful to create a object manually and check in the data manager where it was created.

---

keywords:
CreateObject(), GetDiagramSelection(), ShowEditDialog()

PowerFactory files

File Name	Description
PythonTraining_Start.pfd	Network used for this exercise
PythonTraining_Finish.pfd	Solution for this exercise
01_AddOn.py	Python script-file, solution of exercise 8.2.1
02_AddOn_select.py	Python script-file, solution of exercise 8.2.2
03_AddOn_dialog.py	Python script-file, solution of exercise 8.2.3

9 Engine Mode

Purpose: Running *PowerFactory* in Engine Mode.

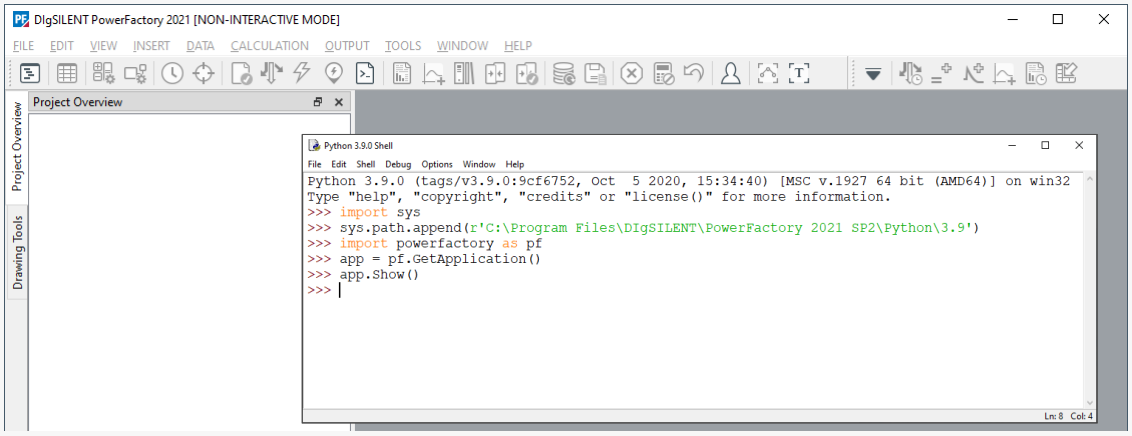
Contents: basic of how to start and execute some commands from Engine Mode.

9.1 Presentation: Engine Mode

Notes:

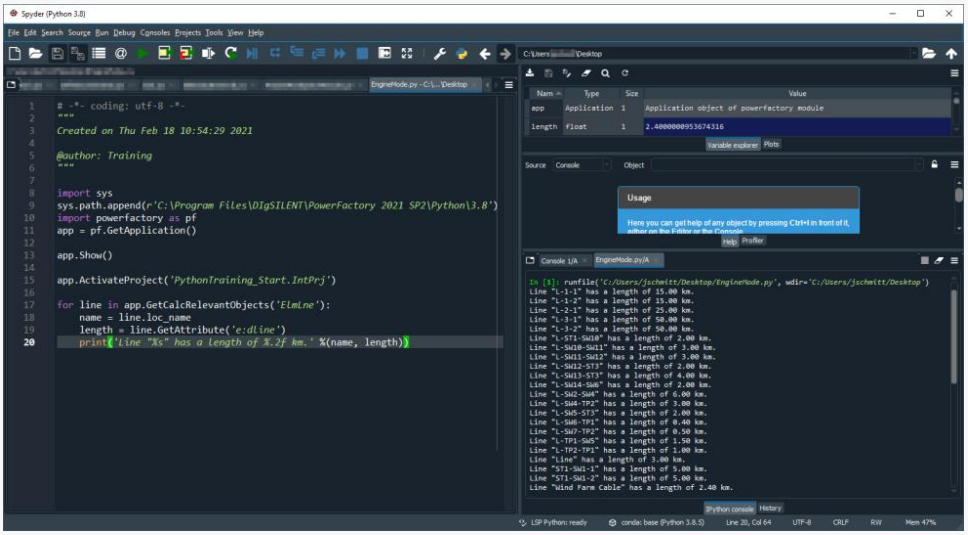
Engine Mode

- *PowerFactory* may be run externally by Python.
- Script must additionally import the file path to the dynamic module (“powerfactory.pyd“)



Anaconda

- *PowerFactory* can also be started in Non-Interactive-Mode from the Anaconda Console.



9.2 Running *PowerFactory* in Engine Mode

To be able to run *PowerFactory* externally by using Python first what has to be done is to import the dynamic Python module ("powerfactory.pyd"), which interfaces with the *PowerFactory* Application Programming Interface. This module 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 build-in functions (DPL functions)

To import the powerfactory module, the Python interpreter has to know where the module is located. To do this first import sys module. sys module provides access to interpreter and to functions that interact strongly with it. On this way you have a possibility to append the file path to the dynamic *PowerFactory* module.

```
1 import sys
2 sys.path.append(r"C:\Program Files\DIgSILENT\PowerFactory 20xx\Python\3.x")
```

Under sys.path you can find a list of strings that specifies the search path of the interpreter for modules. After path is being added to sys.path list powerfactory module can be imported. After importing powerfactory module all rules for coding are the same as for coding inside of *PowerFactory*. That means you can gain access to the *PowerFactory* environment by adding:

```
1 import powerfactory as pf
2 app = pf.GetApplication('UserName', ['Password'])
```

Note: "UserName" represents the name of the user, which will be activated, after the "GetApplication" method has been successfully executed. If existing, also a password has to be passed.

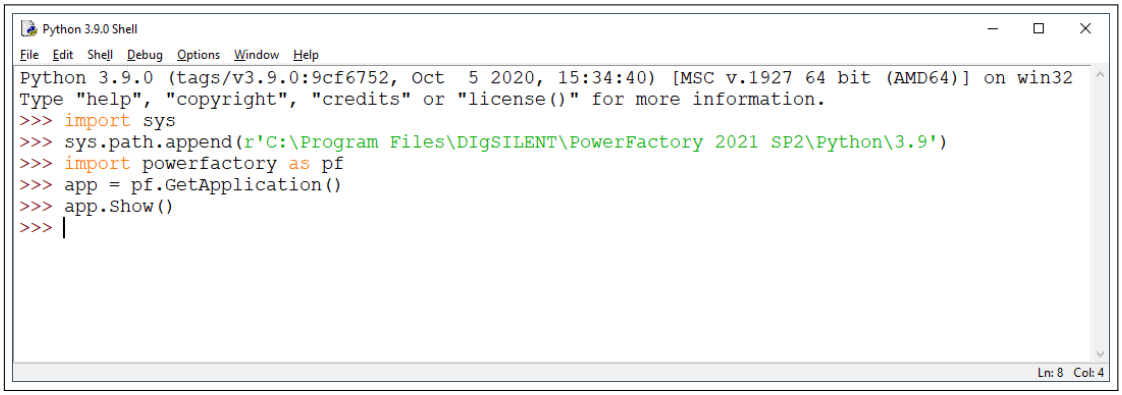


Figure 9.1: Python Shell

By calling dir() function for the Application object "app" you can get overview over methods you can call on app.

```
dir(app)
```

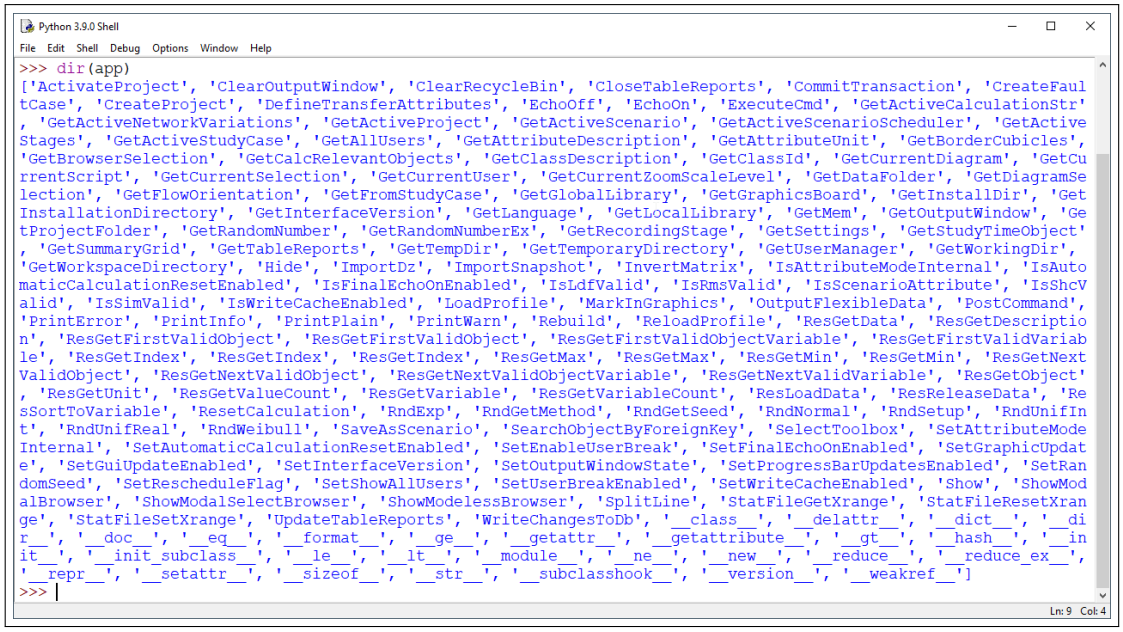


Figure 9.2: List of methods applicable on Application object app

A list of methods that can be called on app object are available on the figure 9.2, for example method **Show()**, that will make the *PowerFactory* window visible.

```
app.Show()
```

Recapitulation for starting running *PowerFactory* in Engine Mode:

- import sys module
- append the path to the powerfactory.pyd
- import powerfactory module
- call GetApplication() function
- write your Python code

Note: If an error message appears when importing the powerfactory module stating “DLL load failed: the specified module could not be found”. This means that Microsoft Visual C++ Redistributable for Visual Studio 2012 package is not installed on the computer. To overcome this problem the user should add the PowerFactory installation directory to the os path variable within his python script.

```
import os
os.environ["PATH"] = r'C:\Program Files\DIgSILENT\PowerFactory 20xx;'
                                                + os.environ["PATH"]
```

9.3 Example 1

Task is to open existing project inside of *PowerFactory*, to execute Load Flow calculation and represent the result values in Python.

- Open Python Shell window.
- Import **sys** module and add powerfactory module path.
- Import powerfactory module and call **GetApplication** method.
- Make sure that *PowerFactory* is open in Engine Mode.
- Open an existing project (*.IntPrj). You can use here **ActivateProject**.
- Call Command Object ComLdf and Execute load flow calculation
- With **GetCalcrelevantObjects()** method assign all line objects (*.ElmLne) to the list “lines”
- Print the results of loading for all lines in Python Shell output Window.

```
import sys
sys.path.append(r"C:\Program Files\DIgSILENT\PowerFactory 20xx\Python\3.x")
import powerfactory as pf
app = pf.GetApplication('<UserName>')
app.Show()
proj = app.ActivateProject('Python_Start.IntPrj')
ldf = app.GetFromStudyCase("ComLdf")
ldf.Execute()
lines = app.GetCalcRelevantObjects("*.ElmLne")
for line in lines:
        value = line.GetAttribute('c:loading')
        name = line.loc_name
        print('Loading of the line: %s = %.3f' %(name,value))
```

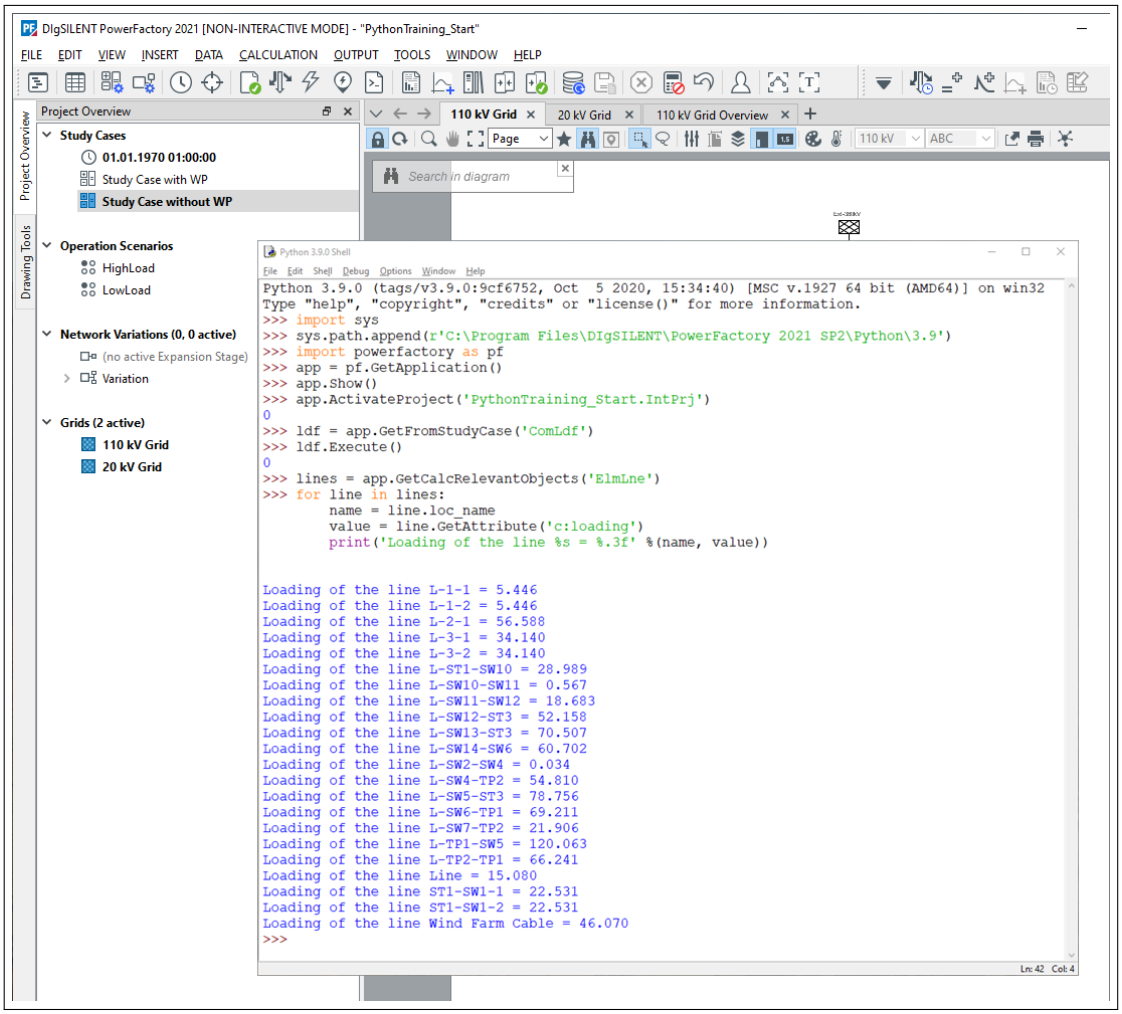


Figure 9.3: Python Shell window

9.4 Run *PowerFactory* from Command Prompt window by using Python

To be able to access Python through Command Prompt window, the command window has to recognise the word “python”. To check if this is the case write python in command window

```
c:\Users \UserName> python
```

If the command window does not recognise the word “python” (see Figure 9.4) the following message “*python is not recognized as an internal or external command, operable program or batch file*” will be printed.

Figure 9.4: Command Prompt window

This means that Python is not added to the DOS path. This is not done by default and has to be done only once manual.
To be able to change DOS path you will have to do following:

- Go to **System Properties** and select **Advanced system settings** (see Figure 9.5)

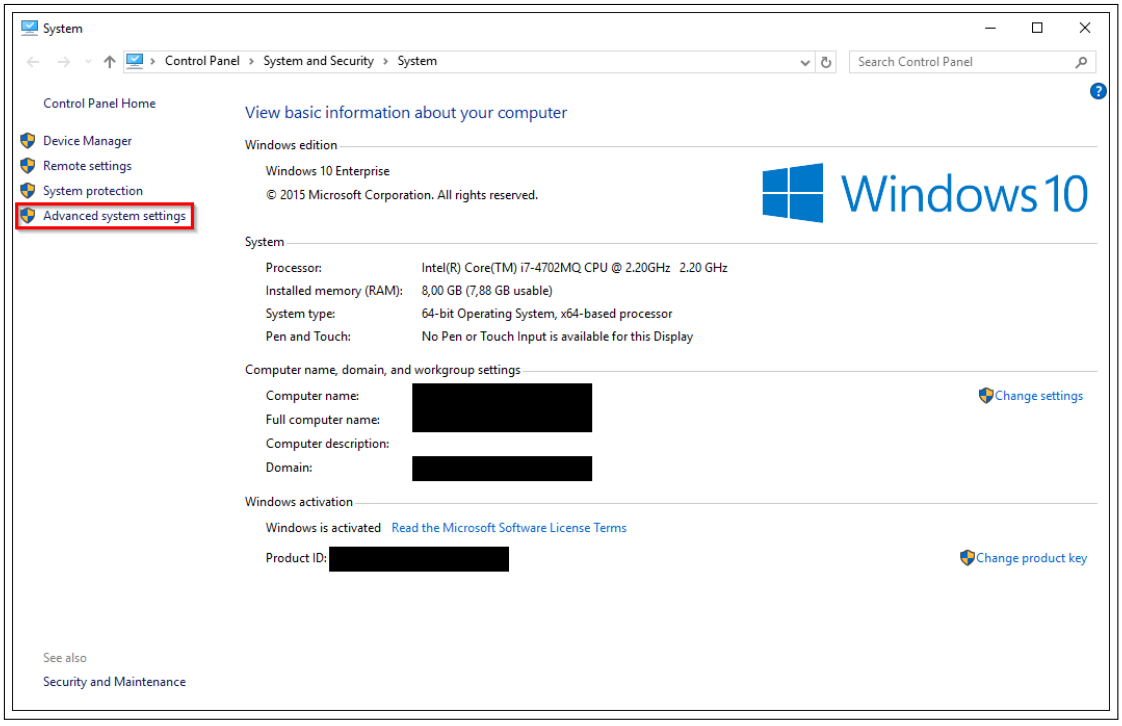


Figure 9.5: System Properties

- After selecting **Advanced system settings** new window will open. Select **Environment Variables** and scroll the **System variables** section till the **path** variable is not reach.

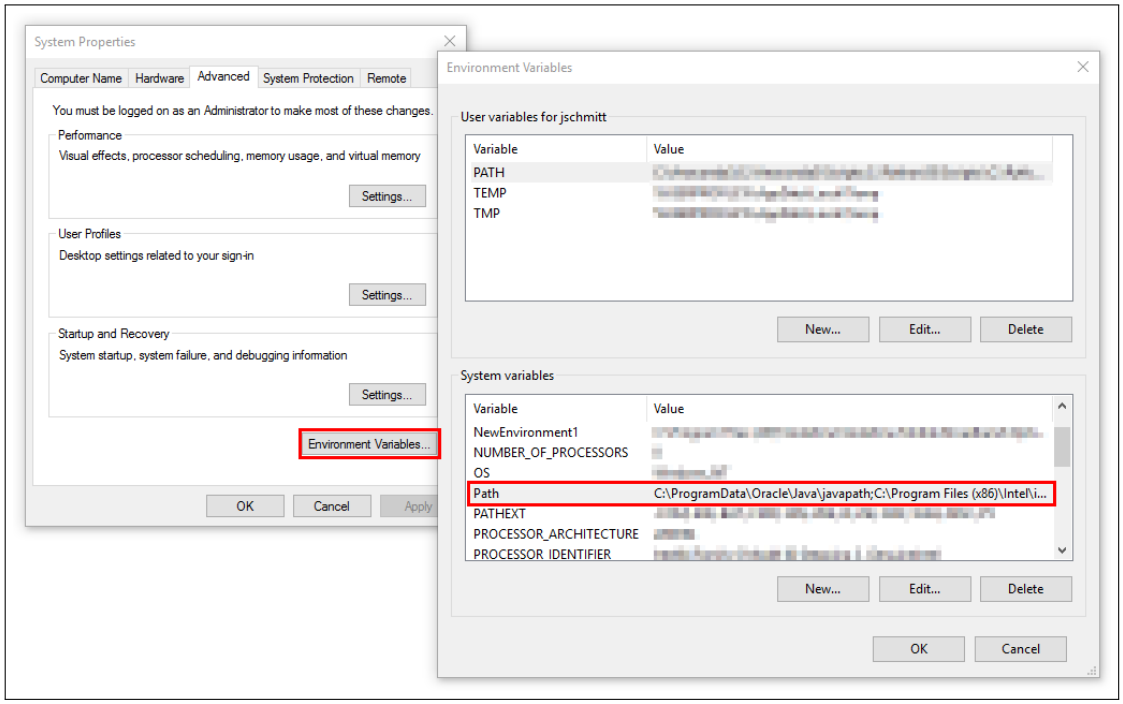


Figure 9.6: Step 2

- Add Python installation path (e.g C:\Python39). Separator between two neighbour paths ";" is obligatory

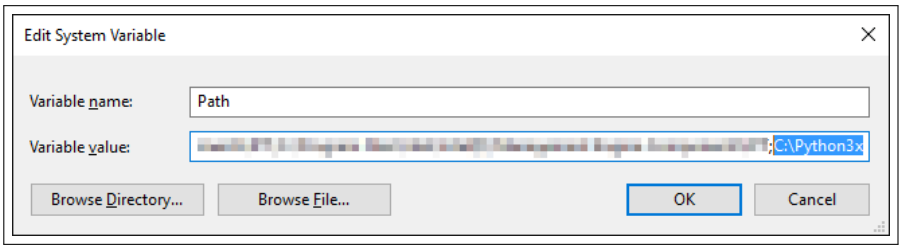


Figure 9.7: Changing the DOS path

- After the procedure has being done, open Command Prompt again and write "python". If no mistake has been done python will be recognised and opened (see Figure 9.8)

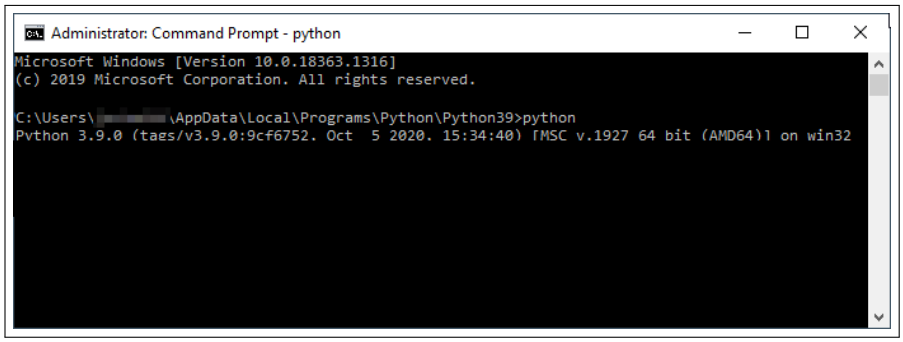


Figure 9.8: Command Prompt window with open Python interpreter

9.4.1 Example

Start *PowerFactory* out of Command Prompt window.
Task:

- Start Command Prompt and test if Python will be recognized and run from Command Prompt.
- Import *PowerFactory* module in Python and call GetApplication() Method as usual.
- Call .Show() method to see *PowerFactory* window

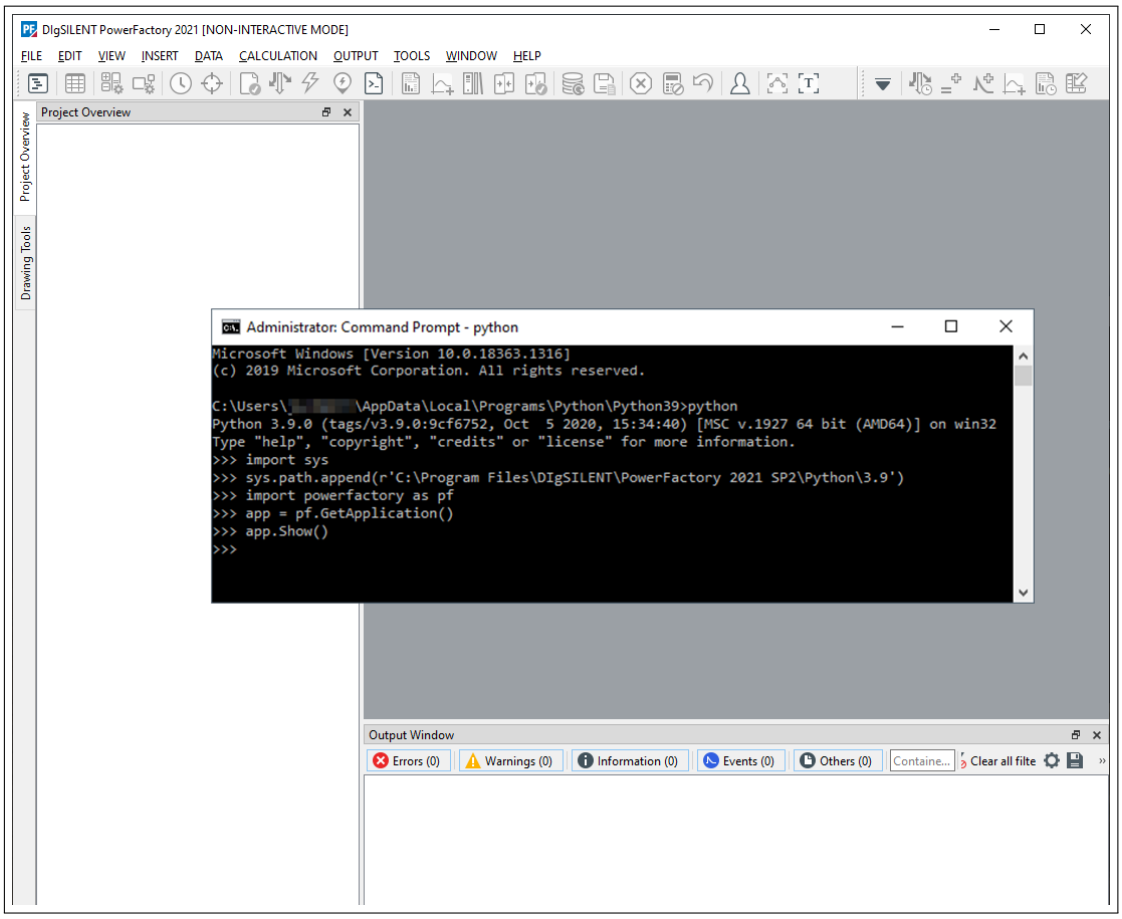


Figure 9.9: Command Prompt window with open PowerFactory window

10 External File Access

Purpose: Access an external file to read data from it and use it in *PowerFactory* or to write data from *PowerFactory* into the file.

Contents: Write in an external csv file.
read from an external csv file.

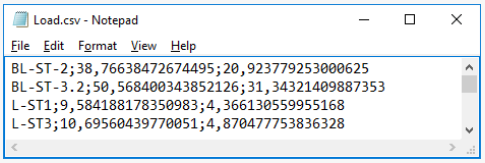
10.1 Presentation: External Files

Notes:

File Access

- Sometimes it is necessary to have access to external data files, that contain information a user could need.
- file = **open**(filename, mode) → *function opens the file and returns a file object*
  - mode: 'w', 'r', 'r+',…
- file.**read**(), file.**readline()** → *returns the content (complete or one line) of the file*

```
file.read()

for row in file:

    print row
```



- file.**write**(string) → *writes the content of the string to the file*
- file.**close()** → *closes the file*

External Files 2

10.2 File Access

Writing output data to files and reading input data from files is an important part of software interfaces.

In this exercise a simple file-based interface will be created. In the first part we will use Python to write data from *PowerFactory* to a Comma Separated Values (*.csv) file, and in the second part we will read the data from a Comma Separated Values file and write it into *PowerFactory*.

10.2.1 Write to File

The Python script created in this exercise writes all load values in a .csv file.

- Create a new Python script called 'Write to File' in the project folder *Library* → *Scripts* of the project 'Python_Start'.
- Create a Python file (*.py*) in the external editor and name it 'WriteFile.py' and link it to the Python script 'Write to File'
- The following operations should be carried out by the script:
  - The path/location of the csv file should be given as an input inside of the Python script.

```
fobj = open("E:\\PYT_08_ExtFiles\\CSVFile.csv","w")
```

    or

```
fobj = open(r"E:\PYT_08_ExtFiles\CSVFile.csv","w")
```

    The "r" in front of a string tells Python to read the following string as it is. So the double backslash is not needed.
    If at the given location no csv file with the name *CSVFile* exists, it will be automatically created.
  - It is also possible to use just the name of the file without a path. In this case the folder of the .py-Script is used.

```
fobj = open("CSVFile.csv","w")
```

  - The file located in the corresponding path should have the name 'Load.csv'.
  - Open the file in 'write'-mode.
  - Write the active and reactive power of all loads into the file together with an identifier for the load. For example:

```
fobj.write('{};{};{}\n'.format(Load.loc_name,str(Load.GetAttribute
                ("m:P:bus1")),str(Load.GetAttribute("m:Q:bus1"))))
```

  - The format of the file should be Comma Separated Values (*.csv).
  - After writing the csv file do not forget to close it.

```
fobj.close()
```

- Open the file with Excel or a text editor and check the values.

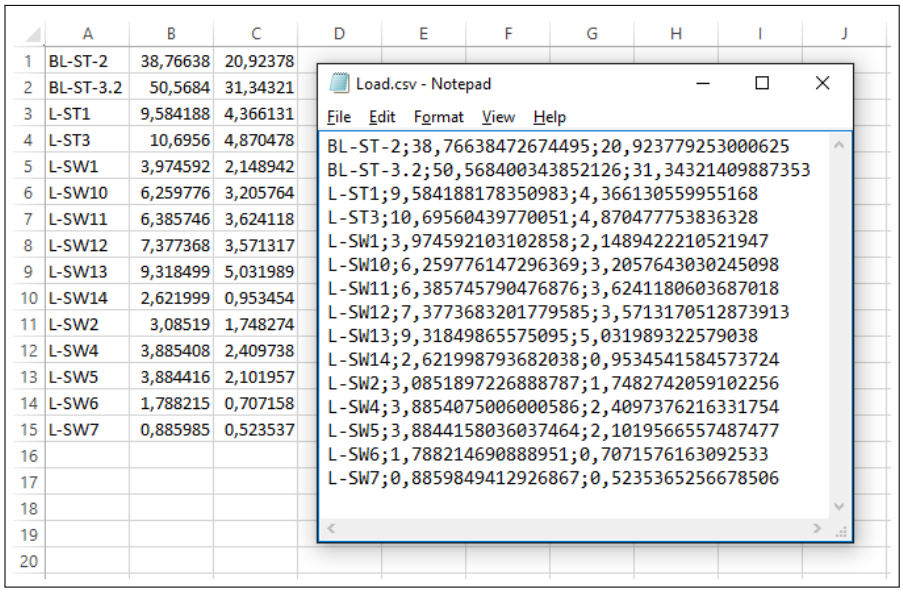


Figure 10.1: Output of the script 'Write To File' shown in Excel and in Notepad

10.2.2 Read from File

The Python script created in this exercise reads the values from the Comma Separated Values (*.csv) file and sets the values read in the file to the to the corresponding object in *PowerFactory*.

In order to perform this task, the file from the previous exercise will be modified first:

- Save the Operation Scenario as 'High demand'
- Open the 'Load.csv' file created in Exercise 10.2.1 and increase the value of Active and Reactive Power by 10%
- Save the file as 'LoadIncreased.csv'

Then a new script will be created:

- Create a new Python script called 'Read from File' in the project folder *Library* → *Scripts* of the project 'Python_Start'.
- Create a Python file (*.py*) in the external editor and name it 'ReadFile.py' and link it to the Python script 'Read from File'
- The following operations should be carried out by the script:
  - Open the file at a location specified as an input parameter of the script in 'reading'-mode.

```
1        csvFile = open("CSVFile.csv","r")
```

  - After file has been opened read the contents of the file and define used delimiter.

```
1        csvFileContents = csv.reader(csvFile,delimiter=';')
```

  - Use a **for** loop to read/access each row of the csv file one after another.

```
1          for row in csvFileContents:
```

All data read in this way are placed in a list of strings. Each row contains the following data:
['Name', 'P_value', 'Q_value']

- Read the first string element to get the load name.
- Search the first load in the list of all calculation relevant loads in the project that has the same name as defined in *row[0]*.
- Read the second and third element from the list *row* and convert these strings into double values.
- Assign these values to the loads parameters *plini* and *qlini*.
- Print the load name and the power values into the output window for control.

PowerFactory files

File Name	Description
Python_Start.pfd	Network used for this exercise
WriteFile.py	Python script-file, solution of exercise 10.2.1
ReadFile.py	Python script-file, solution of exercise 10.2.2
PYT_08_ExtFiles_Finished.pfd	Solution of exercise 10.2

11 Graphical User Interface (GUI)

Purpose: Get familiar with graphical user interface build in library that Python offers, and implementation of the this library in *PowerFactory*.

Contents: tkinter module.
Tkinter objects widgets.

11.1 Presentation: Graphical User Interface

Notes:

GUI

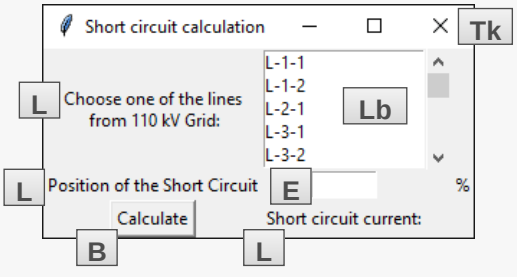
- Purpose of the GUI is to make the usage of the stored script easier and efficient
- It makes possible to define a script, that outer users can use without knowing the code behind the script.
- The **tkinter** module is the standard Python interface to the Tk GUI toolkit.[1]
- It is included with Python as a library and is easy to use

https://wiki.python.org/moin/TkInter

[1] https://docs.python.org/2/library/tkinter.html

tkinter

- Importing
  from tkinter import *
- Panel=Tk() - Panel is now parent window, where all other objects will be placed **Tk**
- Tkinter objects:
  - Button **B**
  - Label **L**
  - Entry **E**
  - Listbox **Lb**
  - …
- .grid(arg1,arg2) method places objects on application screen.
- .mainloop() method must be called, generally after all the static objects are created, to start processing events.



11.2 Input and Output GUI

Graphical User Interface makes the handling of the scripts easier. This makes possible to for example, use scripts, change inputs and print results without knowing the code behind the script.

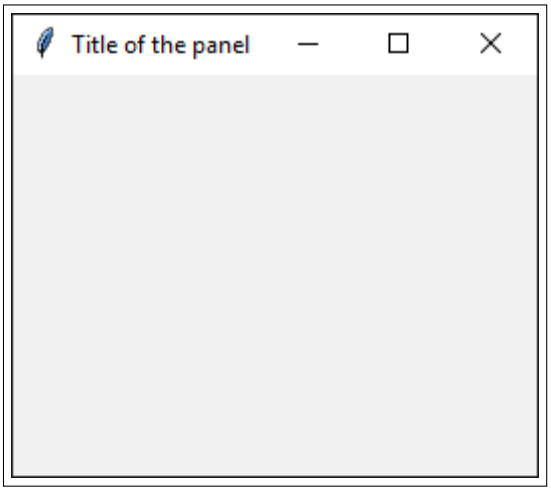
In this exercise special attention will be put on Graphical User Interface usage in *PowerFactory*. It will be shown how to program a script that creates a GUI for inputting parameters and executing some commands.

11.2.1 Basic use of tkinter module

Since Tkinter is a built-in module the user does not have to install any additional python package. To use it first you will have to import the tkinter module. After importing the module you may start building your GUI by creating an empty window for user/script communication.

```
import tkinter
Panel = tkinter.Tk()
Panel.title('Title of the panel')
```

Results of the code:

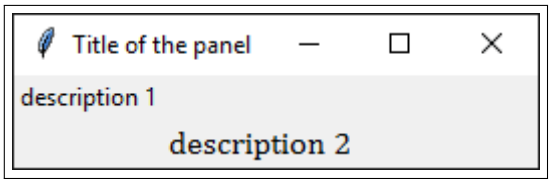


We will present you some of the basic widgets (tkinter objects) that you will need for this exercise. For more information on this please refer to official python tkinter literature.

- **Label**

  A Label contains some written information. This can be the description of some fields in order to inform other users what they should input or select,.. To define a Label widget

```
label1 = tkinter.Label(Panel,text = 'description 1')
label1.grid(row = 0,column = 0)

tkinter.Label(Panel,text='description 2', font='Cambria').grid(row=1,
```

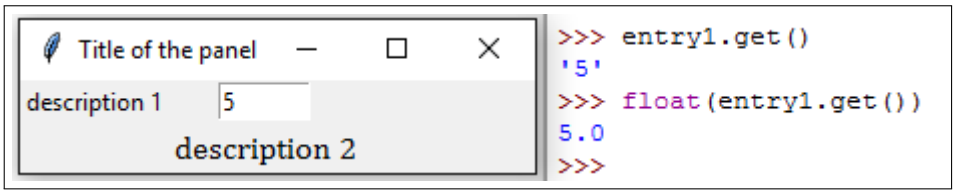


Where a widget should be placed on a tkinter panel is defined by using the method **.grid()** with two input parameters row and column position (starting from 0,0)

- **Entry**

  Entry is the first widget allowing us to enter some value that can than later be used in our script. To define the Entry just call the Entry function and as input pass the tkinter.Tk object (Panel) and the width of the entry field. To get the entered value use the **get()** method of the entry widget. Note that the get() function always returns a string.

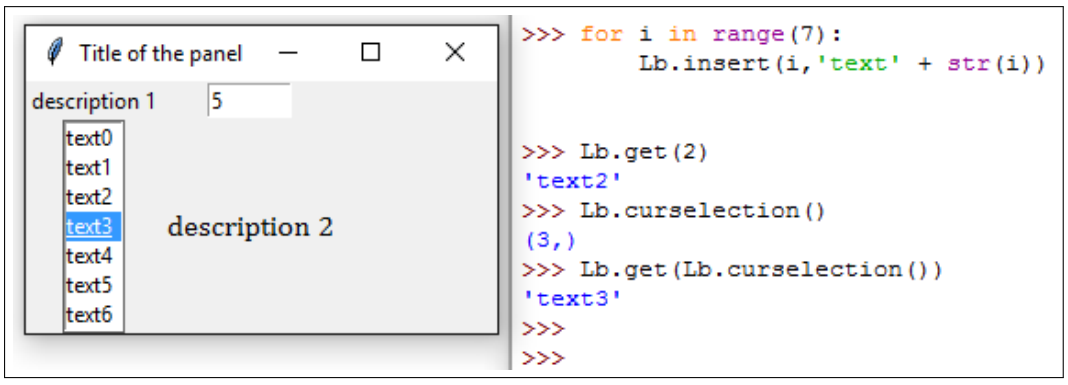
```
entry1 = tkinter.Entry(Panel, width = 7)
entry1.grid(row = 0, column = 1)

entry1.get()
```



- **Listbox**

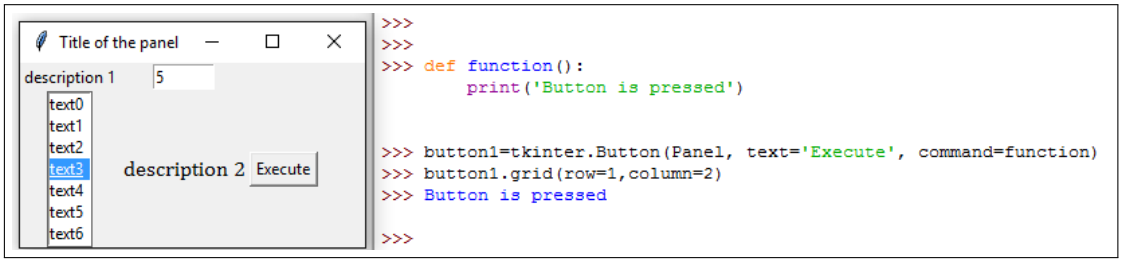
  A Listbox contains a list of options that the user can select. To insert a value in the Listbox use the method **insert(position,value)**, to get values from the Listbox use the method **get(position)**, to get the position of the selection made by a left mouse click use the method **curselection()**.

```
Lb = tkinter.Listbox(Panel, width = 5, height = 7)
Lb.grid(row = 1, column = 0)
```



- **Button**

A Button widget calls a function that should be executed after pressing the button. This function should always be defined before the button widget is created.



11.2.2 Creating the Graphical User Interface

Following tasks should be done:

- Create a new Python script called 'Graphical User Interface' in the project folder *Library* → *Scripts*
- Create a Python file (*.py*) in the external editor and name it 'GUI.py' and link it to the Python script 'Graphical User Interface'
- GUI input should consider:
  - **Input for the Line where the Short Circuit occurs**: a list of the lines available on the Network. The User should be able to select one line of the list, and so define where the short circuit should occur.
  - **Input of position of Short Circuit**: an empty entry field where the position of the short circuit in % of the total line length should be entered.
  - **Execute button**: An execute button to start the Short Circuit Calculation.
  - **Label with results**: The results of the calculation should be available also on GUI.
- Script should take entries from the user and put them in corresponding fields of the Object *ComShc*
- After pressing the button **Calculate** the Short Circuit Calculation should be executed and the results should appear on the result label.

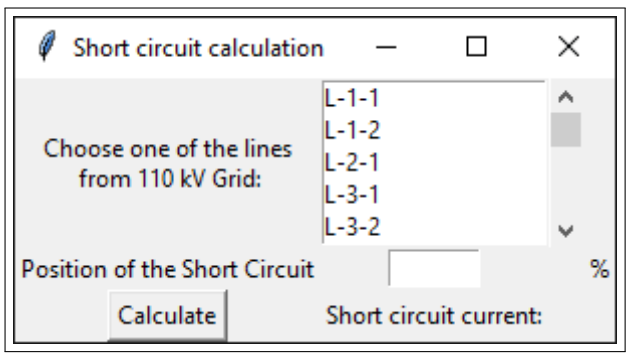


Figure 11.1: Possible Input Window

keywords:
tkinter, ClearOutputWindow, GetFromStudyCase, GetCalcRelevantObjects, get, replace, GetAttribute, Label, Tk, Listbox, Entry, Button, mainloop

Hint: Method **mainloop()** must be called, after all the static widgets are created, to start processing events. This is not needed if using tkinter in IDLE

11.2.3 Optional: Verification of input fields

If the User does not input all the parameters in the Input window or some for the parameters are outside the expected range, the GUI Application will not perform the calculation and will shut down the GUI Window or will block it.

Therefore, the idea of this exercise is to expand the python file 'GUI.py' so that the script limits the parameter range and also displays a message when there is an empty field reminding the user to insert the corresponding value, as shown in Figure 11.2

The input field "Position of the Short Circuit", should be limited according to the following criteria:

- Both separators for decimal numbers "." and "," are allowed
- Only numbers between 0 and 100 are permissible
- No string entering is allowed

For solving this issues there are many possibilities, for example:

- Use the **.replace()** method for the separators
- Use the **if** statement for the range checking
- Use the **try:** and **except:** statements string checking

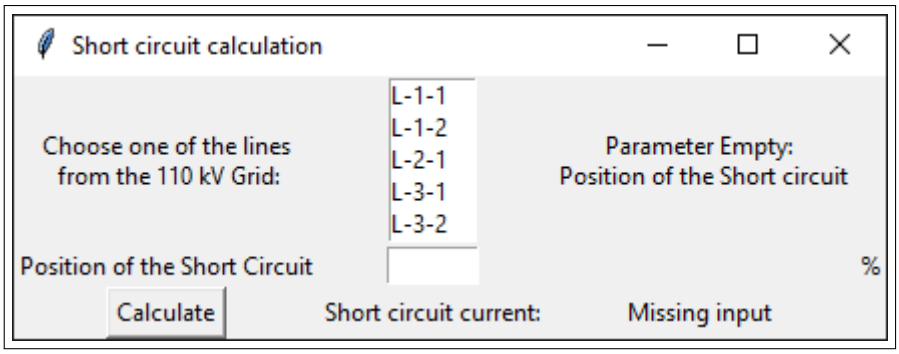


Figure 11.2: Verification of input fields

PowerFactory files

File Name	Description
PYT_Results_Start.pfd	Initial network with result file to be used in the exercise
GUI.py	Python script-file, solution of exercise 11.2.2
GUI_Expanded.py	Python script-file, solution of exercise 11.2.3
PYT_07_GUI_Finished.pfd	Solution of exercise 11.2